```
'''
Flowchart of program:
> Import tools and modules from Python
> Set up cache directory
> Load AI model: T5
> Load the correct tokenizer
> Specify task: question and answer pipeline
> Specify question and document folder
> Set up PyQt5 GUI
> When "Run Analysis" is clicked:
    > Start function to extract data from files in a separate thread
    > For each text document in folder:
        > Extract information from text
        > Process through T5 AI engine
    > Update progress bar
> Display results in GUI
'''
# The os library interacts with the operating system
import os
# Transformers library, developed by Hugging Face, is for natural
language processing
# pipeline is a function in the transformers library;
# makes use of the pre-trained models simpler
# AutoTokenizer automatically selects and loads the appropriate tokenizer
# Tokenization converts the raw text into format that can be understood
by machine learning models
# AutoModelForSeq2SeqLM class selects the correct architecture based on
the model being used
from transformers import pipeline, AutoTokenizer, AutoModelForSeq2SeqLM
# os.environ represents environment variables in the current process
# Can read or set variables
# Use Wayland instead of X11
os.environ['QT_QPA_PLATFORM'] = 'wayland'
os.environ['XDG_SESSION_TYPE'] = 'wayland'

# GUI related imports
# PyQt5 provides tools for graphical user interfaces
import sys
from PyQt5.QtWidgets import (QApplication, QMainWindow, QWidget,
QVBoxLayout,
                             QHBoxLayout, QLabel, QLineEdit, QPushButton,
                             QTextEdit, QProgressBar, QFileDialog)
from PyQt5.QtCore import QThread, pyqtSignal, QCoreApplication, Qt

# Set up cache directory
cache_dir = '/home/vboxuser/transformers_cache'
# Creates a directory at the path specified in "cache_dir"
# If it exists already, it will not throw an error
os.makedirs(cache_dir, exist_ok=True)
# The information in variable "cache_dir" is placed into
# the environment variable "TRANSFORMERS_CACHE"
os.environ['TRANSFORMERS_CACHE'] = cache_dir
# Print the location of the cache directory defined previously
print(f"Using cache directory: {cache_dir}")
```

```
# The following code will load the T5 engine (Text-to-Text Transfer
Transformer).
# Can do translation, summarization, question/answer, text classification
########
# Assigns string identifier to the variable "model_name"
model_name = "t5-small"
# Loads and initializes the correct tokenizer associated with the AI
model
# The tokenizer preprocesses the text; is separate from the actual model
tokenizer = AutoTokenizer.from_pretrained(model_name)
# Loads and initializes actual model
# Assigns to variable "model"
model = AutoModelForSeq2SeqLM.from_pretrained(model_name)
# Utilizes the pipeline function to specify the task type ("question and
answer")
# Uses the model and tokenzier
# Assigns to variable "qa_pipeline" which can be used for
question/answering tasks
qa_pipeline = pipeline("text2text-generation", model=model,
tokenizer=tokenizer)
# Confirms that the model has been loaded without errors
print(f"Model {model_name} loaded successfully")

# Class for running analysis in a separate thread
class AnalysisThread(QThread):
    progress_update = pyqtSignal(int)
    analysis_complete = pyqtSignal(list)

    def __init__(self, folder, question):
        QThread.__init__(self)
        self.folder = folder
        self.question = question

    def run(self):
        results = self.extract_breastfeeding_info(self.folder,
self.question)
        self.analysis_complete.emit(results)

    # Function to extract information from documents
    # Parameters are "documents_folder" and "question"
    # "question" is the query that will determine what information is
extracted
    #  Will print out the text file name and its length
    def extract_breastfeeding_info(self, documents_folder, question):

        results = []
        txt_files = [f for f in os.listdir(documents_folder) if
f.endswith('.txt')]
        total_files = len(txt_files)

        for i, document in enumerate(txt_files):
            print(f"Processing file: {document}")
            try:
                with open(os.path.join(documents_folder, document), "r")
as file:
                    context = file.read()
                if not context.strip():
                    print(f"Warning: File {document} is empty!")
                else:
                    # Prints file name and length
                    print(f"File {document} contains text. Length:
{len(context)} characters")

                    # Format input for T5
                    input_text = f"question: {question} context:
{context}"

                    # Use the text-to-text pipeline
                    output = qa_pipeline(input_text, max_length=50,
num_return_sequences=1)

                    answer_text = output[0]['generated_text']

                    # T5 doesn't provide a confidence score
                    # For example, we could check if the answer is not
empty or meets a certain length
                    if len(answer_text.strip()) > 0:
                        results.append((document, answer_text, 1.0))  #
Using 1.0 as a placeholder score
                    else:
                        print(f"Empty answer for question: {question} in
document: {document}")

            except Exception as e:
                print(f"Error reading file {document}: {e}")

            # Update progress
            self.progress_update.emit(int((i + 1) / total_files * 100))

        # Send the "results" list back to the point where the function
was called
        return results

# Create new class called MainWindow for the GUI
class MainWindow(QMainWindow):
    def __init__(self):
        super().__init__()
        self.initUI()

# initUI sets up the basic GUI
    def initUI(self):
        # Window Title
        self.setWindowTitle("Breastfeeding information using local AI
engine T5")
        # Sets position (x axis in pixels, y axis in pixels, width in
pixels, height in pixels
        self.setGeometry(100, 100, 600, 400)
        # Create widget
        central_widget = QWidget()
        # Set widget as central_widget
        self.setCentralWidget(central_widget)
        # Creates vertical box for central_widget
        layout = QVBoxLayout(central_widget)

        # Create a label widget for "Question"
        layout.addWidget(QLabel("Question:"))
        # Create text Entry widget for question input
        self.question_entry = QLineEdit()
        # Adds the text entry to the layout
        layout.addWidget(self.question_entry)

        # Create a label widget for "Folder"
        layout.addWidget(QLabel("Folder:"))
        # Lines up widgest horizontally
        folder_layout = QHBoxLayout()
        # Create text Entry widget for folder input
        # QLineEdit is a one-line text editor
        self.folder_entry = QLineEdit()
        # Add folder entry line to layout
        folder_layout.addWidget(self.folder_entry)
        # Create Button widget called "Select Folder"
        select_folder_button = QPushButton("Select Folder")
        # When the button is clicked, activate the select_folder function
        select_folder_button.clicked.connect(self.select_folder)
        # Add "select_folder_button" to the layout
        folder_layout.addWidget(select_folder_button)
        # Adds "folder_layout" to the larger layout structure
        layout.addLayout(folder_layout)

        # Create Button widget called "Run Analysis"
        run_button = QPushButton("Run Analysis")
        # When the button is clicked, activate the "run_analysis"
function
        run_button.clicked.connect(self.run_analysis)
        # Add the "run_button" to the overall layout
        layout.addWidget(run_button)

        # Create Progressbar widget
        self.progress_bar = QProgressBar()
        # Add the "progress_bar" widget to the overall layout
        layout.addWidget(self.progress_bar)

        # Create Text widget for displaying results
        # QTextEdit() is a multiline text editor
        self.results_text = QTextEdit()
        # Add the "results_text" text box to the overall layout
        layout.addWidget(self.results_text)

    # Function to select folder using file dialog
    # QFileDialog provides interface for selecting folders/files
    def select_folder(self):
        options = QFileDialog.Options()
        options |= QFileDialog.DontUseNativeDialog
        folder = QFileDialog.getExistingDirectory(self, "Select Folder",
options=options)
        if folder:
            print(f"Selected folder: {folder}")  # Debug print
            self.folder_entry.setText(folder)
        else:
            print("No folder selected")

    # Function to run the analysis
    def run_analysis(self):
        try:
            # Put question user entered into "question" variable
            question = self.question_entry.text()
            folder = self.folder_entry.text()

            if not question or not folder:
                print("Please enter both a question and select a
folder.")
                return

            print(f"Starting analysis with question: '{question}' in
folder: '{folder}'")  # Debug print

            self.progress_bar.setValue(0)
            self.results_text.clear()

            # Runs the analysis
            self.analysis_thread = AnalysisThread(folder, question)
            # Updates the progress bar

self.analysis_thread.progress_update.connect(self.update_progress)
            # Sends results to screen

self.analysis_thread.analysis_complete.connect(self.display_results)
            self.analysis_thread.start()
        except Exception as e:
            print(f"Error in run_analysis: {str(e)}")
            traceback.print_exc()

    # Function to update progress bar
    def update_progress(self, value):
        self.progress_bar.setValue(value)

    # Function to display results
    def display_results(self, results):
        if results:
            for doc, answer, _ in results:
                self.results_text.append(f"Document: {doc}\nAnswer:
{answer}\n\n")
        else:
            self.results_text.append("No answers found.")
```

```
# This block only runs if the script is executed directly (not imported)
if __name__ == '__main__':
  # Create the primary application
    app = QApplication(sys.argv)

    # Try to enable high DPI scaling
    if hasattr(QCoreApplication, 'setAttribute'):
        try:
            QCoreApplication.setAttribute(Qt.AA_EnableHighDpiScaling)
        except AttributeError:
            pass  # Attribute doesn't exist, skip it

    # Set up the application style called Fusion
    app.setStyle('Fusion')

    # Create the main window
    main_window = MainWindow()
    # Show the main window
    main_window.show()
    # Start the application and listen for events
    # Keeps window open
    sys.exit(app.exec_())
```