Code:-

```
import pandas as pd
from tkinter import Tk
from tkinter.filedialog import askopenfilename
from pyxdameraulevenshtein import damerau_levenshtein_distance
import spacy
from sklearn.model_selection import train_test_split
import torch
from transformers import RobertaTokenizer, RobertaForSequenceClassification, AdamW
from sklearn.metrics import accuracy_score

# Load pre-trained RoBERTa model and tokenizer for intent classification
model_name = 'roberta-base'
tokenizer = RobertaTokenizer.from_pretrained(model_name)
model = RobertaForSequenceClassification.from_pretrained(model_name, num_labels=3)
model.eval()

# Load SpaCy for entity extraction and matching
nlp = spacy.load("en_core_web_sm")

# Global variables to store the filtered and split datasets
filtered_data = None
train_data = None
test_data = None

# Function to upload the dataset
def upload_file():
   Tk().withdraw()
   file_path = askopenfilename(title="Select CSV/Excel File",
                               filetypes=[("CSV Files", "*.csv"), ("Excel Files",
"*.xlsx")])
   if file_path:
       if file_path.endswith('.csv'):
           data = pd.read_csv(file_path)
       elif file_path.endswith('.xlsx'):
           data = pd.read_excel(file_path)
       return data
   else:
       return None

# Function to split the dataset into training and testing (80/20 split)
def split_data(data):
   return train_test_split(data, test_size=0.2, random_state=42)

# Function to find the best match using Damerau-Levenshtein distance
```

```
def damerau_levenshtein_match(query, choices):
    best_match = None
    best_distance = float('inf')

    for choice in choices:
        distance = damerau_levenshtein_distance(query, choice)
        if distance < best_distance:
            best_distance = distance
            best_match = choice

    return best_match, best_distance

# Function to filter the dataset based on 'Project', 'DISCIPLINE', and 'DASHBOARD
NAME'
def filter_data(data):
    project_input = input("Enter Project Name: ").strip()
    discipline_input = input("Enter Discipline: ").strip()
    dashboard_input = input("Enter Dashboard Name: ").strip()

    project_choices = data['Project'].dropna().unique().tolist()
    discipline_choices = data['DISCIPLINE'].dropna().unique().tolist()
    dashboard_choices = data['DASHBOARD NAME'].dropna().unique().tolist()

    best_project, project_distance = damerau_levenshtein_match(project_input,
project_choices)
    best_discipline, discipline_distance =
damerau_levenshtein_match(discipline_input, discipline_choices)
    best_dashboard, dashboard_distance = damerau_levenshtein_match(dashboard_input,
dashboard_choices)

    print(f"Best match for 'Project': {best_project} (Distance:
{project_distance})")
    print(f"Best match for 'DISCIPLINE': {best_discipline} (Distance:
{discipline_distance})")
    print(f"Best match for 'DASHBOARD NAME': {best_dashboard} (Distance:
{dashboard_distance})")

    filtered = data[
        (data['Project'] == best_project) &
        (data['DISCIPLINE'] == best_discipline) &
        (data['DASHBOARD NAME'] == best_dashboard)
    ]

    if filtered.empty:
        print("No records found with the specified filters.")
```

```
        return None
    return filtered

# Function to classify user intent using RoBERTa
def classify_intent(query):
    inputs = tokenizer(query, return_tensors='pt', padding=True, truncation=True,
max_length=128)
    outputs = model(**inputs)
    logits = outputs.logits
    predicted_label = torch.argmax(logits, dim=1).item()

    intent_labels = {0: "information_request", 1: "comparison", 2: "other"}

    return intent_labels[predicted_label]

# Function to extract and match entities using spaCy
def extract_entities(text):
    doc = nlp(text)
    entities = [(ent.text, ent.label_) for ent in doc.ents]
    keywords = [token.lemma_ for token in doc if not token.is_stop and not
token.is_punct]

    return entities, keywords

# Function to train RoBERTa on custom intent classification data (optional)
def train_intent_classifier(train_data):
    # Preprocess the data for training
    texts = train_data['text'].tolist()  # Replace with your text column
    labels = train_data['label'].tolist()  # Replace with your label column

    inputs = tokenizer(texts, return_tensors='pt', padding=True, truncation=True,
max_length=128)
    labels = torch.tensor(labels)

    # Define an optimizer and loss function
    optimizer = AdamW(model.parameters(), lr=1e-5)

    model.train()
    for epoch in range(3):  # Train for 3 epochs
        optimizer.zero_grad()
        outputs = model(**inputs, labels=labels)
        loss = outputs.loss
        loss.backward()
        optimizer.step()
```

```
    print(f"Training complete. Final loss: {loss.item()}")

# Function to match the user's query with 'daxformula_Processed' and return
'daxformula'
def match_query(filtered_data):
    query = input("Enter your query to match with daxformula_Processed: ").strip()

    # Classify the user's query intent using RoBERTa
    intent = classify_intent(query)
    print(f"Classified user query intent: {intent}")

    # Extract entities and keywords from the user query
    user_entities, user_keywords = extract_entities(query)
    print(f"Extracted Entities from user query: {user_entities}")
    print(f"Extracted Keywords from user query: {user_keywords}")

    # Get all the values from 'daxformula_Processed' column
    processed_formulas = filtered_data['daxformula_Processed'].dropna().tolist()

    # Perform Damerau-Levenshtein distance matching
    best_match, distance = damerau_levenshtein_match(query, processed_formulas)

    if best_match is not None:
        # Get the corresponding row and 'daxformula' for the best match
        matched_row = filtered_data[filtered_data['daxformula_Processed'] ==
best_match]

        # Extract entities and keywords from the matched row
        dataset_entities, dataset_keywords =
extract_entities(matched_row.iloc[0]['daxformula_Processed'])
        print(f"\nBest Match: {best_match} (Distance: {distance})")
        print(f"Corresponding daxformula: {matched_row.iloc[0]['daxformula']}")
        print(f"Extracted Entities from dataset: {dataset_entities}")
        print(f"Extracted Keywords from dataset: {dataset_keywords}")

        # Perform entity matching
        user_entity_texts = set(ent[0] for ent in user_entities)
        dataset_entity_texts = set(ent[0] for ent in dataset_entities)

        # Match based on common entities
        common_entities = user_entity_texts.intersection(dataset_entity_texts)
        if common_entities:
            print(f"Matched entities between query and dataset: {common_entities}")
        else:
            print("No common entities found between query and dataset.")
```

```python
    else:
        print("\nNo match found for the query.")

# Chatbot interaction flow
def chatbot_session():
    print("Welcome to the Machine Learning Chatbot!")
    file = upload_file()

    if file is not None:
        print("File uploaded successfully.")

        # Step 1: Split the dataset into training and testing sets
        global train_data, test_data
        train_data, test_data = split_data(file)
        print(f"Data split into Training (80%) and Testing (20%)", flush=True)

        # Step 2: Filter the dataset for interaction
        global filtered_data
        filtered_data = filter_data(train_data)

        if filtered_data is not None:
            # Step 3: Query matching and interaction
            while True:
                user_input = input("\nYou can ask 'Make a query', 'Change filters', or 'Quit': ").lower()

                if user_input == 'make a query':
                    match_query(filtered_data)
                    print("\nQuery processed. You can ask again or change filters.", flush=True)


                elif user_input == 'change filters':
                    filtered_data = filter_data(train_data)
                    print("\nFilters updated. You can ask a query now.", flush=True)

                elif user_input == 'quit':
                    print("Exiting the chatbot. Goodbye!", flush=True)
                    break

                else:
                    print("I didn't understand that. Please try again.", flush=True)
    else:
        print("No file uploaded. Please try again.", flush=True)

chatbot_session()
```

Documentation: Initial Approach with RoBERTa and Tkinter for Chatbot UI

1. **Objective and Overview**
   - This initial approach focuses on developing a chatbot for intent classification and entity extraction, aiming to enhance user interaction in accessing dataset-specific insights. Using a combination of machine learning techniques, the chatbot can classify intents and extract entities, allowing users to query project-related data effectively.

2. **Core Components and Functionalities**

   - Intent Classification with RoBERTa
     - Model Selection: Utilized the `RoBERTa-base` model from Hugging Face for its powerful language understanding capabilities.
     - Implementation:
       - The model is fine-tuned with a 3-label classification (e.g., `information_request`, `comparison`, and `other`) for user intent classification.
       - The intent classification function processes the input using `RobertaTokenizer` and infers labels through the pre-trained model, providing high accuracy in intent detection.
     - Results: The RoBERTa-based classifier quickly identifies the user's intent, setting a foundation for relevant responses or redirection within the chatbot.

   - Entity Extraction and Matching with SpaCy and Damerau-Levenshtein Distance
     - SpaCy Integration:
       - The chatbot uses SpaCy’s `en_core_web_sm` for NER (Named Entity Recognition) to identify and extract entities from user queries.
       - Entity matching facilitates identifying the best available project data that aligns with the user’s inquiry.
     - Damerau-Levenshtein Distance for Best Match:
       - To improve query matching accuracy, the Damerau-Levenshtein algorithm calculates and identifies the closest matches for user-provided project names, disciplines, or dashboard identifiers, helping prevent minor spelling errors from disrupting matches.

   - Enhanced UI with Tkinter for File Uploading and Filtering
     - UI for User Convenience: Tkinter provides a straightforward file upload interface for users to select project datasets in CSV or Excel formats.
     - Interactive Filters: Through user inputs, the chatbot filters data by project, discipline, and dashboard names, allowing users to select precise data for querying.
     - Guided Prompts and Responses: Tkinter enables clear user feedback, keeping users informed of their selected filters and available options.

3. **Workflow and Logic**

   - Step 1: Data Upload and Preparation
     - Users can upload a dataset (CSV or Excel) via Tkinter’s file dialog. Upon upload, the data is filtered by project, discipline, and dashboard name, streamlining the dataset based on user-provided filters.

   - Step 2: User Query Processing

- Each query undergoes intent classification and entity extraction. The chatbot processes intent through RoBERTa and then extracts relevant project data using entity matching.

- Step 3: Query Matching and Response Generation
  - Matched queries are presented back to the user with options for re-querying, updating filters, or exiting. By leveraging Damerau-Levenshtein distance, the system also aligns user input with dataset values, ensuring accurate matches.

4. **Future Improvements and Considerations**
   - Further Model Training: The RoBERTa model can be fine-tuned on more varied datasets to further enhance intent classification.
   - Expansion of UI Functionalities: Tkinter's interface may expand to provide additional features like displaying matched data samples or allowing more complex data filtering.