**AUTO GRADER**

**By Ansh Aya**

**09/23/2024**

**Introduction**

AutoGrader is a smart system that uses Artificial Intelligence (AI) and Large Language Models (LLMs) to make grading faster and more accurate. By automating the process, AutoGrader helps teachers save time and ensures consistent results for every student. The system is built using advanced technology from Microsoft Azure, LangChain, and Streamlit, which makes it easy to use and scalable for both small and large institutions. It uses predefined rubrics, which are grading guidelines set by teachers, to ensure fairness in every assessment. By automating grading, educators can focus more on teaching, while students receive quick, reliable feedback. AutoGrader aims to improve the quality of education by making the grading process smoother, faster, and more transparent.

**Business Objectives**

**Improve Operational Efficiency: Grading takes a lot of time, and teachers often struggle to keep up with their workload. AutoGrader automates the entire grading process, allowing teachers to spend more time on lesson planning, student interaction, and improving the overall teaching experience.**

**Reduce Grading Costs: Many schools and institutions spend extra money hiring teaching assistants or staff just to handle grading. With AutoGrader, these extra costs are reduced, as the system can manage grading tasks automatically.**

**Enhances Customer Satisfaction: Both teachers and students benefit from quick, fair, and transparent grading. AutoGrader ensures consistency across all assessments, and students receive detailed feedback, helping them understand their strengths and areas that need improvement.**

**By addressing these key areas, AutoGrader aims to make grading more efficient, affordable, and satisfying for everyone involved.**

**Project Scope**

The scope of AutoGrader extends across various critical aspects of the grading process, covering grading automation, system scalability, and cloud-based deployment. Here is a breakdown of the key areas included within the scope of the project:

**1. Automated Rubric-Based Grading**

The core feature of AutoGrader is its ability to generate consistent grades based on predefined rubrics. These rubrics can be fully customized to align with the specific grading criteria of the educator or institution. AutoGrader’s flexibility allows for grading based on quantitative (points-based) as well as qualitative feedback. This means that the system can handle grading tasks from short-answer questions to long-form essays with ease.

**2. Multi-Format Assignment Support**

AutoGrader is designed to handle assignments in multiple formats including PDF, DOCX, and TXT. These file types are commonly used by students to submit their work, and the system ensures that text extraction is performed accurately regardless of format. The system’s ability to process a variety of file formats makes it capable of operating in different educational environments.

**3. Scalability and Performance Optimization**

AutoGrader is built for scalability, making it capable of handling grading tasks for small classes, as well as large academic institutions with thousands of students. The system is designed to efficiently manage high volumes of submissions while maintaining high-performance standards. AutoGrader continuously gathers feedback from educators and adjusts to improve grading accuracy and speed.

**4. Cloud-Based Deployment**

AutoGrader is deployed on the cloud, which offers several advantages, including enhanced security, scalability, and accessibility. The cloud-based nature of the system ensures that it can be accessed from any location, making it easier for educators to manage grading remotely. The system complies with educational data privacy regulations, ensuring that student data remains secure throughout the grading process.

**Methodology**

The methodology for developing AutoGrader is based on a systematic approach that ensures the grading process is optimized for accuracy, scalability, and ease of use. Below is a detailed overview of the stages involved in the AutoGrader workflow:

**1. Document Loading and Splitting**

The system begins by loading and splitting uploaded assignments into manageable text chunks. Whether the assignment is in PDF, DOCX, or TXT format, AutoGrader extracts the relevant text for processing. This ensures that the system can handle lengthy submissions without any loss of detail or information.

**2. System Prompt Generation**

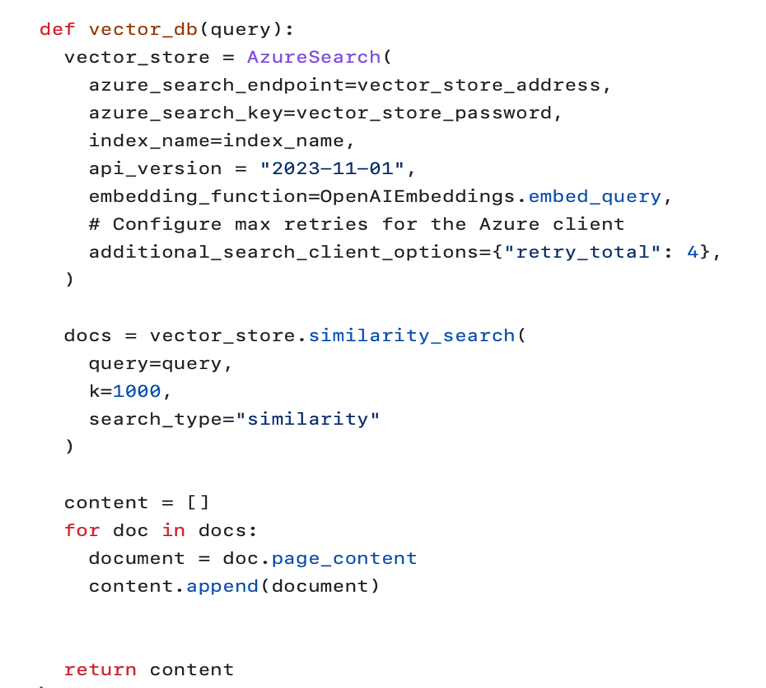
A critical component of AutoGrader is its tailored system prompts. These prompts guide the LLM during the grading process, ensuring that the AI aligns with the predefined rubrics. The system generates custom prompts based on the assignment type, rubric criteria, and any specific instructions provided by the educator. These prompts help ensure that the LLM focuses on key elements of student responses, such as technical accuracy, problem-solving, and coding style, if applicable. This customization allows for highly accurate and context-aware grading.

A screenshot of a computer error

Description automatically generated

**3. Azure AI Search Indexing and Retrieval**

Once the Username is collected, the predefined rubrics stored in Azure AI Search is retrieved through matching username. This enables fast comparisons between student submissions and model rubrics. The **vector-based indexing** allows the system to efficiently search for relevant data points, enabling real-time grading for even large volumes of assignments. This capability ensures that the system can handle grading in an academic environment where hundreds or thousands of rubrics and assignments being submitted simultaneously.



**4. Grading Execution via LangChain’s LLMChain**

The actual grading is performed using **LangChain’s LLMChain**. This is where the AI, guided by the predefined rubric and the custom system prompt, evaluates each submission and generates a score. The LLM evaluates responses based on their content, structure, and alignment with the rubric criteria.

**5. Streamlit User Interface**

The final stage in the methodology involves the **Streamlit user interface**. This user-friendly interface allows educators to upload assignments, view the grading results, and make modifications if necessary. The interface is designed to be intuitive, making it easy for educators to navigate the grading results and provide any additional feedback to students.

**APPLICATION**

A screenshot of a computer

Description automatically generated

Image 1: Collecting Username.

A screenshot of a computer

Description automatically generated

Image 2: Verifying Predefined Rubrics.

A screenshot of a computer

Description automatically generated

Image 3: Requesting Assignment for Grading.

A screenshot of a black screen

Description automatically generated

Image 4: Upload Page.

A screenshot of a computer

Description automatically generated

Image 5: Grades and Feedback - Part 1.

A screenshot of a computer

Description automatically generated

Image 6: Grades and Feedback - Part 2.

**Conclusion**

AutoGrader is an AI-powered system that provides a streamlined, scalable, and efficient solution to grading challenges in modern education. By automating the grading process, AutoGrader delivers consistent and accurate grades based on predefined rubrics, ensuring fairness and transparency. With its support for multiple file formats, real-time feedback, and a user-friendly interface, AutoGrader significantly enhances both the educator and student experience.

The system’s integration with cutting-edge technologies like Azure AI Search, LangChain, and Streamlit makes it robust, secure, and highly scalable. Whether in small classrooms or large academic institutions, AutoGrader can handle the demands of modern education, transforming how assignments are graded. The cloud-based deployment ensures that AutoGrader is accessible from anywhere, making it a valuable tool in today’s increasingly digital and remote educational landscape.

Working of Algorithms (Appendix):

#Import necessary libraries and packages

import streamlit as st

from App.vectordb import vector\_db

from App.chain import get\_chain,get\_scores

from App.chat\_history import format\_chat\_history

from App.document\_handler import process\_document, extract\_answers

from langchain.chat\_models import ChatOpenAI

from langchain.chains import LLMChain, create\_retrieval\_chain

from langchain.prompts.chat import ChatPromptTemplate

from langchain\_community.vectorstores.azuresearch import AzureSearch

from langchain\_openai import OpenAIEmbeddings

from langchain.docstore.document import Document

from langchain\_community.document\_loaders import PyPDFLoader,Docx2txtLoader,TextLoader

from langchain\_text\_splitters import RecursiveCharacterTextSplitter

import tempfile

import os.path

import pathlib

import re

import pdfplumber

import docx

## Set up the environment

# Load secret keys

secrets = st.secrets # Accessing secrets (API keys) stored securely

openai\_api\_key = secrets["openai"]["api\_key"] # Accessing OpenAI API key from secrets

os.environ["OPENAI\_API\_KEY"] = openai\_api\_key # Setting environment for OpenAI

azure\_api\_key = secrets["azure"]["api\_key"]

os.environ["AZURE\_API\_KEY"] = azure\_api\_key

os.environ["AZURE\_AI\_SEARCH\_SERVICE\_NAME"] = "https://ragservices.search.windows.net"

vector\_store\_address = "https://ragservices.search.windows.net"

vector\_store\_password = azure\_api\_key

index\_name = "predefined\_rubrics"

model = "text-embedding-ada-002"

#Save all the variables in Streamlit sessions

if "messages" not in st.session\_state:

st.session\_state.messages = []

if "chat\_history" not in st.session\_state:

st.session\_state.chat\_history = []

if "uploaded\_file" not in st.session\_state:

st.session\_state.uploaded\_file = None

if "rubrics" not in st.session\_state:

st.session\_state.rubrics = None

if "chain" not in st.session\_state:

st.session\_state.chain = None

if "extracted\_answers" not in st.session\_state:

st.session\_state.extracted\_answers = None

if "user\_name" not in st.session\_state:

st.session\_state.user\_name = None

def get\_chain(assignment,predefined\_rubrics,chat\_history):

system\_prompt = """

You are an expert grader, your name is AutoGrader. Your job is to grade {assignment} based on {predefined\_rubrics}.

Start by greeting the user respectfully, collect the name of the user. Save the name of the user in a variable called user\_name.

Display the name to the user exactly in the following format and ask if you can use this user name to fetch their predefined rubrics.

user\_name =

Only after verifying the user\_name, move to the next step.

Next step is to verify {predefined\_rubrics} with the user by displaying whole exact {predefined\_rubrics} to them clearly.

Only after successfully verifying predefined\_rubrics, move to the next step.

Next step is to ask the user to upload the assignment through navigating to uploag page from left hand side.

Go through the {assignment} and highlight the mistakes that user made, make sure you explain all the mistakes in detail with soultions.

Be consistent with the scores and feedbacks generated.

Lastly, ask user if they want any modification or adjustments to the scores generated, if user says no then end the conversation.

Keep the chat history to have memory and do not repeat questions.

chat history: {chat\_history}

"""

prompt = ChatPromptTemplate.from\_messages(

[("system", system\_prompt), ("human", "{input}")]

)

prompt.format\_messages(input = "query", assignment = "st.session\_state.extracted\_answers", predefined\_rubrics = "st.session\_state.rubrics", chat\_history = "st.session\_state.chat\_history")

model\_name = "gpt-4o"

llm = ChatOpenAI(model\_name=model\_name)

st.session\_state.chain = LLMChain(llm = llm,prompt = prompt)

st.session\_state.chat\_active = True

return st.session\_state.chain

def get\_scores(query):

chains = get\_chain(st.session\_state.extracted\_answers,st.session\_state.rubrics,st.session\_state.chat\_history)

response = chains.invoke({"input": query, "assignment": st.session\_state.extracted\_answers, "predefined\_rubrics": st.session\_state.rubrics,"chat\_history": st.session\_state.chat\_history})

try:

answer = response['text']

except:

ans = response['answer']

answer = ans['text']

return answer

def format\_chat\_history(messages):

formatted\_history = ""

for message in messages:

role = "user" if message["role"] == "user" else "Assistant"

content = message["content"]

formatted\_history += f"{role}: {content}\n"

return formatted\_history

# Function to extract text from uploaded files

def process\_document(uploaded\_file):

file\_details = {"filename": uploaded\_file.name, "filetype": uploaded\_file.type}

# Save file to a temporary directory

temp\_dir = tempfile.mkdtemp()

path = os.path.join(temp\_dir, uploaded\_file.name)

with open(path, "wb") as f:

f.write(uploaded\_file.getvalue())

# Determine file extension

file\_extension = uploaded\_file.name.split(".")[-1].lower()

# Load document based on its extension

if file\_extension == "txt":

loader = TextLoader(path)

elif file\_extension == "pdf":

loader = PyPDFLoader(path)

elif file\_extension == "docx":

loader = Docx2txtLoader(path)

else:

st.error("Unsupported file type.")

return None

# Load documents and split text

docs = loader.load()

text\_splitter = RecursiveCharacterTextSplitter(chunk\_size=1000, chunk\_overlap=0)

documents = text\_splitter.split\_documents(docs)

return documents

# Function to extract answers using regex patterns

def extract\_answers(uploaded\_file):

file\_details = {"filename": uploaded\_file.name, "filetype": uploaded\_file.type}

# Save file to a temporary directory

temp\_dir = tempfile.mkdtemp()

path = os.path.join(temp\_dir, uploaded\_file.name)

with open(path, "wb") as f:

f.write(uploaded\_file.getvalue())

# Determine file extension

file\_extension = uploaded\_file.name.split(".")[-1].lower()

if file\_extension == "txt":

text = uploaded\_file.read().decode('utf-8')

elif file\_extension == "pdf":

with pdfplumber.open(uploaded\_file) as pdf:

pages = [page.extract\_text() for page in pdf.pages]

text = "\n".join(pages)

elif file\_extension == "docx":

pages = docx.Document(uploaded\_file)

text = "\n".join([para.text for para in pages.paragraphs])

extracted\_answers = []

# Pattern

pattern = r"(Question\s\*\d:.\*?)(Answer\s\*\d:.\*)"

# Use re.search to iterate through the matches

search\_result = re.search(pattern, text, re.DOTALL)

if search\_result:

# Extract the question and answer from the matched groups

question = search\_result.group(1).strip()

extracted\_answers.append(question)

answer = search\_result.group(2).strip()

answers\_cleaned = re.sub(r'(^#)', r'\\#', answer, flags=re.MULTILINE)

extracted\_answers.append(answers\_cleaned)

else:

extracted\_answers = st.write("Answers not found")

return extracted\_answers

def vector\_db(query):

vector\_store = AzureSearch(

azure\_search\_endpoint=vector\_store\_address,

azure\_search\_key=vector\_store\_password,

index\_name=index\_name,

api\_version = "2023-11-01",

embedding\_function=OpenAIEmbeddings.embed\_query,

# Configure max retries for the Azure client

additional\_search\_client\_options={"retry\_total": 4},

)

docs = vector\_store.similarity\_search(

query=query,

k=1000,

search\_type="similarity"

)

content = []

for doc in docs:

document = doc.page\_content

content.append(document)

return content

# Streamlit app interface

st.title("Automatic Grading System")

# Multi-page navigation

page = st.sidebar.selectbox("Choose a page", ["Home", "Upload Assignment"])

if page == "Home":

# Display chat messages from history on app rerun

for message in st.session\_state.messages:

with st.chat\_message(message["role"]):

st.markdown(message["content"])

if query := st.chat\_input("Ask your question here"):

# Display user message in chat message container

with st.chat\_message("user"):

st.markdown(query)

# Add user message to chat history

st.session\_state.messages.append({"role": "user", "content": query})

st.session\_state.chat\_history = format\_chat\_history(st.session\_state.messages)

answer = get\_scores(query)

#Extract user name from chat

pattern = r'user\_name\s\*=\s\*"?(\w+)"?'

search\_results = re.search(pattern, answer, re.DOTALL)

if search\_results:

st.session\_state.user\_name = search\_results.group(0)

if st.session\_state.user\_name:

#Fetch predefined rubrics using user name

st.session\_state.rubrics = vector\_db(st.session\_state.user\_name)

# Display assistant response in chat message container

with st.chat\_message("assistant"):

st.markdown(answer)

# Add assistant response to chat history

st.session\_state.messages.append({"role": "assistant", "content": answer})

# Button to clear chat messages

def clear\_messages():

st.session\_state.messages = []

st.button("Clear", help = "Click to clear the chat", on\_click=clear\_messages)

if page == "Upload Assignment":

# File uploader

uploaded\_file = st.file\_uploader("Upload your assignment", type=["txt", "pdf", "docx"])

# Document Handler:

if uploaded\_file:

# Read file content

st.session\_state.uploaded\_file = process\_document(uploaded\_file)

# Extract answers using regex patterns

st.session\_state.extracted\_answers = extract\_answers(uploaded\_file)

st.write("Assignment Uploaded Successfully")