## Code:-

```
import pandas as pd
from fuzzywuzzy import fuzz
import numpy as np
import time
import tensorflow_hub as hub

def load_data(file_path):
    return pd.read_excel(file_path)

def lowercase_all(df):
    return df.applymap(lambda x: x.lower() if isinstance(x, str) else x)

def generate_embeddings(model, df):
    embeddings = {}
    for column in df.columns:
        column_embeddings = model(df[column].astype(str).tolist()).numpy()
        embeddings[column] = column_embeddings
    return embeddings

def compute_cosine_similarity(query_embedding, entry_embeddings):
    similarities = np.dot(entry_embeddings, query_embedding.T)
    return similarities.max()

def calculate_similarities(df, query, model):
    query_embedding = model([query]).numpy()
    results = []
    embeddings = generate_embeddings(model, df)

    for index, row in df.iterrows():
        row_similarities = []
        for column in df.columns:
            entry_embeddings = embeddings[column][index].reshape(1, -1)
            semantic_similarity = compute_cosine_similarity(query_embedding,
entry_embeddings)

            # Fuzzy logic match
            fuzzy_similarity = fuzz.ratio(query, str(row[column]))
            fuzzy_similarity /= 100  # Normalize fuzzy score to [0, 1] range

            # Combine both similarities
            combined_similarity = (semantic_similarity + fuzzy_similarity) / 2
            row_similarities.append((column, combined_similarity))
```

```
        max_similarity = max(row_similarities, key=lambda x: x[1])[1]
        results.append((index, max_similarity))

    return results

def find_best_match(similarities):
    best_match = max(similarities, key=lambda x: x[1], default=None)
    return best_match

def process_data(file_path, query):
    df = load_data(file_path)
    df = lowercase_all(df)
    model = hub.load("https://tfhub.dev/google/universal-sentence-encoder/4")
    similarities = calculate_similarities(df, query, model)
    best_match = find_best_match(similarities)
    return best_match, df

def main():
    file_path = input("Enter the path to the Excel file: ")
    user_query = input("Enter the query to match: ")
    start_time = time.time()
    best_match, df = process_data(file_path, user_query)

    if best_match is not None:
        index, similarity = best_match
        print("File path:", file_path)
        print("Query given:", user_query)
        print("Best Match Found at index:", index)
        print("Similarity Score:", similarity)
        print(df.iloc[index])
    else:
        print("No match found")

    end_time = time.time()
    runtime = end_time - start_time
    print(f"Runtime: {runtime} seconds")

if __name__ == "__main__":
    main()
```

Documentation: Hybrid Model Using Universal Sentence Encoder (USE) and Damerau-Levenshtein for Query Matching

1. **Objective and Overview**

- This hybrid model is designed to enhance query-to-entry matching by leveraging semantic embedding capabilities of the Universal Sentence Encoder (USE) alongside fuzzy matching. USE provides deep contextual embeddings, making it effective in identifying semantic similarities, while fuzzy logic matches the string similarity, capturing approximate text matches. This combined approach balances semantic understanding and tolerance for slight textual discrepancies.

2. **Core Components and Functionalities**

- Data Loading and Preprocessing
  - Data Loading: The `load_data` function loads data from an Excel file, allowing for flexible data importation.
  - Preprocessing: The `lowercase_all` function standardizes text entries to lowercase, ensuring uniformity and preventing case-sensitivity issues during matching.

- Universal Sentence Encoder (USE) for Semantic Similarity
  - Model Selection: The Universal Sentence Encoder (USE) is loaded from TensorFlow Hub, which provides pre-trained embeddings optimized for semantic similarity tasks.
  - Embedding Generation: The `generate_embeddings` function generates embeddings for each column in the dataset, which are later used to calculate semantic similarity with the user query.
  - Query Encoding: The user query is encoded as a USE embedding, which is compared with dataset embeddings to calculate semantic similarity scores.

- Fuzzy Matching for Approximate String Similarity
  - Fuzzy Score Calculation: Using the FuzzyWuzzy library, the `calculate_similarities` function calculates a fuzzy similarity score between the user query and each entry. The score is normalized to fall within the [0,1] range for compatibility with cosine similarity scores.

- Hybrid Similarity Calculation
  - Combined Similarity Score: The model calculates both semantic and fuzzy similarity scores and combines them by averaging. This combined score provides a balanced view of semantic closeness and textual similarity.
  - Result Filtering: Rows with a high combined similarity score are considered as potential matches, with the highest similarity row being selected as the best match.

3. **Workflow and Logic**

- Step 1: File Upload and Data Preparation
  - The model loads the dataset from an Excel file, applying necessary lowercase transformations. This standardization ensures a consistent baseline for matching.

- Step 2: Matching Process with USE Embeddings and Fuzzy Matching

- The user inputs a query, which is transformed into a USE embedding. The hybrid model then calculates both semantic and fuzzy similarities for each row in the dataset and combines these scores to produce a final similarity measure.

- Step 3: Best Match Selection
  - The model identifies the best match for the query by finding the row with the highest combined similarity score. The selected row index, similarity score, and row details are then displayed to the user.

4. **Performance and Runtime Considerations**
   - Efficiency of USE: The Universal Sentence Encoder offers a strong trade-off between accuracy and performance, as it quickly computes contextual embeddings. Fuzzy matching with FuzzyWuzzy adds minimal overhead, and the combination approach ensures balanced and efficient processing.
   - Runtime: The model's runtime is affected by the dataset size and query complexity but remains efficient for most typical datasets.

5. **Limitations and Future Enhancements**
   - Threshold Adjustment: A static threshold may limit accuracy, so implementing dynamic thresholds or optimized thresholds based on context could enhance accuracy.
   - USE Fine-tuning: Fine-tuning the USE model on specific domain-related data could improve matching performance, particularly for specialized datasets.