Code:-

```
import pandas as pd
import numpy as np
import os
import time
from sentence_transformers import SentenceTransformer
from sklearn.metrics import classification_report
from pyxdameraulevenshtein import damerau_levenshtein_distance
def load_data(file_path):
    file_extension = os.path.splitext(file_path)[1]
    if file_extension == '.csv':
        return pd.read_csv(file_path)
    elif file_extension in ['.xls', '.xlsx']:
        return pd.read_excel(file_path)
    else:
        raise ValueError("Unsupported file format.")
def lowercase_all(df):
    df = df.map(lambda x: x.lower() if isinstance(x, str) else x)
    return df
def generate_embeddings(model, df):
    embeddings = {}
    for column in df.columns:
        column_embeddings = model.encode(df[column].astype(str).tolist(),
convert_to_tensor=True)
        embeddings[column] = column_embeddings.cpu().numpy()
    return embeddings
def compute_damerau_levenshtein_similarity(query, entry):
    max_len = max(len(query), len(entry))
    if max_len == 0:
        return 1.0
    distance = damerau_levenshtein_distance(query, entry)
    return 1 - (distance / max_len)
def calculate_semantic_similarities(df, query, model):
    query_embedding = model.encode([query], convert_to_tensor=True).cpu().numpy()
    embeddings = generate_embeddings(model, df)
    results = []
    for index, row in df.iterrows():
        row_similarities = []
        for column in df.columns:
            entry = str(row[column])
            entry_embedding = embeddings[column][index].reshape(1, -1)
            semantic_similarity = np.dot(entry_embedding, query_embedding.T).max()
            damerau_levenshtein_similarity =
compute_damerau_levenshtein_similarity(query, entry)
```

```
            combined_similarity = (semantic_similarity +
damerau_levenshtein_similarity) / 2
            row_similarities.append((column, combined_similarity))
        max_similarity = max(row_similarities, key=lambda x: x[1])[1]
        if max_similarity >= 0.5:
            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 compute_metrics(df, best_match):
    y_true = df['ground_truth'].tolist()
    y_pred = [1 if i == best_match[0] else 0 for i in range(len(df))]
    report = classification_report(y_true, y_pred, output_dict=True)
    return pd.DataFrame(report).transpose()
def process_data(file_path, query):
    df = load_data(file_path)
    df = lowercase_all(df)
    df['ground_truth'] = [1 if i == 0 else 0 for i in range(len(df))]
    model = SentenceTransformer('all-MiniLM-L6-v2',
tokenizer_kwargs={'clean_up_tokenization_spaces': True})
    similarities = calculate_semantic_similarities(df, query, model)
    best_match = find_best_match(similarities)

    return best_match, df
def main():
    file_path = input("Enter the path to the file: ")
    user_query = input("Enter the query to match: ")
    start_time = time.time()
    best_match, df = process_data(file_path, user_query)
    print("File path:", file_path)
    print("Query given:", user_query)
    if best_match is not None:
        index, similarity = best_match
        print("Best Match Found at index:", index)
        print("Similarity Score:", similarity)
        print(df.drop(columns=['ground_truth']).iloc[index])
    else:
        print("Sorry, No match found with the input query, Please try again!")
    end_time = time.time()
    runtime = end_time - start_time
    print(f"Runtime: {runtime} seconds")
if __name__ == "__main__":
    main()
```

Documentation: Hybrid Model Using MiniLM and Damerau-Levenshtein for Query Matching

1. **Objective and Overview**
   - This initial hybrid model approach is designed to accurately match user queries to entries in a dataset. By combining the semantic strength of MiniLM embeddings with Damerau-Levenshtein distance for string matching, the model achieves a balance between semantic meaning and string similarity. This approach is particularly useful for tasks that require both a high level of understanding of query intent and tolerance for small textual variations.

2. **Core Components and Functionalities**

   - Data Loading and Preprocessing
     - Data Loading: The `load_data` function supports multiple file formats, including CSV and Excel, to allow for flexibility in the dataset's source.
     - Preprocessing: The `lowercase_all` function ensures that all text entries are converted to lowercase, standardizing the data and preventing case-sensitivity issues during similarity calculations.
     - Ground Truth Setup: A basic ground truth column (`ground_truth`) is created to simulate a binary classification scenario for demonstration purposes.

   - MiniLM Model for Semantic Similarity
     - Model Selection: The model leverages the `all-MiniLM-L6-v2` model from the Sentence Transformers library, which is optimized for embedding-based similarity tasks.
     - Embedding Generation: The `generate_embeddings` function creates embeddings for each column in the dataset. These embeddings are later used to compute semantic similarity with the user's query.
     - Query Encoding: The user's query is encoded as an embedding, which is then compared against the dataset's entries to assess the level of semantic similarity.

   - Damerau-Levenshtein Distance for String Similarity
     - String Comparison: The `compute_damerau_levenshtein_similarity` function calculates the Damerau-Levenshtein similarity between the user query and each dataset entry, taking into account potential typos or transpositions.

   - Hybrid Similarity Calculation
     - Combined Similarity: The `calculate_semantic_similarities` function computes both semantic and Damerau-Levenshtein similarities, then averages them to achieve a final similarity score for each entry.
     - Result Filtering: Entries with a similarity score above a threshold (set at 0.5 by default) are considered as potential matches, and the highest similarity match is selected as the best match.

3. **Workflow and Logic**

   - Step 1: File Upload and Dataset Preparation

- The program prompts the user to upload a dataset (CSV or Excel) using the `load_data` function. The dataset is then preprocessed to standardize text format and set up ground truth labels for evaluation.

- Step 2: Semantic and String Similarity Matching
  - The user inputs a query, which is processed to generate both semantic and Damerau-Levenshtein similarity scores for each row in the dataset. This hybrid similarity approach balances both meaning and textual proximity, leading to an accurate match.

- Step 3: Selecting the Best Match
  - The highest-scoring match is retrieved through the `find_best_match` function, and its index and similarity score are displayed to the user.

- Step 4: Model Evaluation (Optional)
  - If the dataset includes a ground truth, the program computes classification metrics (e.g., accuracy, precision, recall) through `compute_metrics`, allowing for an assessment of the model's performance on the matching task.

4. **Performance and Runtime Considerations**
   - The hybrid approach is computationally efficient, leveraging MiniLM embeddings for rapid semantic similarity scoring and Damerau-Levenshtein for lightweight string comparisons. Runtime for processing varies based on dataset size and query complexity but generally remains within an acceptable range for most use cases.

5. **Limitations and Future Enhancements**
   - Threshold Sensitivity: The similarity threshold can impact the model's performance. A dynamic threshold or context-specific threshold optimization could enhance accuracy in future iterations.
   - Fine-tuning MiniLM: Fine-tuning MiniLM on specific domain-related data could improve semantic similarity matching for specialized datasets.