# AI-based Web Application development and deployment

Estimated Time: 60 minutes

## Overview

In this project, we make use of the embedded Watson AI libraries, to create an application that would perform sentiment analysis on a provided text. We then deploy the said application over the web using Flask framework.

## Project guidelines

For the completion of this project, you'll have to complete the following 8 tasks, based on the knowledge you have gained through the course.

## Tasks and objectives:

- Task 1: Clone the project repository
- Task 2: Create a sentiment analysis application using Watson NLP library
- Task 3: Format the output of the application
- Task 4: Package the application
- Task 5: Run Unit tests on your application
- Task 6: Deploy as web application using Flask
- Task 7: Incorporate Error handling
- Task 8: Run static code analysis

Let's get started !

# About Embeddable Watson AI libraries

In this project, you'll be using embeddable libraries to create an AI powered Python application.

Embeddable Watson AI libraries include the NLP library, the text-to-speech library and the speech-to-text library. These libraries can be embedded and distributed as part of your application. For your convenience, these libraries have been pre-installed on Skills Network Labs Cloud IDE for use in this project.

The NLP library includes functions for sentiment analysis, emotion detection, text classification, language detection, etc. among others. The speech-to-text library contains functions that perform the transcription service and generates written text from spoken audio. The text-to-speech library generates natural sounding audio from written text. All available functions, in each of these libraries, calls pretrained AI models that are all available on the Cloud IDE servers, available to all users for free.

These libraries may also be accessed through your personal systems. The guidelines for the same are available on the Watson AI library page.

# Task 1: Clone the project repository

The Github repository of the project is available on the URL mentioned below.

```
https://github.com/ibm-developer-skills-network/zzrjt-practice-project-emb-ai.git
```

- Open a new Terminal and make the directory `practice_project` using `mkdir` command and change the current directory `practice_project` using `cd` command

```
mkdir practice_project
cd practice_project
```

- Clone this GitHub repo, using the Cloud IDE terminal to your project to a folder named `practice_project`.

▶ Click here for hint
▼ Click here for solution

git clone https://github.com/ibm-developer-skills-network/zzrjt-practice-project-emb-ai.git practice_project

cd practice_project

- Ensure Python 3.11 and required libraries are available:

```
python3.11 -V
pip3.11 show requests flask pylint
```

- Install missing libraries:

```
python3.11 -m pip install requests flask pylint
```

- Upon completion, the project tab should have the folder structure as shown in the image.

# Task 2: Create a sentiment analysis application using Watson NLP library

NLP sentiment analysis is the practice of using computers to recognize sentiment or emotion expressed in a text. Through NLP, sentiment analysis categorizes words as positive, negative or neutral.

Sentiment analysis is often performed on textual data to help businesses monitor brand and product sentiment in customer feedback, and understanding customer needs. It helps attain the attitude and mood of the wider public which can then help gather insightful information about the context.

For creating the sentiment analysis application, we'll be making use of the Watson Embedded AI Libraries. Since the functions of these libraries are already deployed on the Cloud IDE server, there is no need of importing these libraries to our code. Instead, we need to send a POST request to the relevant model with the required text and the model will send the appropriate response.

A sample code for such an application could be

```
import requests
def <function_name>(<input_args>):
    url = '<relevant_url>'
    headers = {<header_dictionary>}
    myobj = {<input_dictionary_to_the_function>}
    response = requests.post(url, json = myobj, headers=header)
    return response.text
```

*Note: The response of the Watson NLP functions is in the form of an object. For accessing the details of the response, we can use `text` attribute of the object by calling `response.text` and make the function return the response as simple text.*

For this project, you'll be using the BERT based Sentiment Analysis function of the Watson NLP Library. For accessing this function, the URL, the headers and the input json format is as follows.

```
URL: 'https://sn-watson-sentiment-bert.labs.skills.network/v1/watson.runtime.nlp.v1/NlpService/SentimentPredict'
Headers: {"grpc-metadata-mm-model-id": "sentiment_aggregated-bert-workflow_lang_multi_stock"}
Input json: { "raw_document": { "text": text_to_analyse } }
```

Here, `text_to_analyze` is being used as a variable that holds the actual written text which is to be analyzed.

In this task, you need to create a new file named `sentiment_analysis.py` in `practice_project` folder. In this file, write the function for running sentiment analysis using the Watson NLP BERT Sentiment Analysis function, as discussed above. Let us call this function `sentiment_analyzer`. Assume that that text to be analysed is passed to the function as an argument and is stored in the variable `text_to_analyse`.

▼ Click here for the solution
sentiment_analysis.py

```
import requests  # Import the requests library to handle HTTP requests
def sentiment_analyzer(text_to_analyse):  # Define a function named sentiment_analyzer that takes a string input (text_to_analyse)
    url = 'https://sn-watson-sentiment-bert.labs.skills.network/v1/watson.runtime.nlp.v1/NlpService/SentimentPredict'  # URL of the sentiment analysi
    myobj = { "raw_document": { "text": text_to_analyse } }  # Create a dictionary with the text to be analyzed
    header = {"grpc-metadata-mm-model-id": "sentiment_aggregated-bert-workflow_lang_multi_stock"}  # Set the headers required for the API request
    response = requests.post(url, json = myobj, headers=header)  # Send a POST request to the API with the text and headers
    return response.text  # Return the response text from the API
```

This application can now be called using Python shell. To test the application, open a Python shell using python3.11 to open the python shell on the current directory i.e practice_project. *Make sure that the current directory is `practice_project`.*

```
python3.11
```

In the python shell, import the function `sentiment_analyzer`.

▶ Click here for hint
▼ Click here for solution

```
from sentiment_analysis import sentiment_analyzer
```

After successful import, test your application with the text “I love this new technology.”

```
sentiment_analyzer("I love this new technology")
```

The result expected is as shown in the image below. To exit the python shell, press `Ctrl+Z` or type `exit()`.

This completes the Task 2. Note that in the output, the information relevant to us is only the `label` and the `score`. In the following task, you will extract this information from this output.

# Task 3: Format the output of the application

The output of the application created is in the form of a dictionary, but has been formatted as a text. To access relevant pieces of information from this output, we need to first convert this text into a dictionary. Since dictionaries are the default formatting system for JSON files, we make use of the in-built Python library `json`.

Let's see how this works.

First, in a Python shell, import the json library.

```
import json
```

Next, run the sentiment_analyzer function for the text “I love this new technology”, just like in Task 2, and store the output in a variable called `response`.

```
from sentiment_analysis import sentiment_analyzer
response = sentiment_analyzer("I love this new technology")
```

Now, pass the `response` variable as an argument to the json.loads function and save the output in `formatted_response`. Print `formatted_response` to see the difference in the formatting.

```
formatted_response = json.loads(response)
print(formatted_response)
```

The expected output of the above mentioned steps is shown in the image below.

Note that the absence of single quotes on either side of the response indicates that this is no longer a text, but is a dictionary instead. To access the correct information from this dictionary, we need to access the keys appropriately. Since this is a nested dictionary structure, i.e. a dictionary of dictionaries, the following statements need to be used to get the label and the score outputs from this response.

```
label = formatted_response['documentSentiment']['label']
score = formatted_response['documentSentiment']['score']
```

Check the contents of `label` and `score` to verify the output.

Now, for Task 3, incorporate the above mentioned technique and make changes to the `sentiment_analysis.py` file. The expected output from calling the `sentiment_analyzer` function should now be a dictionary with 2 keys, label and score, each having the appropriate value extracted from the response of the Watson NLP function. Verify your changes by testing the modified function in a python shell.

▼ Click here for the solution

```
import requests
import json
def sentiment_analyzer(text_to_analyse):
    # URL of the sentiment analysis service
    url = 'https://sn-watson-sentiment-bert.labs.skills.network/v1/watson.runtime.nlp.v1/NlpService/SentimentPredict'
    # Constructing the request payload in the expected format
    myobj = { "raw_document": { "text": text_to_analyse } }
    # Custom header specifying the model ID for the sentiment analysis service
    header = {"grpc-metadata-mm-model-id": "sentiment_aggregated-bert-workflow_lang_multi_stock"}
    # Sending a POST request to the sentiment analysis API
    response = requests.post(url, json=myobj, headers=header)
    # Parsing the JSON response from the API
    formatted_response = json.loads(response.text)
    # Extracting sentiment label and score from the response
    label = formatted_response['documentSentiment']['label']
    score = formatted_response['documentSentiment']['score']
```

```
    # Returning a dictionary containing sentiment analysis results
    return {'label': label, 'score': score}
```

At completion, the expected output of the function is shown in the image below.

To exit the python shell, type `exit()` or press `Ctrl+Z`.

# Task 4: Package the application

In this task, you have to package the final application you created in tasks 2 and 3

Let's keep the name of the package as `SentimentAnalysis`. The steps involved in packaging are:

1. Create a folder in the working directory, with the name as the package name.

▶ Click here for hint
▼ Click here for solution

```
mkdir SentimentAnalysis
```

2. Move the application code (i.e., the module) into the package folder.

▶ Click here for hint
▼ Click here for solution

```
mv ./sentiment_analysis.py ./SentimentAnalysis
```

3. Create the new file as __init__.py file, inside the package folder to reference the module.

▶ Click here for hint
▼ Click here for solution
Insert this following line to __init__.py

```
from . import sentiment_analysis
```

The final folder structure should look as shown in the image below.

`SentimentAnalysis` is now a valid package and can be imported into any file in this project.

To test this, run a python shell in the terminal and try importing the `sentiment_analyzer` function from the package.

▶ Click here for the hint
▼ Click here for the solution

```
from SentimentAnalysis.sentiment_analysis import sentiment_analyzer
```

No error message received after import statement would indicate that the package is now ready for usage. Test the function by running the following statement in the shell.

```
sentiment_analyzer("This is fun.")
```

The output received would look as shown below.

To exit the python shell, type `exit()` or press `Ctrl+Z`.

# Task 5: Run Unit tests on your application

Since now we have a functional application, it is required that we run unit tests on some test cases to check the validity of its outputs.

For running unit tests, we need to create a new file that calls the required application function from the package and tests its for a known text and output pair.

For this, complete the following steps.

1. Create a new file in `practice_project` folder, called `test_sentiment_analysis.py`.
2. In this file, import the `sentiment_analyzer` function from the `SentimentAnalysis` package. Also import the `unittest` library.

▼ Click here for the solution

```
from SentimentAnalysis.sentiment_analysis import sentiment_analyzer
import unittest
```

3. Create the unit test class. Let's call it TestSentimentAnalyzer. Define `test_sentiment_analyzer` as the function to run the unit tests.

▼ Click here for the solution

```
class TestSentimentAnalyzer(unittest.TestCase):
    def test_sentiment_analyzer(self):
```

4. Define 3 unit tests in the said function and check for the validity of the following statement - label pairs.
   “I love working with Python”: “SENT_POSITIVE”
   “I hate working with Python”: “SENT_NEGATIVE”
   “I am neutral on Python”: “SENT_NEUTRAL”

▶ Click here for hint
▼ Click here for solution

```
class TestSentimentAnalyzer(unittest.TestCase):
    def test_sentiment_analyzer(self):
        # Test case for positive sentiment
        result_1 = sentiment_analyzer('I love working with Python')
        self.assertEqual(result_1['label'], 'SENT_POSITIVE')
        # Test case for negative sentiment
        result_2 = sentiment_analyzer('I hate working with Python')
```

```
        self.assertEqual(result_2['label'], 'SENT_NEGATIVE')
        # Test case for neutral sentiment
        result_3 = sentiment_analyzer('I am neutral on Python')
        self.assertEqual(result_3['label'], 'SENT_NEUTRAL')
```

5. Call the unit tests.

▼ Click here for solution
Add the following line at the end of the file.

```
unittest.main()
```

Now that the file is ready, execute the file to perform unit tests. Upon successful execution, the output of this file should be as shown in the image below.

# Task 6: Deploy as web application using Flask

Now that the application is ready, it is time to deploy it for usage over a web interface. To ease the process of deployment, you have been provided with 3 files which are going to be used for this task.

- Verify directory structure:

```
practice_project/
├── SentimentAnalysis/
│   ├── __init__.py
│   ├── sentiment_analysis.py
├── templates/
│   ├── index.html
├── static/
│   ├── mywebscript.js
├── server.py
```

- This `index.html` in `templates` folder file has the code for the web interface that has been designed for this lab. This is being provided for you in completion and is to be used as is. You are not required to make any changes to this file.

The interface is as shown in the image.

- On clicking the `Run Sentiment Analysis` button, on the html interface, calls this mywebscript.js javascript file in `static` folder which executes a GET request and takes the text provided by the user as input. This text, saved in a variable named `textToAnalyze` is then passed on to the server file to be sent to the application. This file is also being provided to you in completion and is expected to be used as is. You are not required to make any changes to this file.
- Open `server.py` in the `practice_project` folder.This task revolves around the completion of this file. You can complete this file by completing the following 5 steps.

a. *Import the relevant libraries and functions*

In this file, you'll need the Flask library along with its `render_template` function (for deploying the HTML file) and `request` function (to initiate the GET request from the web page).

You also would need to import the `sentiment_analyzer` function from the `SentimentAnalysis` package.

Add the relevant lines of code, importing the said functions, in `server.py`

▼ Click here for solution

```
from flask import Flask, render_template, request
from SentimentAnalysis.sentiment_analysis import sentiment_analyzer
```

b. *Initiate the Flask app by the name* `Sentiment Analyzer`

Put the knowledge gained in Module 2 of this course and add the statement to server.py, that initiates the application and names it `Sentiment Analyzer`.

▼ Click here for solution

```
app = Flask("Sentiment Analyzer")
```

c. *Define the function* `sent_analyzer`

The purpose of this function is two fold. First, the function should send a GET request to the HTML interface to receive the input text. Note that the GET request should reference `textToAnalyze` variable as defined in the `mywebscript.js` file. Store the incoming text to a variable `text_to_analyze`. Now, as the second function, call your `sentiment_analyzer` application with `text_to_analyze` as the argument.

Also, format the returning output of the function in a formal text. For e.g.
`The given text has been identified as POSITIVE with a score of 0.99765.`

▶ Click here for hint
▼ Click here for solution
The function should read like this.

```
@app.route("/sentimentAnalyzer")
def sent_analyzer():
    # Retrieve the text to analyze from the request arguments
    text_to_analyze = request.args.get('textToAnalyze')
    # Pass the text to the sentiment_analyzer function and store the response
    response = sentiment_analyzer(text_to_analyze)
    # Extract the label and score from the response
    label = response['label']
    score = response['score']
    # Return a formatted string with the sentiment label and score
    return "The given text has been identified as {} with a score of {}.".format(label.split('_')[1], score)
```

Note: The function uses the Flask decorator `@app.route("/sentimentAnalyzer")` as referenced in the `mywebscript.js` file.

d. *Render the HTML template using* `render_index_page`

This function should simply run the render_template function on the HTML template, `index.html`.

▼ Click here for solution

```
@app.route("/")
def render_index_page():
    return render_template('index.html')
```

e. *Run the application on* `localhost:5000`

Finally, upon file execution, run the application on host: `0.0.0.0` (or localhost) on port number 5000.

▶ Click here for solution

To deploy the application, execute the file *server.py* from the terminal.

```
python3.11 server.py
```

The output would look like this.

The app is now running on localhost:5000. To access the application, go to the Skills Network Toolbox tab and click on Launch Application. Enter the Application port as 5000, and click on Your Application.

The application interface will open. Use the interface to test your application.

To stop the application, press `Ctrl+C`.

# Task 7: Incorporate Error handling

To incorporate error handling, we need to identify the different forms of error codes that may be received in response to the GET query initiated by the `sent_analyzer` function in `server.py`.

This is already a part of the Watson NLP Library functions and can be observed on the terminal console where the code is running.

Consider the image shown below.

The codes indicate that the initial GET request was successful (200), the request was then successfully transferred to the Watson Library (304) and then the GET request to generate the response was also conducted successfully.

In the case of invalid entries, the system responds with 500 error code, indicating that there is something wrong at the server end.
Invalid entry could be anything that the model is not able to interpret. However, in the situation of this error, this application output doesn't get updated.

Note that the output on the interface is the same as before, the text being analyzed is a random text and the Watson AI libraries are throwing a 500 error confirming that the model has not been able to process the request.

To fix this bug in our application, we need to study the response received from the Watson AI library function, when the server generates 500 error. To test this, we need to retrace the steps taken in Task 2, and test the Watson AI library with an invalid string input.

Open a python shell in the terminal and run the following commands to check the required output after updating sentiment_analysis.py file with below.

```
import requests
# Define the URL for the sentiment analysis API
url = "https://sn-watson-sentiment-bert.labs.skills.network/v1/watson.runtime.nlp.v1/NlpService/SentimentPredict"
# Set the headers with the required model ID for the API
headers = {"grpc-metadata-mm-model-id": "sentiment_aggregated-bert-workflow_lang_multi_stock"}
# Define the first payload with nonsensical text to test the API
myobj = { "raw_document": { "text": "as987da-6s2d aweadsa" } }
# Make a POST request to the API with the first payload and headers
response = requests.post(url, json=myobj, headers=headers)
# Print the status code of the first response
print(response.status_code)
# Define the second payload with a meaningful text to test the API
myobj = { "raw_document": { "text": "Testing this application for error handling" } }
# Make a POST request to the API with the second payload and headers
response = requests.post(url, json=myobj, headers=headers)
# Print the status code of the second response
print(response.status_code)
```

The console response looks as shown in the image below. The red boxes indicate the invalid text and its status code received, and the yellow boxes indicate the valid text and its status code received.

This enables you to modify the application in such a fashion, that we can send different outputs for different status codes.

In the first part of this task, you have to modify the `sentiment_analyzer()` function to return the both `label` and `score` as `None` in case of invalid text entry.

▶ Click here for hint
▶ Click here for the solution

Now, in `server.py`, the response to be sent to the console should also be different for the valid and invalid input types.
For invalid input, let the console print `Invalid input ! Try again.`

▶ Click here for hint
▶ Click here for the solution

Now, your application is capable of responding appropriately to any form of inputs.

# Task 8: Run static code analysis

Finally, in Task 8, we check the quality of your coding skills as per the PEP8 guidelines by running static code analysis.

Normally, this is done at the time of packaging and unit testing the application. However, we have kept this step at the end of this project since the codes are updated in all tasks before this. Once your files for this project are now ready, let us test them for adherence to the PEP8 guidelines.

The first step in this process is to install the `PyLint` library using the terminal.

▶ Click here for the solution

Next, use pylint to run static code analysis on `server.py`.

▶ Click here for the solution

If all aspects of PEP8 guide have been incorporated in your code, then the score generated should be 10/10. In case it isn't, follow the instructions given by pylint library to modify to correct the code appropriately.

This concludes the practice project.

# (Optional) Additional Exercises

Interested learners can the try the following exercises on your own, for enhanced understanding of the concepts learnt through this project. No solution is being provided for these exercises. However, feel free to discuss and share your solutions with your peers in the course discussion forums.

1. Run static code analysis on `sentiment_analysis.py`. Try to achieve a 10/10 score. Hint: *Docstrings*
2. Test the capacity of your application in handling sentences of languages other than English, for e.g. French, German, etc. See if the application responds with an invalid text error.
3. Currently, if the application is run WITHOUT supplying an input, i.e. leaving the text blank, the model still throws the same error of invalid text. Try including a special case, where a blank input receives a different error message.

# Conclusion

Congratulations on completing this project.

With the completion of this project, you have:

1. Created an AI based sentiment analysis application using Watson NLP embedded libraries.
2. Formatted the output received from the Watson NLP library function to extract relevant information from it.
3. Packaged the application and made it importable to any python code for usage.
4. Ran unit tests on the application and checked the validity of its outputs for different inputs.
5. Deployed the application using Flask framework.
6. Incorporated error handling capability in the application, such that a response code of 500 receives an appropriate response from the application.
7. Ran static code analysis on the code files to confirm their adherence to the PEP8 guidelines.