

# **Deliverable for I'mbesideyou** **Data Scientist Internship**

**Vishnu Tirth Bysani**

**Prompt Engineering Documentation**

## Prompt Engineering

Prompt engineering refers to the process of meticulously creating or crafting prompts to effectively instruct a language model like GPT-3.5 to generate desired responses. It involves finding the right combination of words and situational context to generate the specific information or type of response a user desires.

Effective prompt engineering can significantly influence the quality and relevance of the generated output. In an age where AI-driven solutions are becoming common—from chatbots in customer service to AI-powered content generators—prompt engineering is the bridge ensuring effective human-AI communication. It's not just about getting the right answer; it's about ensuring AI comprehends the intent, context, and the subtleties behind every query.

The aspects that make up a good prompt are:

- **Instruction.** This is the core directive of the prompt. It tells the language model what you want it to do.
- **Context.** Context provides additional information regarding the background of the instruction that you have provided which helps it to pinpoint what is being asked.
- **Input data.** This is the specific information or data you want the model to process. It could be a paragraph, a set of numbers, or even a single word.
- **Output indicator.** This is especially useful in role-playing scenarios, this element guides the model on the format or type of response desired. It tells the model what type of response is expected for the input given.


I have included in this report the prompts that I have used while making the EDA and Clustering report.

**Prompt:** How to plot a heatmap using seaborn library?

**Output:**

Here's an example code to create a heatmap:

python

 Copy code

```
import seaborn as sns
import matplotlib.pyplot as plt


# Generate some random data (replace this with your actual data)
data = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]

# Create a heatmap using Seaborn
sns.heatmap(data, annot=True, cmap='coolwarm')
```

With the help of this prompt, I was able to plot the heatmap of the emotion\_df, gaze\_df and transcript\_df dataframes. This gave a visual representation of the correlation between each column.

**Prompt:** How to delete last row from a dataframe ?

**Output:**



To delete the last row from a DataFrame in pandas, you can use the `DataFrame.drop()` method. You'll need to specify the index label (or index position) of the row you want to delete.

Here's an example:

```
python Copy code
```

```
import pandas as pd

# Create a sample DataFrame
data = {'A': [1, 2, 3],
        'B': [4, 5, 6]}

df = pd.DataFrame(data)

# Display the original DataFrame
print("Original DataFrame:")
print(df)


# Get the index label of the last row
last_row_index = df.index[-1]

# Delete the last row
df = df.drop(last_row_index)

# Display the DataFrame after deleting the last row
print("\nDataFrame after deleting last row:")
print(df)
```

Output:

less

 Copy code

Original DataFrame:

	A	B
0	1	4
1	2	5
2	3	6

DataFrame after deleting last row:

	A	B
0	1	4
1	2	5

In this example, we first create a sample DataFrame `df`. We then get the index label of the last row using `df.index[-1]`. Finally, we use `df.drop(last_row_index)` to delete the last row.

Alternatively, if you prefer to delete the last row based on its position, you can use `df.drop(df.index[-1])` directly.




Keep in mind that by default, `df.drop()` returns a new DataFrame with the specified row(s) removed. If you want to modify the original DataFrame in place, you can pass `inplace=True` as an argument to the `drop()` method.

With the help of this prompt, I managed to drop the last row of the `gaze_df` obtained from student 1, to maintain uniformity, as it had 1 row more than that of `emotion_df`. I used the command `gaze_df.drop(gaze_df.index[-1])`.

**Prompt:** Which plot is good for only plotting 0 and 1 values


**Output:**



For a column with only 0 and 1 values, you can use a **bar plot** or a **count plot**. These plots are effective for visualizing the distribution of binary data.   

#### 1. Bar Plot:

A bar plot (also known as a bar chart or bar graph) displays the frequency of each category in a categorical variable. In this case, it would show the count of 0s and 1s.

```
python  Copy code


import matplotlib.pyplot as plt

# Assuming 'binary_column' is the name of your column
plt.bar(['0', '1'], df['binary_column'].value_counts(), color=['blue', 'green'])
plt.xlabel('Value')
plt.ylabel('Count')
plt.title('Distribution of 0s and 1s')
plt.show()
```

This will give you a visual representation of how many 0s and 1s are present in your column.

#### 2. Count Plot:

A count plot is a specific type of bar plot that shows the count of observations in each category.

```
python  Copy code

import seaborn as sns

# Assuming 'binary_column' is the name of your column
sns.countplot(x='binary_column', data=df)
plt.xlabel('Value')
plt.ylabel('Count')
plt.title('Distribution of 0s and 1s')
plt.show()
```

This type of plot is particularly useful for visualizing the frequency distribution of binary variables.

Both of these plots are suitable for visualizing columns with only 0 and 1 values. Choose the one that you find most informative and visually appealing for your specific context.

I used the count plot to plot the values of gaze when working with the gaze data.

**Prompt:** I want to select all rows but only columns from the 9<sup>th</sup> column till the end in a dataframe.

## Output:



You can select all rows but only specific columns starting from the 9th column until the end of the DataFrame using DataFrame slicing in pandas. You can use the `iloc` indexer for this purpose.

Here's an example:

python

Copy code

```
import pandas as pd

# Create a sample DataFrame
data = {'A': [1, 2, 3, 4],
        'B': [5, 6, 7, 8],
        'C': [9, 10, 11, 12],
        'D': [13, 14, 15, 16],
        'E': [17, 18, 19, 20],
        'F': [21, 22, 23, 24],
        'G': [25, 26, 27, 28],
        'H': [29, 30, 31, 32],
        'I': [33, 34, 35, 36],
        'J': [37, 38, 39, 40]}

df = pd.DataFrame(data)

# Select columns from the 9th column till the end
selected_columns = df.iloc[:, 8:]

print(selected_columns)
```

Output:

css


 Copy code

```
      I    J
0  33  37
1  34  38
2  35  39
3  36  40
```

In this example, we use `df.iloc[:, 8:]` to select all rows (':') and columns starting from the 9th column index (indexing is 0-based).

If you want to select columns by label (column names), you can use `df.loc` instead:

python

 Copy code

```
selected_columns = df.loc[:, 'I':]
```

This will select all rows and columns from 'I' to the last column.

This prompt was required as to remove the undesirable columns from the transcript\_df dataframe which were all before 9<sup>th</sup> column. This method is better as it is more concise than mentioning each column name for using the pd.drop method.



**Prompt:** How to column wise concatenate two dataframes?

**Output:**



If you want to concatenate two DataFrames column-wise, assuming they have the same number of rows, you can use the `pd.concat()` function with `axis=1`.



Here's an example:

python

Copy code

```
import pandas as pd

# Create two sample DataFrames with the same number of rows
df1 = pd.DataFrame({'A': [1, 2, 3],
                    'B': [4, 5, 6]})

df2 = pd.DataFrame({'C': [7, 8, 9],
                    'D': [10, 11, 12]})

# Concatenate the DataFrames column-wise
result_df = pd.concat([df1, df2], axis=1)

print(result_df)
```

Output:

css

Copy code

	A	B	C	D
0	1	4	7	10
1	2	5	8	11
2	3	6	9	12

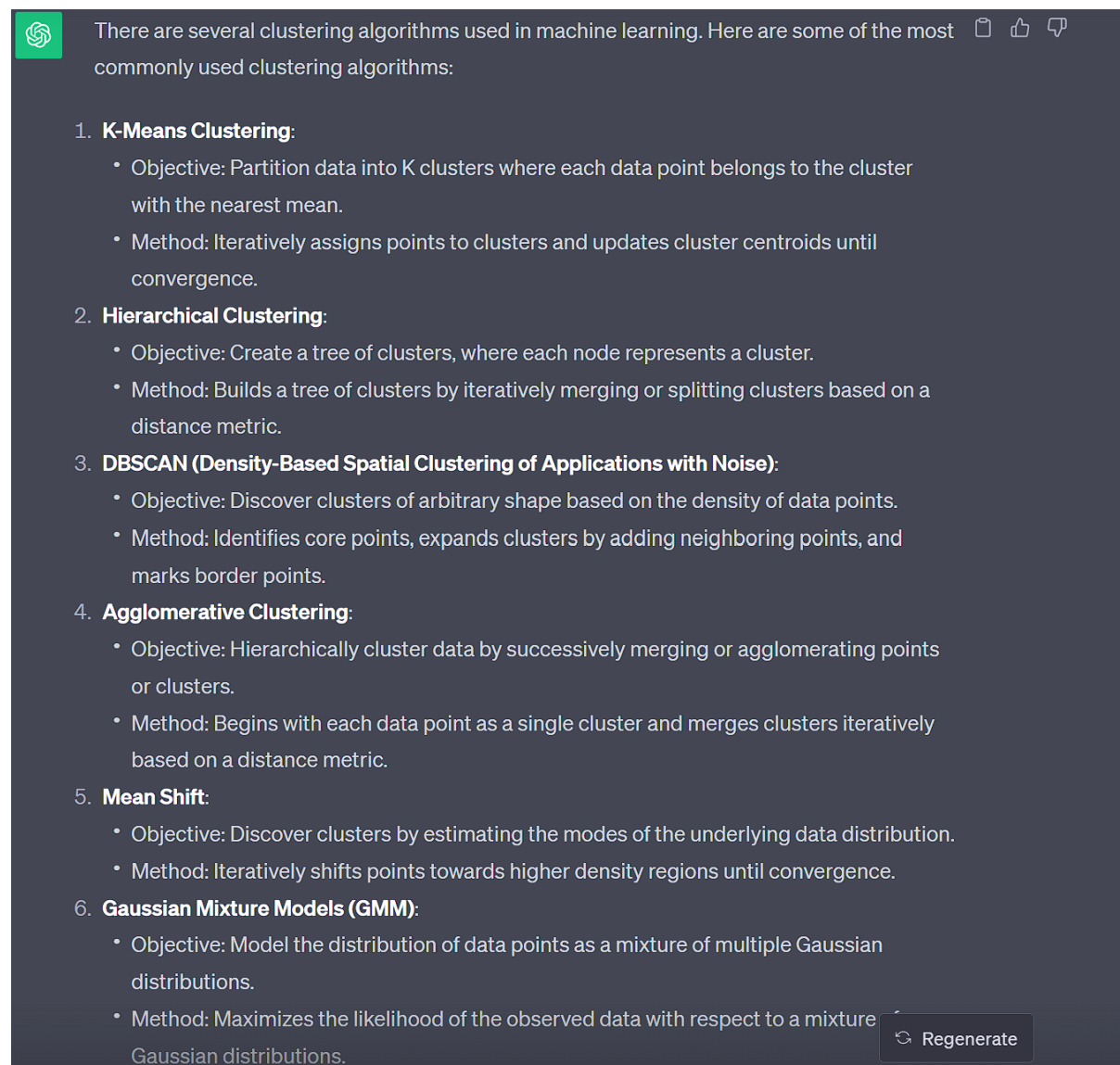
In this example, `pd.concat([df1, df2], axis=1)` concatenates `df1` and `df2` column-wise. The resulting DataFrame `result_df` has columns 'A', 'B' from `df1`, and columns 'C', 'D' from `df2`.

Make sure that the DataFrames have the same number of rows for this operation to work correctly. If the DataFrames have different numbers of rows, you may need to align them or handle the mismatched rows appropriately for your specific use case.

Using this prompt, I was able to concatenate the emotion dataframe(emotion\_df) and gaze dataframe(gaze\_df) into one dataframe(result1\_df), to make my prediction process easier.

**Prompt:** What are the most commonly used clustering algorithms?

**Output:**



The screenshot shows a ChatGPT interface with a dark background. On the left is the OpenAI logo. The main text area contains the following content:

There are several clustering algorithms used in machine learning. Here are some of the most commonly used clustering algorithms:

- 1. K-Means Clustering:**
  - Objective: Partition data into K clusters where each data point belongs to the cluster with the nearest mean.
  - Method: Iteratively assigns points to clusters and updates cluster centroids until convergence.
- 2. Hierarchical Clustering:**
  - Objective: Create a tree of clusters, where each node represents a cluster.
  - Method: Builds a tree of clusters by iteratively merging or splitting clusters based on a distance metric.
- 3. DBSCAN (Density-Based Spatial Clustering of Applications with Noise):**
  - Objective: Discover clusters of arbitrary shape based on the density of data points.
  - Method: Identifies core points, expands clusters by adding neighboring points, and marks border points.
- 4. Agglomerative Clustering:**
  - Objective: Hierarchically cluster data by successively merging or agglomerating points or clusters.
  - Method: Begins with each data point as a single cluster and merges clusters iteratively based on a distance metric.
- 5. Mean Shift:**
  - Objective: Discover clusters by estimating the modes of the underlying data distribution.
  - Method: Iteratively shifts points towards higher density regions until convergence.
- 6. Gaussian Mixture Models (GMM):**
  - Objective: Model the distribution of data points as a mixture of multiple Gaussian distributions.
  - Method: Maximizes the likelihood of the observed data with respect to a mixture of Gaussian distributions.

At the bottom right of the text area is a button labeled "Regenerate".

Out of these, I have tried out the most commonly used algorithms from each Clustering model i.e., Centroid Model (K-Means Clustering), Distribution Model (Gaussian Mixture Model) and Connectivity Model (Agglomerative Clustering) on the given data.

**Prompt:** What is the meaning of `random_state` and `n_init` in K-Means?

**Output:**



In the context of the K-means clustering algorithm, `random_state` and `n_init` are parameters that affect how the algorithm initializes centroids and how many times it runs to find the best clustering. 📄 👍 💬

1. `random_state`:

- The `random_state` parameter is used to set the random seed for centroid initialization. K-means begins by placing centroids randomly in the feature space. However, different random initializations can lead to different results. By setting a specific random seed (an integer), you ensure that the random initialization is reproducible. This can be important for debugging and ensuring consistent results across different runs of the algorithm.
- For example, if you set `random_state=0`, the centroids will be initialized in a specific way, and if you run the K-means algorithm again with the same `random_state`, you'll get the same initial centroids.
- If `random_state` is not set, the centroids will be initialized based on the system's internal random number generator, which might lead to different results on different runs.

2. `n_init`:

- K-means is susceptible to local minima, which means that different initializations can lead to different final cluster assignments. To mitigate this, the algorithm is typically run multiple times with different initializations, and the best result (with the lowest inertia) is chosen.
- The `n_init` parameter determines how many times the algorithm will be run with different centroid seeds. The algorithm will then choose the best result based on the lowest inertia (inertia is a measure of how far the points within a cluster are from the centroid).
- A larger `n_init` value increases the likelihood of finding the global minimum, but it also increases the computational cost. The default value for `n_init` in scikit-learn's K-means implementation is 10.
- It's worth noting that using a higher `n_init` will make the algorithm take more time to complete.

In summary, `random_state` controls the random initialization of centroids, making the initialization process reproducible, while `n_init` controls how many times the algorithm runs with different initializations to find the best clustering.

**Prompt:** Is there a way by which K-Means algorithm can automatically determine the number of times it should run?

## Output:



If `n_init` is set to `auto`, it means that the K-means algorithm will automatically determine the number of times it should run with different initializations based on the size of the dataset and the number of clusters.



In scikit-learn's implementation, when `n_init` is set to `auto`, it will use the following heuristic:

- If the number of samples in the dataset is less than or equal to 2500, `n_init` will be set to 10.
- If the number of samples in the dataset is greater than 2500, `n_init` will be set to 5.

This heuristic is designed to balance the trade-off between computational efficiency and the likelihood of finding a good clustering solution.

Keep in mind that setting `n_init` to `auto` is a convenient way to let the algorithm make an informed decision about how many times to run based on the size of your data. If you have a large dataset, it may reduce the computational burden compared to manually specifying a large number of initializations.

The auto option of `n_init` allows the K-Means algorithm to automatically figure out how many times the algorithm should run to get the best possible clustering.

**Prompt:** How can I use cross-validation to evaluate performance of a clustering algorithm?

## Output:



Using cross-validation for unsupervised learning, such as clustering, involves assessing the stability and robustness of the clustering results across different subsets of the data. While traditional cross-validation methods (like k-fold cross-validation) may not be directly applicable to unsupervised tasks, there are specialized techniques for evaluating clustering performance.

One such method is **Cluster Validation Indices**. These indices provide quantitative measures of the quality of clustering results. Here are a few commonly used cluster validation indices:

1. **Silhouette Score:**
  - Measures how similar an object is to its own cluster compared to other clusters. It ranges from -1 to 1, with higher values indicating better-defined clusters.
2. **Davies-Bouldin Index:**
  - Measures the average similarity between each cluster and its most similar cluster, where lower values indicate better clustering.
3. **Calinski-Harabasz Index:**
  - Evaluates the ratio of between-cluster dispersion to within-cluster dispersion, where higher values indicate better-defined clusters.
4. **Adjusted Rand Index (ARI):**
  - Compares the similarity between the true and predicted clusters, accounting for chance. It ranges from -1 to 1, with higher values indicating better clustering.

Cross-validation cannot be directly used in unsupervised learning to check the accuracy of the clustering results. With the help of this prompt, I was able to find quantitative measures like Cluster Validation Index to evaluate which clustering algorithm works the best.

**Prompt:** Which Cluster Validation Index is the best?

**Output:**



There is no one-size-fits-all answer to which cluster validation index is "best" because the choice of index depends on the specific characteristics of your data and the underlying clustering structure. Different indices may be more appropriate for different types of data and cluster distributions.



Here are some general guidelines for choosing a cluster validation index:

1. **Silhouette Score:**

- Pros: Suitable for a wide range of clustering methods, works well for convex and non-convex clusters.
- Cons: Sensitive to the shape of clusters and assumes clusters are well-defined and separated.
- Use when: You have well-defined, convex clusters and want to evaluate the quality of individual clusters.

2. **Davies-Bouldin Index:**

- Pros: Lower values indicate better-defined clusters, does not assume any particular shape of clusters.
- Cons: Computationally more expensive than some other indices.
- Use when: You want to evaluate the compactness and separation of clusters, and you don't want to assume any specific shape of clusters.

3. **Calinski-Harabasz Index:**

- Pros: Evaluates the ratio of between-cluster dispersion to within-cluster dispersion, suitable for convex and non-convex clusters.
- Cons: Sensitive to the scale of features, may favor compact, well-separated clusters.
- Use when: You want to evaluate the overall quality and separation of clusters.



#### 4. **Adjusted Rand Index (ARI):**

- Pros: Provides a measure of similarity between true and predicted clusters, adjusted for chance.
- Cons: Assumes a ground truth (true labels) for comparison, may not be suitable for exploratory clustering without labeled data.
- Use when: You have labeled data and want to compare the predicted clusters to the true labels.

#### 5. **Homogeneity, Completeness, and V-measure:**

- Pros: Provide detailed information on the quality of clustering with respect to ground truth labels.
- Cons: Require labeled data for comparison.
- Use when: You have labeled data and want to evaluate specific aspects of clustering performance.

#### 6. **Contingency Table-Based Metrics** (e.g., Mutual Information, Normalized Mutual Information):

- Pros: Measure the mutual information between predicted and true clusters.
- Cons: Require labeled data for comparison.
- Use when: You have labeled data and want to measure the amount of information shared between true and predicted clusters.

This prompt shows the pros and cons of every type of Cluster Validation Index. Among all of them, the best one seemed like Davies-Bouldin Index. So, I used Davies Boudlin Index to evaluate the performance of each clustering algorithm on the given datasets.

**Prompt:** How to use Davies Bouldin Index?

**Output:**



To use the Davies-Bouldin Index (DBI) for cluster validation in Python, you can use the `davies_bouldin_score` function from the `sklearn.metrics.cluster` module. Here's an example:



python

Copy code

```
from sklearn.metrics import davies_bouldin_score
from sklearn.cluster import KMeans
import numpy as np

# Assuming X is your data and k is the number of clusters
# Initialize KMeans clustering algorithm
kmeans = KMeans(n_clusters=k, random_state=0)

# Fit KMeans on your data
kmeans.fit(X)

# Get the predicted cluster labels
labels = kmeans.labels_

# Calculate the Davies-Bouldin Index
db_index = davies_bouldin_score(X, labels)

print(f'Davies-Bouldin Index: {db_index}')
```

In this example, `X` represents your data and `k` is the number of clusters you used for the KMeans algorithm.

Here's a breakdown of the steps:



Here's a breakdown of the steps:

1. **Import Libraries:**

- Import the necessary libraries, including `'davies_bouldin_score'` from `'sklearn.metrics.cluster'` and `'KMeans'` from `'sklearn.cluster'`.

2. **Initialize KMeans:**

- Initialize the KMeans clustering algorithm with the desired number of clusters (`'k'`).

3. **Fit KMeans:**

- Fit the KMeans algorithm on your data (`'X'`).

4. **Get Predicted Labels:**

- Get the predicted cluster labels for each data point.

5. **Calculate Davies-Bouldin Index:**

- Use `'davies_bouldin_score'` to calculate the DBI. It takes the data (`'X'`) and the predicted cluster labels as inputs.

6. **Print DBI:**

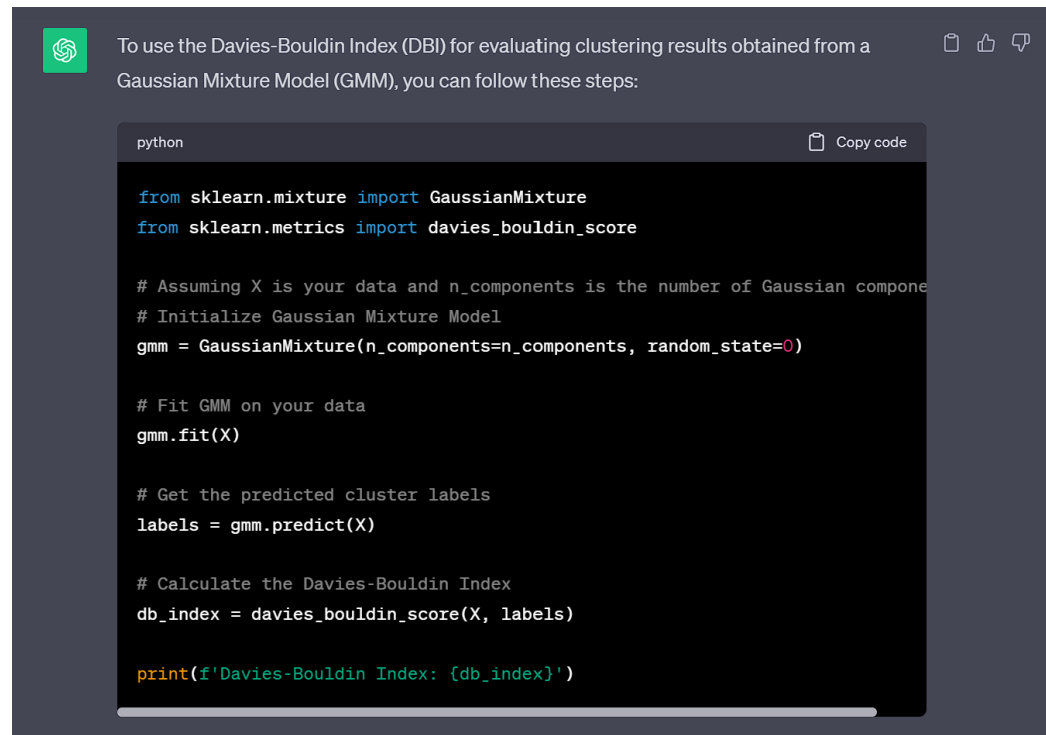
- Print out the calculated Davies-Bouldin Index.

The DBI measures the average similarity between each cluster and its most similar cluster, where lower values indicate better clustering. Keep in mind that this is a relative measure, so it's best used for comparing different clusterings of the same data rather than as an absolute measure of clustering quality.

Using this prompt, I was able to implement DB Index for K-Means algorithm.

**Prompt:** How to use Davies-Bouldin Index for Gaussian Mixture Model?

**Output:** Implementing DB Index for GMM.



```
python
from sklearn.mixture import GaussianMixture
from sklearn.metrics import davies_bouldin_score

# Assuming X is your data and n_components is the number of Gaussian components
# Initialize Gaussian Mixture Model
gmm = GaussianMixture(n_components=n_components, random_state=0)

# Fit GMM on your data
gmm.fit(X)

# Get the predicted cluster labels
labels = gmm.predict(X)

# Calculate the Davies-Bouldin Index
db_index = davies_bouldin_score(X, labels)

print(f'Davies-Bouldin Index: {db_index}')
```

Here's a breakdown of the steps:

1. **Import Libraries:**

- Import the necessary libraries, including `GaussianMixture` from `sklearn.mixture` and `davies_bouldin_score` from `sklearn.metrics.cluster`.

2. **Initialize GMM:**

- Initialize the Gaussian Mixture Model with the desired number of components (`n_components`).

3. **Fit GMM:**

- Fit the GMM on your data (`X`).

4. **Get Predicted Labels:**

- Get the predicted cluster labels for each data point.

5. **Calculate Davies-Bouldin Index:**

- Use `davies_bouldin_score` to calculate the DBI. It takes the data (`X`) and the predicted cluster labels as inputs.


6. **Print DBI:**

- Print out the calculated Davies-Bouldin Index.

The DBI measures the average similarity between each cluster and its most similar cluster, where lower values indicate better clustering. Remember that DBI is a relative measure and is best used for comparing different clusterings of the same data rather than as an absolute measure of clustering quality.

**Prompt:** How to use Davies Bouldin Index for Agglomerative Clustering?

**Output:** Implementing DB Index for Agglomerative Clustering.



Certainly! To use the Davies-Bouldin Index (DBI) for evaluating clustering results obtained from Agglomerative Clustering, you can follow these steps:

```
python
```

```
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import davies_bouldin_score

# Assuming X is your data and n_clusters is the number of clusters
# Initialize Agglomerative Clustering
agg_cluster = AgglomerativeClustering(n_clusters=n_clusters)

# Fit Agglomerative Clustering on your data
agg_cluster.fit(X)

# Get the predicted cluster labels
labels = agg_cluster.labels_

# Calculate the Davies-Bouldin Index
db_index = davies_bouldin_score(X, labels)

print(f'Davies-Bouldin Index: {db_index}')
```

Copy code

Here's a breakdown of the steps:

1. **Import Libraries:**

- Import the necessary libraries, including `AgglomerativeClustering` from `sklearn.cluster` and `davies_bouldin_score` from `sklearn.metrics.cluster`.

2. **Initialize Agglomerative Clustering:**

- Initialize Agglomerative Clustering with the desired number of clusters (`n_clusters`).

3. **Fit Agglomerative Clustering:**

- Fit the Agglomerative Clustering algorithm on your data (`X`).

4. **Get Predicted Labels:**

- Get the predicted cluster labels for each data point.

5. **Calculate Davies-Bouldin Index:**

- Use `davies_bouldin_score` to calculate the DBI. It takes the data (`X`) and the predicted cluster labels as inputs.

6. **Print DBI:**

- Print out the calculated Davies-Bouldin Index.

The DBI measures the average similarity between each cluster and its most similar cluster, where lower values indicate better clustering. As with any clustering evaluation metric, it's important to remember that the DBI provides a relative measure of clustering quality and is best used for comparing different clusterings of the same data.

**Prompt:** Value of Davies-Bouldin Index for better clustering?

## Output:



For the Davies-Bouldin Index (DBI), lower values indicate better clustering. Therefore, a smaller DBI implies better-defined and more separated clusters.



Here's a quick summary:

- **Low DBI:** Indicates that clusters are well-separated and distinct from each other. This is desirable and suggests a better clustering solution.
- **High DBI:** Indicates that clusters are not well-separated and may be overlapping. This suggests that the clustering solution may not be optimal.

Keep in mind that while the DBI is a useful metric for evaluating clustering results, it should be used in conjunction with other cluster validation metrics, and the choice of the "best" clustering solution may also depend on the specific characteristics of your data and the problem you're trying to solve.

With the help of this prompt, we can compare the DB Index of all the clustering algorithms. On comparison it was found that agglomerative clustering had the lowest DB Index, meaning it was the best performing clustering algorithm for our datasets.