MODEL DESIGN and TRAINING:

```
import pandas as pd
import joblib
import time
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.cluster import MiniBatchKMeans
from sklearn.metrics import davies_bouldin_score, calinski_harabasz_score
import seaborn as sns
import matplotlib.pyplot as plt
```

- These lines import the necessary libraries and modules for the code to run, including `pandas` for data manipulation, `joblib` for saving the trained model, `time` for measuring execution time, `numpy` for numerical operations, `TfidfVectorizer` for text feature extraction, `MiniBatchKMeans` for clustering, and `davies_bouldin_score` and `calinski_harabasz_score` for evaluating clustering performance. Additionally, `seaborn` and `matplotlib.pyplot` are imported for data visualization.

```
import warnings
warnings.filterwarnings("ignore")
```

- This line imports the `warnings` module and filters out any warnings to be ignored, preventing them from being displayed in the console.

```
start = time.time()
```

- This line records the starting time of the program execution.

```
df = pd.read_csv("cleaned.csv")
print('\n Dataset read \n')
```

- This line reads the dataset from a CSV file named “cleaned.csv” into a pandas DataFrame `df`. The DataFrame contains the cleaned data used for clustering. The following line prints a message indicating that the dataset has been read.

```
vectorizer = TfidfVectorizer()
```

- This line creates an instance of the `TfidfVectorizer` class, which will be used to convert text data into a numerical representation using the Term Frequency-Inverse Document Frequency (TF-IDF) technique.

```
documents = df['RecipeCategory'].tolist()
```

- This line extracts the 'RecipeCategory' column from the DataFrame `df` and converts it into a list named `documents` . The 'RecipeCategory' column contains the category labels for the recipes.

```
tfidf_matrix = vectorizer.fit_transform(documents)
print('\n Vectorizer fitted \n')
```

- This line applies the `fit_transform()` method of the `TfidfVectorizer` object to the list of documents (`documents`). It converts the text data into a TF-IDF matrix representation. The resulting matrix is stored in the variable `tfidf_matrix` . The following line prints a message indicating that the vectorizer has been fitted.

```
batch_size = 5000
num_batches = len(df) // batch_size
```

- These lines define the batch size for training the clustering model (`batch_size`) and calculate the number of batches needed to cover the entire dataset (`num_batches`). The dataset is divided into smaller batches to train the model in smaller increments for efficiency.

```
x = df['RecipeCategory'].unique()
num_clusters = len(x)
model = MiniBatchKMeans(n_clusters=num_clusters, batch_size=100)
print('\n Model created \n Beginning model training....... \n')
```

- These lines determine the number of unique categories in the 'RecipeCategory' column (`num_clusters`). It is used as the number of clusters for the MiniBatchKMeans model. The `MiniBatchKMeans` object is created with the specified number of clusters and a batch size of 100. The following line prints a message indicating that the model has been created and training is starting.

```
for i in range(num_batches):
    start_index = i * batch_size
    end_index = start_index + batch_size

    batch_data = tfidf_matrix[start_index:end_index]




    if batch_data.shape[0] > 0:
        model.partial_fit(batch_data)

    print(f"Iteration: {i + 1}/{num_batches}")
print('\n Model trained \n')
```

- This loop iterates over each batch in the range of `num_batches`. For each iteration, it determines the start and end indices of the current batch based on the batch size. It extracts the corresponding batch data from the `tfidf_matrix`. The model is then trained on the batch data using the `partial_fit()` method, which allows incremental training. If the batch data is not empty (has more than 0 samples), the model is updated. The current iteration number is printed for tracking progress. After the loop finishes, a message is printed indicating that the model has been trained.

```
joblib.dump(model, 'clustering_model.pkl')
print('\n Model saved \n')
```

- These lines save the trained model to a file named 'clustering_model.pkl' using the `joblib.dump()` function. The model can be loaded from this file later for predictions. The following line prints a message indicating that the model has been saved.

```
print("Predicting labels...")
labels = []
for i, sample in enumerate(tfidf_matrix):
    print(f"Processing sample {i + 1}/{tfidf_matrix.shape[0]}")
    label = model.predict(sample)
    labels.append(label)
print("Label prediction completed.\n\n\n")
```

- This block of code predicts labels for each sample in the `tfidf_matrix` using the trained model. It iterates over each sample in the matrix, retrieves the predicted label using the `predict()` method of the model, and appends the label to the `labels` list. Messages are printed to track the progress of sample processing and label prediction.

```
num_batches_eval = 30
batch_size_eval = len(df) // num_batches_eval
```

- These lines define the number of batches (`num_batches_eval`) and the batch size (`batch_size_eval`) for evaluating the clustering performance metrics. The evaluation is performed on a subset of the data.

```
davies_bouldin_scores = []
calinski_harabasz_scores = []
silhouette_scores = []
```

- These lines create empty lists to store the evaluation scores for the Davies-Bouldin Index, Calinski-Harabasz Index, and Silhouette Score.

```
print("******** Calculating Davies-Bouldin Index ********")
davies_bouldin = 0.0

for i in range(num_batches_eval):
    start_index = i * batch_size_eval
    end_index = start_index + batch_size_eval

    batch_data = tfidf_matrix[start_index:end_index]
    batch_labels = labels[start_index:end_index]

    davies_bouldin += davies_bouldin_score(batch_data.toarray().astype(np.float16), batch_label
s)
    davies_bouldin_scores.append(davies_bouldin_score(batch_data.toarray().astype(np.float16), b
atch_labels))

    print(f"Davies-Bouldin Iteration: {i + 1}/{num_batches_eval}")

davies_bouldin /= num_batches_eval
print(f"Davies-Bouldin Index: {davies_bouldin}")
print("Davies-Bouldin calculation completed.")
```

- This block of code calculates the Davies-Bouldin Index for evaluating the clustering performance. It iterates over each batch in the evaluation dataset and calculates the Davies-Bouldin Index for each batch using the `davies_bouldin_score()` function. The calculated score for each batch is added to the `davies_bouldin_scores

` list. The average Davies-Bouldin Index is then computed by dividing the sum of scores by the number of batches. The index value is printed, and a message is displayed to indicate the completion of the calculation.

```
print("\n\n******** Calculating Calinski-Harabasz Index *********")
calinski_harabasz = 0.0

for i in range(num_batches_eval):
    start_index = i * batch_size_eval
    end_index = start_index + batch_size_eval

    batch_data = tfidf_matrix[start_index:end_index]
    batch_labels = labels[start_index:end_index]

    calinski_harabasz += calinski_harabasz_score(batch_data.toarray().astype(np.float16), batch_
labels)
    calinski_harabasz_scores.append(calinski_harabasz_score(batch_data.toarray().astype(np.float
16), batch_labels))

    print(f"Calinski-Harabasz Iteration: {i + 1}/{num_batches_eval}")

calinski_harabasz /= num_batches_eval
print(f"Calinski-Harabasz Index: {calinski_harabasz}")
print("Calinski-Harabasz calculation completed.\n\n")
```

- This block of code calculates the Calinski-Harabasz Index for evaluating the clustering performance. It iterates over each batch in the evaluation dataset and calculates the Calinski-Harabasz Index for each batch using the `calinski_harabasz_score()` function. The calculated score for each batch is added to the `calinski_harabasz_scores` list. The average Calinski-Harabasz Index is then computed by dividing the sum of scores by the number of batches. The index value is printed, and a message is displayed to indicate the completion of the calculation.

```
plt.figure(figsize=(12, 6))
sns.lineplot(x=range(num_batches_eval), y=davies_bouldin_scores)
plt.xlabel('Number of Clusters')
plt.ylabel('Davies-Bouldin Score')
plt.title('Davies-Bouldin Index Over Iterations')

for i, score in enumerate(davies_bouldin_scores):
    plt.annotate(f'{score:.2f}', (i, score), textcoords="offset points", xytext=(0,10), ha='cent
er')

plt.show()
```

- This block of code visualizes the Davies-Bouldin scores over iterations. It creates a line plot using `sns.lineplot()` with the number of iterations on the x-axis and the Davies-Bouldin scores on the y-axis. Annotations are added to each score point on the plot. The plot is displayed using `plt.show()`.

```
plt.figure(figsize=(12, 6))
sns.lineplot(x=range(num_batches_eval), y=calinski_harabasz_scores)
plt.xlabel('Number of Clusters')
plt.ylabel('Calinski-Harabasz Score')
plt.title('Calinski-Harabasz Index Over Iterations')

for i, score in enumerate(calinski_harabasz_scores):
    plt.annotate(f'{score:.2f}', (i, score), textcoords="offset points", xytext=(0,10), ha='cent
er')
```

```
plt.show()
```

- This block of code visualizes the Calinski-Harabasz scores over iterations. It creates a line plot using `sns.lineplot()` with the number of iterations on the x-axis and the Calinski-Harabasz scores on the y-axis. Annotations are added to each score point on the plot. The plot is displayed using `plt.show()` .

```
print(f"Number of clusters: {num_clusters}")
print(f"Davies-Bouldin


 Index: {davies_bouldin}")
print(f"Calinski-Harabasz Index: {calinski_harabasz}")
```

- These lines print the number of clusters, Davies-Bouldin Index, and Calinski-Harabasz Index to the console, providing a summary of the clustering results.

```
end = time.time()
execution_time = end - start
print(f"\n Execution time: {execution_time:.2f} seconds")
```

- This block of code calculates the execution time of the program by subtracting the starting time (`start`) from the current time (`end`). The execution time is then printed to the console.

Here's an explanation of the significance of each code block and why it is used:

1. **Importing Libraries**:
    - The first block of code imports the necessary libraries and modules required for the subsequent operations. These libraries provide functionalities for data manipulation, machine learning, evaluation metrics, visualization, and time tracking.
2. **Installation Instructions**:
    - The commented lines provide installation instructions for specific packages (`threadpoolctl` and `fsspec`) that might be required to run the code in specific environments like Anaconda, Jupyter, or Google Colab. The user is instructed to install the packages and restart the kernel if necessary.
3. **Removing Warnings**:
    - This code block suppresses warnings from being displayed in the console. It uses the `warnings` module to filter and ignore warning messages, ensuring a cleaner console output.

4. **Starting Time**:

   - The `start` variable captures the current time when this code block is executed. It marks the starting point for measuring the execution time of the code.

5. **Loading the Dataset**:

   - This code block reads the dataset from a CSV file using the `pd.read_csv()` function from the pandas library. The dataset is loaded into a DataFrame object named `df` . The path to the CSV file needs to be specified.
   - This step is significant as it provides the data on which the clustering model will be trained and evaluated.

6. **Creating a TF-IDF Vectorizer**:

   - The code block initializes a `TfidfVectorizer` object from the `sklearn.feature_extraction.text` module. This vectorizer will be used to convert the text data into numerical features based on the Term Frequency-Inverse Document Frequency (TF-IDF) algorithm.
   - TF-IDF is commonly used in text mining and natural language processing tasks to represent text documents numerically, giving importance to rare words.
   - This step is essential for transforming the text data into a format that can be used by machine learning algorithms.

7. **Initializing Empty Document List**:

   - The `documents` list is initialized as an empty list. It will be used to store the text documents from the dataset.
   - This step is necessary to prepare the data for vectorization and clustering.

8. **Creating the TF-IDF Matrix**:

   - The code block uses the previously created TF-IDF vectorizer (`vectorizer`) to transform the text documents into a TF-IDF matrix representation.
   - The `fit_transform()` method of the vectorizer is applied to the `documents` list, which converts the text data into a sparse matrix of TF-IDF features.
   - This step is crucial for generating the input data for training the clustering model.

9. **Training the Clustering Model in Batches**:

   - The code block sets the parameters for training the MiniBatchKMeans clustering model.
   - The number of clusters (`num_clusters`) is set to the number of unique values in the 'RecipeCategory' column of the dataset.
   - The code then initializes a `MiniBatchKMeans` model object from the `sklearn.cluster` module, specifying the number of clusters and the batch size.
   - Training the model in batches is done to handle large datasets that may not fit into memory all at once. It allows for more efficient memory usage and processing.

- The code block uses a loop to iterate over the batches of data and performs partial fitting of the model on each batch using the `partial_fit()` method.
- This step is significant for training the clustering model and adapting it to large datasets in a memory-efficient manner.

10. **Saving the Trained Model**:

    - After the model has been trained, it is saved to a file using the `joblib.dump()` function from the `joblib` module. The saved model can be loaded later for inference or further analysis.
    - This step is crucial for persisting the trained model for future use without retraining.

11. **Predicting Labels and Evaluating Performance**:

    - This code block predicts labels for each sample in the TF-IDF matrix and evaluates the clustering performance using the Davies-Bouldin Index and Calinski-Harabasz Index.
    - It uses a loop to iterate over each sample in the TF-IDF matrix, predicts the cluster label for each sample using the `predict()` method of the clustering model, and appends the labels to the `labels` list.
    - The code then calculates the average Davies-Bouldin Index and Calinski-Harabasz Index over the batches of data using the respective scoring functions from the `sklearn.metrics` module.
    - This step is significant for evaluating the quality of the clustering model and its ability to group similar recipes together.

12. **Visualizing the Evaluation Scores**:

    - The code block generates visualizations of the Davies-Bouldin scores and Calinski-Harabasz scores over the iterations.
    - It uses the `matplotlib.pyplot` module to create line plots showing the scores as a function of the number of clusters.
    - Annotations are added to the plots to display the exact score value for each iteration.
    - This step helps in visually understanding the behavior of the clustering model and choosing an appropriate number of clusters based on the evaluation scores.

13. **Printing the Clustering Results**:

    - This code block prints the number of clusters, Davies-Bouldin Index, and Calinski-Harabasz Index to the console.
    - It provides a summary of the clustering results and the performance metrics obtained.

14. **Calculating and Printing Execution Time**:

    - The code block calculates the execution time of the entire program by subtracting the starting time (`start`) from the current time (`end`).
    - The execution time is then printed to the console, giving an indication of how long the program took to run.

    - This step is useful for benchmarking and assessing the efficiency of the code.

Each code block plays a crucial role in loading the data, preparing it for clustering, training the model, evaluating its performance, visualizing the results, and reporting the outcomes.