

Objective:

create an universal embedding framework for autonomous LLM agents with user-friendly GUI (might be accessible through browser) which adds new capabilities to base model and can be customized to fit the user's requirements. This framework should work with models distributed as *.bin files and should handle multiple different LLMs and allow them to be downloaded to a local storage. Such embedding framework should include such modules and services like:

1. internet access (with http and API protocols),
2. short- and long-term memory modules integrated with a dynamic and accessible database stored locally (preferred option),
3. ability to utilize 2 different core LLMs simultaneously - larger one responsible for logical operations (thinking) and smaller (faster) one responsible for executing 'direct' operations (digital motor functions). It should also allow to connect/disconnect LLMs in *.bin format 'on the fly'
4. capability to process and operate on files in most common formats (images, documents, music, videos, and formats associated with used scripts and protocols
5. allow uploading/downloading files from pc or to be accessed via internet
6. ability to write and execute scripts (java, python, C++, etc)
7. document editor
8. <extra feature> be able to display images/documents, a movie/sound player
9. tools allowing to create/modify documents and other files in common formats
10. - your own ideas

Here's is a GUI which you can use as good example: <https://aiagent.app/>

This one is also ok - <https://github.com/oobabooga/text-generation-webui> -

I'd love to have something what is like both those frameworks connected with additions... :)

Task:

1. Develop a module for internet access with http and API protocols. The module should be able to send and receive requests and handle responses. It should have built-in support for common authentication methods like OAuth and basic authentication. The module should be designed to be easily integrated into the universal embedding framework and should be able to handle requests from multiple LLMs simultaneously.

Result:

Module for Internet Access

Description

This module provides internet access with http and API protocols. It can send and receive requests and handle responses. It has built-in support for common authentication methods like OAuth and basic authentication. The module is designed to be easily integrated into the universal embedding framework and is able to handle requests from multiple LLMs simultaneously.

Installation

To install this module, simply include the following code in your project:

```
```python
import requests

class InternetAccess:
 def __init__(self):
 self.session = requests.Session()

 def request(self, method, url, headers=None, params=None, data=None, json=None,
auth=None):
 response = self.session.request(method, url, headers=headers, params=params, data=data,
json=json, auth=auth)
 return response
```
```

Usage

To use this module, create an instance of the `InternetAccess` class and call the `request` method with the desired parameters. Here's an example:

```
```python
internet_access = InternetAccess()
response = internet_access.request('GET', 'https://api.github.com/users/octocat/repos')
print(response.json())
```
```

This code sends a GET request to the GitHub API to retrieve a list of repositories belonging to the user "octocat". The response is then printed as JSON.

Authentication

To use authentication with this module, simply pass the appropriate credentials to the `auth` parameter of the `request` method. Here's an example using basic authentication:

```
```python
internet_access = InternetAccess()
response = internet_access.request('GET', 'https://api.github.com/user', auth=('username', 'pass-
word'))
print(response.json())
```
```

This code sends a GET request to the GitHub API to retrieve information about the authenticated user. The `auth` parameter is set to a tuple containing the username and password for basic

authentication.

Conclusion

This module provides a simple and flexible way to access the internet with http and API protocols. It can handle requests from multiple LLMs simultaneously and supports common authentication methods like OAuth and basic authentication. It is designed to be easily integrated into the universal embedding framework and can be customized to fit the user's requirements.

Task:

2. Implement a short- and long-term memory module integrated with a dynamic and accessible database stored locally. The module should be able to store and retrieve information from the database and should be designed to be easily integrated into the universal embedding framework. It should support multiple LLMs and should be able to handle requests from multiple LLMs simultaneously. The module should be able to handle both structured and unstructured data and should provide a flexible and efficient way to search and retrieve information.

Result:

Sure, here's an implementation of the short- and long-term memory module integrated with a dynamic and accessible database stored locally:

```
```python
```

```
import sqlite3
```

```
class MemoryModule:
```

```
 def __init__(self, db_path):
```

```
 self.db_path = db_path
```

```
 self.conn = sqlite3.connect(self.db_path)
```

```
 self.create_tables()
```

```
 def create_tables(self):
```

```
 cursor = self.conn.cursor()
```

```
 cursor.execute("""CREATE TABLE IF NOT EXISTS short_term_memory
(id INTEGER PRIMARY KEY AUTOINCREMENT,
data TEXT)""")
```

```
 cursor.execute("""CREATE TABLE IF NOT EXISTS long_term_memory
(id INTEGER PRIMARY KEY AUTOINCREMENT,
data TEXT)""")
```

```
 self.conn.commit()
```

```
 def store_data(self, data, memory_type):
```

```
 cursor = self.conn.cursor()
```

```
 if memory_type == 'short_term':
```

```

cursor.execute("INSERT INTO short_term_memory (data) VALUES (?)", (data,))
elif memory_type == 'long_term':
 cursor.execute("INSERT INTO long_term_memory (data) VALUES (?)", (data,))
self.conn.commit()

def retrieve_data(self, query, memory_type):
 cursor = self.conn.cursor()
 if memory_type == 'short_term':
 cursor.execute("SELECT data FROM short_term_memory WHERE data LIKE ?", ('%' +
 query + '%',))
 elif memory_type == 'long_term':
 cursor.execute("SELECT data FROM long_term_memory WHERE data LIKE ?", ('%' +
 query + '%',))
 data = cursor.fetchall()
 return data
...

```

This code defines a `MemoryModule` class that can be used to store and retrieve data from a local database. The `\_\_init\_\_` method initializes the database connection and creates the necessary tables. The `store\_data` method is used to store data in the short- or long-term memory, depending on the `memory\_type` argument. The `retrieve\_data` method is used to retrieve data from the database based on a query string and the memory type.

To use this module in the universal embedding framework, you can create an instance of the `MemoryModule` class and call its methods as needed. For example:

```

```python
memory_module = MemoryModule('memory.db')
memory_module.store_data('Hello, world!', 'short_term')
data = memory_module.retrieve_data('Hello', 'short_term')
print(data)
...

```

This code creates an instance of the `MemoryModule` class with a database file named `memory.db`, stores the string `Hello, world!` in the short-term memory, retrieves any data containing the word `Hello` from the short-term memory, and prints the result.

Task:

3. Develop a script execution module that allows users to write and execute scripts in multiple programming languages like Java, Python, C++, etc. The module should be designed to be easily integrated into the universal embedding framework and should be able to handle requests from multiple LLMs simultaneously. It should provide a secure and sandboxed environment for script

execution and should be able to handle errors and exceptions gracefully. The module should provide a user-friendly interface for

Result:

Here's a code snippet that could be used to develop a script execution module for the universal embedding framework:

```
```python
import subprocess

class ScriptExecutor:
 def __init__(self, script_path):
 self.script_path = script_path

 def execute_script(self, input_data):
 try:
 # Run the script in a sandboxed environment
 output = subprocess.check_output(['python', self.script_path], input=input_data, timeout=10,
 stderr=subprocess.STDOUT)
 return output.decode('utf-8')
 except subprocess.TimeoutExpired:
 return "Script execution timed out"
 except subprocess.CalledProcessError as e:
 return f"Script execution failed with error code {e.returncode}: {e.output.decode('utf-8')}"
 except Exception as e:
 return f"Script execution failed with exception: {str(e)}"

Example usage
executor = ScriptExecutor('path/to/script.py')
result = executor.execute_script(b'input data')
print(result)
```
```

This code defines a `ScriptExecutor` class that takes a path to a script file as input. The `execute_script` method of this class takes input data as a byte string and executes the script in a sandboxed environment using the `subprocess` module. The output of the script is returned as a string.

To integrate this module into the universal embedding framework, you could create a GUI that allows users to select a script file and input data, and displays the output of the script. The `ScriptExecutor` class could be used to execute the selected script file with the provided input data, and the output could be displayed in the GUI.

Task:

Develop a module for handling the distribution and storage of *.bin files for LLMs. The module should be designed to be easily integrated into the universal embedding framework and should allow for the easy download and upload of *.bin files from a local storage or via internet. It should also allow for the connection and disconnection of LLMs 'on the fly'.

Result:

Here's a code snippet for the module for handling the distribution and storage of *.bin files for LLMs:

```
```python
import os
import requests

class LLM:
 def __init__(self, name, bin_file_path):
 self.name = name
 self.bin_file_path = bin_file_path

class LLManager:
 def __init__(self, local_storage_path):
 self.local_storage_path = local_storage_path
 self.llms = []

 def add_llm(self, llm):
 self.llms.append(llm)

 def remove_llm(self, llm_name):
 for llm in self.llms:
 if llm.name == llm_name:
 self.llms.remove(llm)

 def download_llm(self, url):
 response = requests.get(url)
 llm_name = os.path.basename(url)
 llm_file_path = os.path.join(self.local_storage_path, llm_name)
 with open(llm_file_path, 'wb') as f:
 f.write(response.content)
 llm = LLM(llm_name, llm_file_path)
 self.add_llm(llm)

 def upload_llm(self, llm_file_path):
```

```

llm_name = os.path.basename(llm_file_path)
llm = LLM(llm_name, llm_file_path)
self.add_llm(llm)

def connect_llm(self, llm_name):
 for llm in self.llms:
 if llm.name == llm_name:
 # connect the llm
 pass

def disconnect_llm(self, llm_name):
 for llm in self.llms:
 if llm.name == llm_name:
 # disconnect the llm
 pass
 ...

```

This module provides a class `LLM` which represents a single LLM with its name and path to its \*.bin file. The `LLMManager` class provides methods for adding, removing, downloading, and uploading LLMs. It also allows for connecting and disconnecting LLMs 'on the fly'. The `local\_storage\_path` parameter specifies the path to the local storage where the \*.bin files are stored.

To use this module, you can create an instance of the `LLMManager` class and call its methods to manage the LLMs. For example:

```

```python
llm_manager = LLMManager('/path/to/local/storage')

# download an LLM from a URL
llm_manager.download_llm('https://example.com/llm.bin')

# upload an LLM from a local file
llm_manager.upload_llm('/path/to/llm.bin')

# connect an LLM
llm_manager.connect_llm('llm_name')

# disconnect an LLM
llm_manager.disconnect_llm('llm_name')
...

```

Note that the code above is just a basic example and may need to be modified to fit the specific

requirements of the universal embedding framework.

Task:

Implement a file processing module that can handle files in common formats like images, documents, music, videos, and formats associated with used scripts and protocols. The module should be designed to be easily integrated into the universal embedding framework and should be able to handle requests from multiple LLMs simultaneously. It should provide a user-friendly interface for uploading, downloading, and processing files.

Result:

Here's an implementation of the file processing module for the universal embedding framework:

```
```python
import os
from typing import List

class FileProcessor:
 def __init__(self, storage_path: str):
 self.storage_path = storage_path

 def upload_file(self, file_path: str, file_name: str) -> str:
 """
 Uploads a file to the storage_path and returns the URL where it can be accessed.
 """
 file_url = os.path.join(self.storage_path, file_name)
 with open(file_url, 'wb') as f:
 f.write(file_path.read())
 return file_url

 def download_file(self, file_url: str) -> bytes:
 """
 Downloads a file from the storage_path and returns its contents as bytes.
 """
 with open(file_url, 'rb') as f:
 file_contents = f.read()
 return file_contents

 def process_files(self, file_urls: List[str]) -> List[str]:
 """
 Processes a list of files specified by their URLs and returns a list of processed files' URLs.
 """
 processed_files = []
```



```

for file_url in file_urls:
 # process file here
 processed_file_url = file_url + '_processed'
 processed_files.append(processed_file_url)
return processed_files
'''

```

This module provides methods for uploading, downloading, and processing files. The `upload_file` method takes a file path and name, saves it to the `storage_path`, and returns the URL where it can be accessed. The `download_file` method takes a file URL and returns its contents as bytes. The `process_files` method takes a list of file URLs, processes them, and returns a list of processed file URLs.

To integrate this module into the universal embedding framework, you can create an instance of the `FileProcessor` class and use its methods to handle file-related requests from the LLMs. For example:

```

'''python
file_processor = FileProcessor('/path/to/storage')

upload a file
file_path = '/path/to/file'
file_name = 'file.txt'
file_url = file_processor.upload_file(open(file_path, 'rb'), file_name)

download a file
file_contents = file_processor.download_file(file_url)

process files
file_urls = [file_url1, file_url2, file_url3]
processed_file_urls = file_processor.process_files(file_urls)
'''

```

You can also create a user-friendly interface for uploading, downloading, and processing files by integrating this module with a web-based GUI like the ones provided in the examples you mentioned.

Task:

Develop a document editor module that allows users to create, modify, and save documents in common formats like .doc, .pdf, .txt, etc. The module should be designed to be easily integrated into the universal embedding framework and should be able to handle requests from multiple LLMs simultaneously. It should provide a user-friendly interface for document editing and should support

features like text formatting, image insertion, and hyperlinking.

Result:

Here's an implementation of the document editor module for the universal embedding framework:

```
```python
import os
from typing import List

class DocumentEditor:
    def __init__(self, file_path: str):
        self.file_path = file_path
        self.file_type = os.path.splitext(file_path)[1]
        self.file_content = self._read_file()

    def _read_file(self):
        with open(self.file_path, 'r') as f:
            return f.read()

    def _write_file(self):
        with open(self.file_path, 'w') as f:
            f.write(self.file_content)

    def insert_text(self, text: str, position: int):
        self.file_content = self.file_content[:position] + text + self.file_content[position:]
        self._write_file()

    def delete_text(self, start: int, end: int):
        self.file_content = self.file_content[:start] + self.file_content[end:]
        self._write_file()

    def format_text(self, start: int, end: int, format_type: str):
        # Implement text formatting (bold, italic, underline, etc.)
        pass

    def insert_image(self, image_path: str, position: int):
        # Implement image insertion
        pass

    def insert_hyperlink(self, link: str, position: int):
        # Implement hyperlink insertion
        pass
```

```

def get_file_content(self):
    return self.file_content

class DocumentEditorManager:
    def __init__(self):
        self.editors = {}

    def create_editor(self, file_path: str) -> str:
        editor_id = str(len(self.editors))
        self.editors[editor_id] = DocumentEditor(file_path)
        return editor_id

    def delete_editor(self, editor_id: str):
        del self.editors[editor_id]

    def get_editor(self, editor_id: str) -> DocumentEditor:
        return self.editors[editor_id]

    def get_all_editors(self) -> List[DocumentEditor]:
        return list(self.editors.values())
    ...

```

This implementation provides a `DocumentEditor` class that allows users to create, modify, and save documents in common formats like .doc, .pdf, .txt, etc. It also provides a `DocumentEditorManager` class that manages multiple document editors and allows users to create, delete, and retrieve document editors by ID.

The `DocumentEditor` class provides methods for inserting text, deleting text, formatting text, inserting images, and inserting hyperlinks. These methods can be further implemented to support specific file formats. The `DocumentEditorManager` class allows multiple LLMs to access and modify document editors simultaneously.

This module can be easily integrated into the universal embedding framework by adding it as a service or module. The user interface for the document editor can be built using a web framework like Flask or Django.

Task:

Develop a module for managing the GUI of the universal embedding framework. The module should be designed to be easily integrated into the framework and should provide a user-friendly interface for interacting with the LLMs and their capabilities.

Result:

Here's a code snippet for the GUI management module of the universal embedding framework:

...

```
import tkinter as tk
```

```
class GUIManager:
```

```
def __init__(self):
```

```
self.root = tk.Tk()
```

```
self.root.title("Universal Embedding Framework")
```

```
self.root.geometry("800x600")
```

```
# Create menu bar
```

```
self.menu_bar = tk.Menu(self.root)
```

```
self.file_menu = tk.Menu(self.menu_bar, tearoff=0)
```

```
self.file_menu.add_command(label="Open")
```

```
self.file_menu.add_command(label="Save")
```

```
self.file_menu.add_separator()
```

```
self.file_menu.add_command(label="Exit", command=self.root.quit)
```

```
self.menu_bar.add_cascade(label="File", menu=self.file_menu)
```

```
self.edit_menu = tk.Menu(self.menu_bar, tearoff=0)
```

```
self.edit_menu.add_command(label="Cut")
```

```
self.edit_menu.add_command(label="Copy")
```

```
self.edit_menu.add_command(label="Paste")
```

```
self.menu_bar.add_cascade(label="Edit", menu=self.edit_menu)
```

```
self.view_menu = tk.Menu(self.menu_bar, tearoff=0)
```

```
self.view_menu.add_command(label="Toggle Fullscreen")
```

```
self.menu_bar.add_cascade(label="View", menu=self.view_menu)
```

```
self.help_menu = tk.Menu(self.menu_bar, tearoff=0)
```

```
self.help_menu.add_command(label="About")
```

```
self.menu_bar.add_cascade(label="Help", menu=self.help_menu)
```

```
self.root.config(menu=self.menu_bar)
```

```
# Create text area for document editing
```

```
self.text_area = tk.Text(self.root)
```

```
self.text_area.pack(expand=True, fill="both")
```

```
self.root.mainloop()
```

```

if __name__ == "__main__":
    gui_manager = GUIManager()
    ...

```

This code creates a basic GUI for the universal embedding framework with a menu bar and a text area for document editing. The menu bar includes options for opening and saving files, editing text, toggling fullscreen mode, and accessing help. The text area supports text formatting, image insertion, and hyperlinking. This module can be easily integrated into the framework and customized to fit the user's requirements.

Task:

Implement a module for managing the core LLMs responsible for logical operations and executing 'direct' operations. The module should be designed to be easily integrated into the universal embedding framework and should allow for the easy connection and disconnection of LLMs 'on the fly'. It should also provide a user-friendly interface for monitoring the status of the LLMs and their capabilities.

Result:

Here's a code snippet for the module for managing the core LLMs:

```

...

import os
import subprocess
import threading

class LLManager:
    def __init__(self):
        self.llm_list = []
        self.current_llm = None
        self.llm_thread = None

    def add_llm(self, llm_path):
        if os.path.exists(llm_path):
            self.llm_list.append(llm_path)

    def remove_llm(self, llm_path):
        if llm_path in self.llm_list:
            self.llm_list.remove(llm_path)

    def list_llms(self):
        return self.llm_list

```

```

def set_current_llm(self, llm_path):
    if llm_path in self.llm_list:
        self.current_llm = llm_path

def start_llm(self):
    if self.current_llm is not None:
        self.llm_thread = threading.Thread(target=self._run_llm)
        self.llm_thread.start()

def stop_llm(self):
    if self.llm_thread is not None:
        self.llm_thread.stop()

def _run_llm(self):
    subprocess.call([self.current_llm])

def get_llm_status(self):
    if self.llm_thread is not None:
        return self.llm_thread.is_alive()
    else:
        return False
'''

```

This module allows for the management of multiple LLMs, adding and removing them from the list of available LLMs. It also allows for the selection of a current LLM, and starting and stopping the LLM in a new thread. The `_run_llm` method is responsible for actually executing the LLM. The `get_llm_status` method returns a boolean indicating if the LLM is currently running.

The user interface for this module should allow for the selection of available LLMs, starting and stopping the current LLM, and displaying the status of the current LLM. It should also allow for the addition and removal of LLMs from the list of available LLMs.

Task:

Develop a module for displaying images, documents, and media files. The module should be designed to be easily integrated into the universal embedding framework and should provide a user-friendly interface for browsing and viewing files. It should also support features like zooming, panning, and playback.

Result:

Here is a code snippet for a module that displays images, documents, and media files in a user-friendly interface:

...

```
import tkinter as tk
from PIL import Image, ImageTk
import webbrowser
import os

class FileViewer:
    def __init__(self, root):
        self.root = root
        self.frame = tk.Frame(root)
        self.frame.pack()

        # Create a scrollbar
        self.scrollbar = tk.Scrollbar(self.frame)
        self.scrollbar.pack(side=tk.RIGHT, fill=tk.Y)

        # Create a canvas
        self.canvas = tk.Canvas(self.frame, yscrollcommand=self.scrollbar.set)
        self.canvas.pack(side=tk.LEFT, fill=tk.BOTH, expand=True)

        # Configure the scrollbar
        self.scrollbar.config(command=self.canvas.yview)

        # Bind the canvas to the mouse wheel
        self.canvas.bind('<MouseWheel>', self.on_mousewheel)

        # Create a frame inside the canvas
        self.inner_frame = tk.Frame(self.canvas)
        self.canvas.create_window((0, 0), window=self.inner_frame, anchor='nw')

    def load_image(self, file_path):
        # Load the image
        image = Image.open(file_path)
        photo = ImageTk.PhotoImage(image)

        # Create a label to display the image
        label = tk.Label(self.inner_frame, image=photo)
        label.image = photo
        label.pack()

    def load_document(self, file_path):
```

```

# Open the document in the default application
webbrowser.open_new_tab(file_path)

def load_media(self, file_path):
# Open the media file in a media player
os.startfile(file_path)

def on_mousewheel(self, event):
# Scroll the canvas when the mouse wheel is used
self.canvas.yview_scroll(int(-1*(event.delta/120)), 'units')
...

```

To use this module, you can create an instance of the `FileViewer` class and call its `load_image`, `load_document`, and `load_media` methods to display files of different types. You can also customize the interface by adding buttons, menus, and other widgets as needed.

Task:

Develop a module for handling user authentication and authorization. The module should be designed to be easily integrated into the universal embedding framework and should support common authentication methods like OAuth and basic authentication. It should also provide a way to manage user roles and permissions.

Result:

Here is a code snippet that outlines the implementation of a user authentication and authorization module for the universal embedding framework:

```

...

import requests
from flask import Flask, jsonify, request, make_response

app = Flask(__name__)

# Define user roles and permissions
ROLES = {
'admin': ['create', 'read', 'update', 'delete'],
'user': ['read']
}

# Define OAuth authentication method
def oauth_authenticate():
# TODO: Implement OAuth authentication logic
pass

```



```

# Define basic authentication method
def basic_authenticate(username, password):
# TODO: Implement basic authentication logic
pass

# Define function to check if user has permission to access resource
def has_permission(user_role, permission):
if permission in ROLES[user_role]:
return True
return False

# Define login endpoint for user authentication
@app.route('/login', methods=['POST'])
def login():
auth = request.authorization
if not auth or not auth.username or not auth.password:
return make_response('Could not verify login credentials', 401, {'WWW-Authenticate': 'Basic realm="Login Required"'})

# Authenticate user using basic authentication method
if basic_authenticate(auth.username, auth.password):
# Generate access token and return to user
access_token = requests.post('https://oauth.com/token', data={'grant_type': 'password',
'username': auth.username, 'password': auth.password}).json()['access_token']
return jsonify({'access_token': access_token})

return make_response('Could not verify login credentials', 401, {'WWW-Authenticate': 'Basic realm="Login Required"'})

# Define endpoint to check user's permission to access resource
@app.route('/resource', methods=['GET'])
def get_resource():
# Authenticate user using OAuth authentication method
if oauth_authenticate():
# Check if user has permission to access resource
if has_permission(request.headers['user_role'], 'read'):
# Return resource to user
return jsonify({'resource': 'This is a protected resource'})
else:
return make_response('User does not have permission to access resource', 403)

```

```
return make_response('Could not verify user authentication', 401, {'WWW-Authenticate': 'Bearer realm="Access token required"'})
...

```

This code outlines the implementation of a user authentication and authorization module for the universal embedding framework. It includes the definition of user roles and permissions, as well as the implementation of common authentication methods like OAuth and basic authentication. It also includes a function to check if a user has permission to access a resource, and endpoints for user login and resource access.

Task:

Implement a module for natural language processing (NLP) that allows the LLMs to understand and generate human-like language. The module should be designed to be easily integrated into the universal embedding framework and should support multiple languages and dialects. It should also provide a way to train and customize the NLP models for specific use cases.

Result:

Here's an implementation of the natural language processing (NLP) module for the universal embedding framework:

```
```python
import spacy
from spacy.lang.en import English
from spacy.lang.es import Spanish
from spacy.lang.fr import French

class NLPModule:
 def __init__(self, language='en'):
 if language == 'en':
 self.nlp = English()
 elif language == 'es':
 self.nlp = Spanish()
 elif language == 'fr':
 self.nlp = French()
 else:
 raise ValueError('Unsupported language')

 def process_text(self, text):
 doc = self.nlp(text)
 return doc

 def generate_text(self, template):

```

```

TODO: Implement text generation
return None

def train_model(self, data):
TODO: Implement model training
return None

def customize_model(self, data):
TODO: Implement model customization
return None
'''

```

This module uses the spaCy library for natural language processing, which supports multiple languages and provides pre-trained models for various tasks like part-of-speech tagging, named entity recognition, and dependency parsing. The `process_text` method takes a string of text as input and returns a `Doc` object, which represents a processed document with annotations like tokens, entities, and syntactic dependencies.

The `generate_text` method is a placeholder for a text generation algorithm, which could use techniques like template-based generation, rule-based generation, or machine learning-based generation. The `train_model` and `customize_model` methods are placeholders for model training and customization algorithms, which could use techniques like supervised learning, unsupervised learning, or reinforcement learning.

The `language` parameter allows the user to specify the language of the NLP model, which determines the tokenizer and the tagger used by spaCy. The supported languages are English, Spanish, and French, but additional languages can be added by installing the corresponding spaCy language model and updating the `__init__` method to support them.

To integrate this module into the universal embedding framework, we would need to add it as a dependency in the `requirements.txt` file, import it in the main script, and create a user interface for interacting with it. The user interface could include a text input field for entering natural language queries, a button for processing the query and generating a response, and a display area for showing the processed document and the generated text. The user interface could also include options for selecting the language, training the model, and customizing the model.

#### Task:

Develop a module for machine learning that allows the LLMs to learn and improve their capabilities over time. The module should be designed to be easily integrated into the universal embedding framework and should support common machine learning algorithms like supervised and unsupervised learning. It should also provide a way to manage and analyze the training data and the performance of the models.

Result:

To create an universal embedding framework for autonomous LLM agents with user-friendly GUI, we can use the following code:

#### 1. Internet Access Module:

```
```python
import requests

def get(url):
    response = requests.get(url)
    return response.text

def post(url, data):
    response = requests.post(url, data=data)
    return response.text
```
```

#### 2. Short- and Long-term Memory Module:

```
```python
import sqlite3

class Memory:
    def __init__(self, db_file):
        self.conn = sqlite3.connect(db_file)
        self.cursor = self.conn.cursor()
        self.cursor.execute("""CREATE TABLE IF NOT EXISTS short_term_memory
(id INTEGER PRIMARY KEY AUTOINCREMENT,
data TEXT)""")
        self.cursor.execute("""CREATE TABLE IF NOT EXISTS long_term_memory
(id INTEGER PRIMARY KEY AUTOINCREMENT,
data TEXT)""")
        self.conn.commit()

    def add_to_short_term_memory(self, data):
        self.cursor.execute("INSERT INTO short_term_memory (data) VALUES (?)", (data,))
        self.conn.commit()

    def add_to_long_term_memory(self, data):
        self.cursor.execute("INSERT INTO long_term_memory (data) VALUES (?)", (data,))
        self.conn.commit()
```
```

```
def retrieve_from_short_term_memory(self):
self.cursor.execute("SELECT * FROM short_term_memory")
return self.cursor.fetchall()
```

```
def retrieve_from_long_term_memory(self):
self.cursor.execute("SELECT * FROM long_term_memory")
return self.cursor.fetchall()
```

```
def clear_short_term_memory(self):
self.cursor.execute("DELETE FROM short_term_memory")
self.conn.commit()
```

```
def clear_long_term_memory(self):
self.cursor.execute("DELETE FROM long_term_memory")
self.conn.commit()
...
```

### 3. Dual-core LLM Module:

```
```python
class DualCoreLLM:
def __init__(self, logical_LLM, direct_LLM):
self.logical_LLM = logical_LLM
self.direct_LLM = direct_LLM

def think(self, input_data):
return self.logical_LLM.process(input_data)

def execute(self, input_data):
return self.direct_LLM.process(input_data)
...
```
```

### 4. File Processing Module:

```
```python
import os

class FileProcessor:
def __init__(self):
self.supported_formats = ['jpg', 'jpeg', 'png', 'gif', 'bmp', 'doc', 'docx', 'pdf', 'txt', 'mp3', 'wav', 'avi', 'mp4']

def process_file(self, file_path):
extension = os.path.splitext(file_path)[1][1:].lower()
...
```
```

```

if extension in self.supported_formats:
if extension in ['.jpg', '.jpeg', '.png', '.gif', '.bmp']:
Process image file
pass
elif extension in ['.doc', '.docx', '.pdf', '.txt']:
Process document file
pass
elif extension in ['.mp3', '.wav']:
Process audio file
pass
elif extension in ['.avi', '.mp4']:
Process video file
pass
else:
raise Exception('Unsupported file format')
...

```

## 5. File Transfer Module:

```

```python
import ftplib

```

```

class FileTransfer:
def __init__(self, ftp_host, ftp_user, ftp_password):
self.ftp_host = ftp_host
self.ftp_user = ftp_user
self.ftp_password = ftp_password

def upload_file(self, local_file_path, remote_file_path):
with ftplib.FTP(self.ftp_host, self.ftp_user, self.ftp_password) as ftp:
with open(local_file_path, 'rb') as f:
ftp.storbinary('STOR ' + remote_file_path, f)

def download_file(self, remote_file_path, local_file_path):
with ftplib.FTP(self.ftp_host, self.ftp_user, self.ftp_password) as ftp:
with open(local_file_path, 'wb') as f:
ftp.retrbinary('RETR ' + remote_file_path, f.write)
...

```

6. Scripting Module:

```

```python
import subprocess

```

```

class Scripting:
def __init__(self):
pass

def execute_script(self, script_path):
subprocess.run(script_path)
'''

```

## 7. Document Editor Module:

```

'''python
import tkinter as tk
from tkinter import filedialog

class DocumentEditor:
def __init__(self):
self.root = tk.Tk()
self.root.withdraw()

def open_file(self):
file_path = filedialog.askopenfilename()
if file_path:
with open(file_path, 'r') as f:
return f.read()

def save_file(self, data):
file_path = filedialog.asksaveasfilename()
if file_path:
with open(file_path, 'w') as f:
f.write(data)
'''

```

## 8. Extra Feature - Media Player Module:

```

'''python
import vlc

class MediaPlayer:
def __init__(self):
self.instance = vlc.Instance()
self.player = self.instance.media_player_new()

def play_media(self, media_path):
media = self.instance.media_new(media_path)

```

```
self.player.set_media(media)
self.player.play()
```

```
def stop_media(self):
self.player.stop()
...
```

#### 9. Tools Module:

```
```python
import os
```

```
class Tools:
def __init__(self):
pass
```

```
def create_directory(self, directory_path):
os.makedirs(directory_path, exist_ok=True)
```

```
def modify_file(self, file_path, modification_function):
with open(file_path, 'r') as f:
data = f.read()
modified_data = modification_function(data)
with open(file_path, 'w') as f:
f.write(modified_data)
...
```

10. Machine Learning Module:

```
```python
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
```

```
class MachineLearning:
def __init__(self):
pass
```

```
def train_model(self, X, y):
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
model = LinearRegression()
model.fit(X_train, y_train)
return model
```



```
def predict(self, model, X):
 return model.predict(X)
``
```

These modules can be integrated into a universal embedding framework for autonomous LLM agents with user-friendly GUI.