

User Interface and Backend Service

Farzad Shahrivari

February 20, 2024

Question

- Create a Simple User Interface:
 - Develop a basic UI (preferably in React) that allows users to upload a test image from the dataset.
 - Upon submission, send the image to the backend for classification.
- Backend Service:
 - Create a backend service that receives the uploaded image from the UI.
 - Use the trained model to predict the class of the uploaded image.
 - Send the predicted class back to the UI.

Solution

Introduction

In completing this task, I utilized the React library to construct a user interface, enabling end-users to effortlessly upload their desired images. To facilitate the file upload process, I employed Axios. Notably, the React service operates on “http://localhost:5173” and seamlessly transmits the uploaded images to “http://localhost:5000”, where the backend server resides. This streamlined mechanism ensures that once an image is uploaded via the frontend, the application efficiently dispatches the file to the backend for label prediction. Subsequently, the prediction results are seamlessly displayed within the frontend application. For the backend service, Flask was employed. Operating on “http://localhost:5000”, this service receives the uploaded images from the frontend and proceeds to execute the machine learning model for label prediction. Upon completion, the predicted label is promptly relayed back to the frontend service for user display.

Libraries

The frontend user interface service relied on several key libraries to ensure functionality and aesthetics. CSS files were leveraged for styling, while React and Axios played pivotal roles in managing state and facilitating HTTP requests, respectively. Additionally, React hooks were utilized to streamline component logic. Moreover, ReactDOM contributed to rendering React components within the application. Furthermore, the eslintrc library assisted in debugging and maintaining code quality standards.

On the backend, Flask served as the foundation for deploying the application, providing robust routing and request handling capabilities. PyTorch, a powerful machine learning framework, was instrumental in loading the trained model for label prediction. Additionally, torchvision proved invaluable for applying necessary preprocessing steps to input images, ensuring optimal model performance. Lastly, PIL (Python Imaging Library) was utilized to effectively load and manipulate images within the application.

User Interface (React)

The user interface of the application, as implemented in the App.jsx file, is designed to facilitate user interactions and display prediction results seamlessly. State hooks are utilized to manage user inputs and hold data fetched from the backend, as well as the prediction result message. File upload functionality is implemented through state hooks, allowing users to select and upload images seamlessly. The name of the uploaded file is displayed below the upload button, providing users with clear feedback on their selection. Users can easily change the uploaded file by clicking the upload button again, ensuring a smooth user experience. Form submissions and file input changes are handled consistently to maintain a coherent user flow. Upon clicking the predict button, the frontend communicates with the backend to retrieve the prediction label generated by the machine learning model. This label is then displayed to the user, providing them with the classification result for the uploaded image.

The frontend application features a visually appealing design, with a title 'ID Card Classification' presented in an animated manner, fading in and revealing itself. Clear instructions guide users on the upload process, and the upload button is enhanced with a logo resembling a blue cloud, enhancing its visual appeal and intuitiveness. During the initial run of the application, users are greeted with the message 'Upload image to predict: No file uploaded' until they select a file and initiate the prediction process. Upon successful file upload, users receive an alert confirming the upload, and the predicted class for the input image is displayed. This feedback loop ensures users are informed of the progress and outcome of their actions. The frontend application is compatible with popular web browsers such as Edge and Chrome, accessible via the URL 'http://localhost:5173'. Additionally, a React logo is displayed on the new tab opened in Edge, providing users with a consistent and recognizable visual experience. Figure 1 illustrates the user interface.

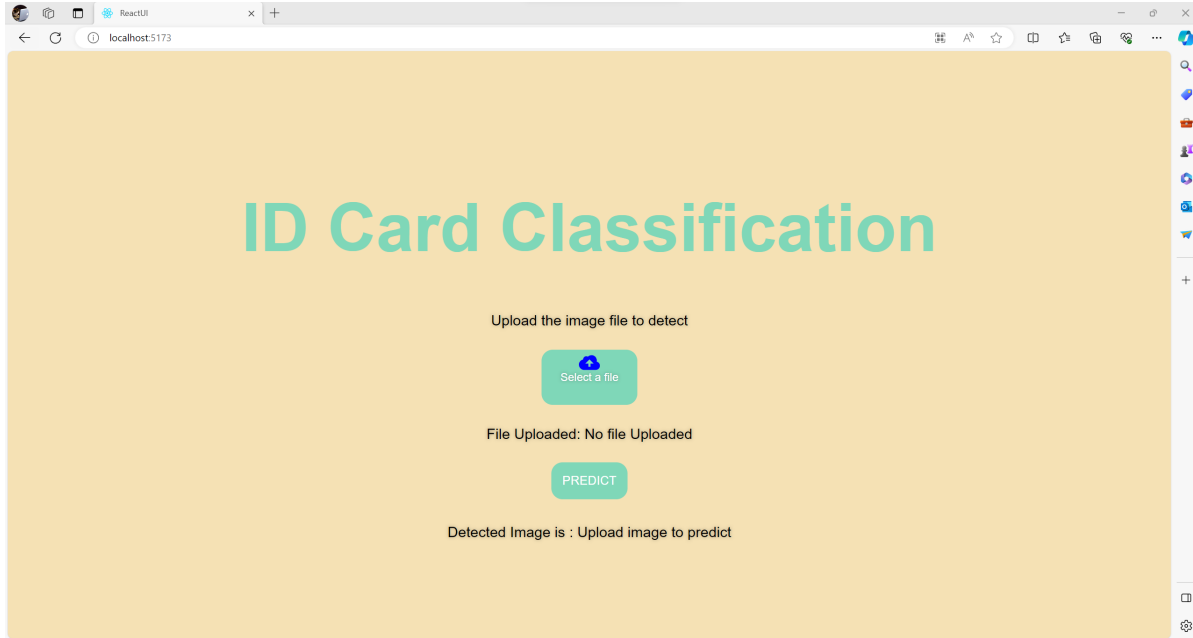


Figure 1: UI Visualization

Backend Service

In the backend service, I first define the myCNN PyTorch class to instantiate a model when loading my trained machine learning model. Using `torch.load`, I load the state dictionary of my machine learning structure and set it to evaluation mode (`model.eval`) to ensure that the model is not affected by dropout layers or batch normalization during inference. This step is crucial for obtaining consistent and reliable predictions. Following model loading, I implement the preprocessing steps required to transform the uploaded image into the desired input tensor format for the loaded machine learning model. These preprocessing steps ensure that the input data conforms to the expected format and facilitates accurate predictions. Subsequently, I set up a Flask application instance and enable Cross-Origin Resource Sharing (CORS) to permit communication with the frontend application running on port 5173. This enables seamless interaction between the frontend React application and the backend Flask server. I define a route handler for the `"/upload"` endpoint, which handles HTTP POST requests. This function receives an uploaded file from the frontend, saves it to the server's file system within a directory named `"uploads"`, and retrieves the file path for further processing. Finally, upon receiving the uploaded image, I make predictions based on the image data using the loaded machine learning model. The predicted labels are then sent back to the frontend for display to the user. Additionally, I implement a check to ensure that duplicate files are not stored on the server by verifying if the uploaded file already exists in the `"uploads"` directory. If a duplicate file is detected, it is removed from the server's file system. The Flask application is configured to run on localhost by default, with port 5000 specified for communication.

How to Run the Application

- Begin by downloading all the contents from the GitHub repository named "Document-Classification-App".
- Next, navigate to the following link to download the "myModel.ckpt" file. Once downloaded, move the file to the "Document-Classification-App/Backend" directory. Ensure that the "myModel.ckpt" file is placed alongside "Backend.py", "Dockerfile", and "requirements.txt" within the same directory. ([Link](#))
- Make sure to have Docker Desktop running on your system (applicable for both Windows and Mac). Open a terminal and navigate to the location where you've stored the "Document-Classification-App" folder using the following command:

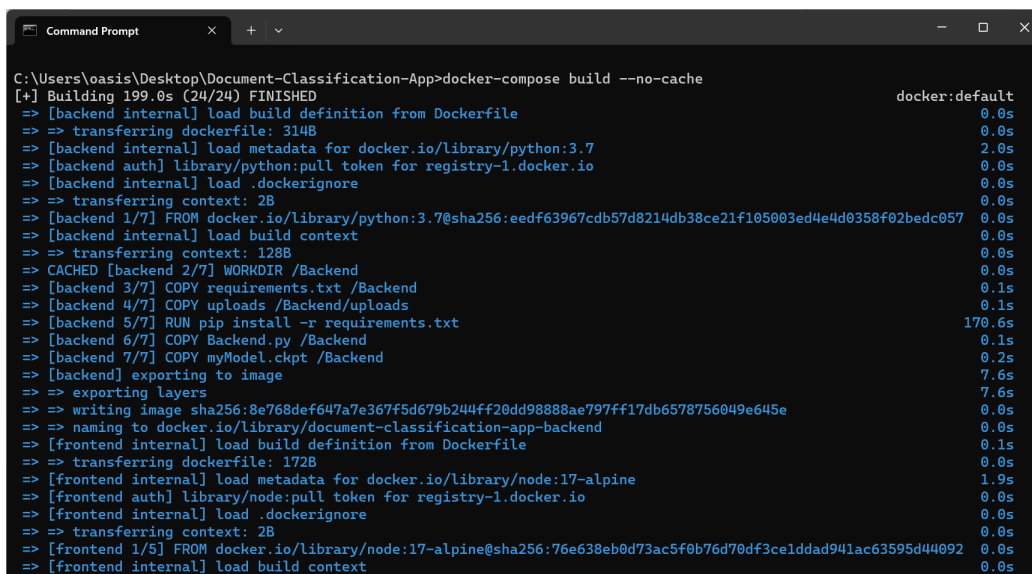
```
cd "path to the folder"/Document-Classification-App
```

Then, execute the command below in the terminal:

```
docker-compose up
```

- Open a web browser and go to "http://localhost:5173". You can now interact with the application by uploading your desired file as many times as you like and then pressing the predict button to see its prediction.

The images below depict the outcome of running the application in the terminal:



```
C:\Users\oasis\Desktop\Document-Classification-App>docker-compose build --no-cache
[+] Building 199.0s (24/24) FINISHED                                docker:default
=> [backend internal] load build definition from Dockerfile        0.0s
=> => transferring dockerfile: 314B                                0.0s
=> [backend internal] load metadata for docker.io/library/python:3.7 2.0s
=> [backend auth] library/python:pull token for registry-1.docker.io 0.0s
=> [backend internal] load .dockerignore                          0.0s
=> => transferring context: 2B                                       0.0s
=> [backend 1/7] FROM docker.io/library/python:3.7@sha256:eedf63967cdb57d8214db38ce21f105003ed4e4d0358f02bedc057 0.0s
=> [backend internal] load build context                          0.0s
=> => transferring context: 128B                                     0.0s
=> CACHED [backend 2/7] WORKDIR /Backend                          0.0s
=> [backend 3/7] COPY requirements.txt /Backend                    0.1s
=> [backend 4/7] COPY uploads /Backend/uploads                    0.1s
=> [backend 5/7] RUN pip install -r requirements.txt               170.6s
=> [backend 6/7] COPY Backend.py /Backend                         0.1s
=> [backend 7/7] COPY myModel.ckpt /Backend                       0.2s
=> [backend] exporting to image                                    7.6s
=> => exporting layers                                              7.6s
=> => writing image sha256:8e768def647a7e367f5d679b244ff20dd98888ae797ff17db6578756049e645e 0.0s
=> => naming to docker.io/library/document-classification-app-backend 0.0s
=> [frontend internal] load build definition from Dockerfile      0.1s
=> => transferring dockerfile: 172B                                  0.0s
=> [frontend internal] load metadata for docker.io/library/node:17-alpine 1.9s
=> [frontend auth] library/node:pull token for registry-1.docker.io 0.0s
=> [frontend internal] load .dockerignore                          0.0s
=> => transferring context: 2B                                       0.0s
=> [frontend 1/5] FROM docker.io/library/node:17-alpine@sha256:76e638eb0d73ac5f0b76d70df3ce1ddad941ac63595d44092 0.0s
=> [frontend internal] load build context                          0.0s
```

Figure 2: My terminal after ruining the docker compose file

Results

To evaluate the functionality and performance of the application, a dataset comprising 27 images was curated. These images were randomly selected from the dataset provided, ensuring a diverse representation of identification cards and passports from various countries.

```
Command Prompt
C:\Users\oasis\Desktop\Document-Classification-App>docker-compose up
[+] Running 3/3
  ✓ Network document-classification-app_default          Created      0.0s
  ✓ Container document-classification-app-backend-1      Created      0.1s
  ✓ Container document-classification-app-frontend-1     Created      0.3s
Attaching to backend-1, frontend-1
frontend-1 | > reactui@0.0.0 start
frontend-1 | > npm run dev
frontend-1 |
frontend-1 | > reactui@0.0.0 dev
frontend-1 | > vite
frontend-1 |
backend-1 | * Serving Flask app 'Backend'
backend-1 | * Debug mode: on
backend-1 | WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI ser
ver instead.
backend-1 | * Running on all addresses (0.0.0.0)
backend-1 | * Running on http://127.0.0.1:5000
backend-1 | * Running on http://172.20.0.2:5000
backend-1 | Press CTRL+C to quit
backend-1 | * Restarting with stat
frontend-1 |
frontend-1 | VITE v5.1.3 ready in 222 ms
frontend-1 |
frontend-1 | → Local:   http://localhost:5173/
frontend-1 | → Network: http://172.20.0.3:5173/
backend-1 | * Debugger is active!
```

Figure 3: My terminal after ruining the docker compose file

```
backend-1 | 172.20.0.1 - - [19/Feb/2024 03:23:17] "GET / HTTP/1.1" 200 -
backend-1 | 172.20.0.1 - - [19/Feb/2024 03:23:17] "GET / HTTP/1.1" 200 -
backend-1 | 172.20.0.1 - - [19/Feb/2024 03:24:04] "POST /upload HTTP/1.1" 200 -
backend-1 | 172.20.0.1 - - [19/Feb/2024 03:24:10] "POST /upload HTTP/1.1" 200 -
backend-1 | 172.20.0.1 - - [19/Feb/2024 03:24:19] "POST /upload HTTP/1.1" 200 -
backend-1 | 172.20.0.1 - - [19/Feb/2024 03:24:33] "POST /upload HTTP/1.1" 200 -
backend-1 | 172.20.0.1 - - [19/Feb/2024 03:24:43] "POST /upload HTTP/1.1" 200 -
backend-1 | 172.20.0.1 - - [19/Feb/2024 03:25:21] "POST /upload HTTP/1.1" 200 -
backend-1 | 172.20.0.1 - - [19/Feb/2024 03:25:40] "POST /upload HTTP/1.1" 200 -
backend-1 | 172.20.0.1 - - [19/Feb/2024 03:25:59] "POST /upload HTTP/1.1" 200 -
backend-1 | 172.20.0.1 - - [19/Feb/2024 03:26:11] "POST /upload HTTP/1.1" 200 -
backend-1 | 172.20.0.1 - - [19/Feb/2024 03:26:18] "POST /upload HTTP/1.1" 200 -
frontend-1 exited with code 0
frontend-1 exited with code 0
backend-1 exited with code 0
```

Figure 4: My terminal after uploading some images and testing the application

For instance, images from the "ID Card of Albania" folder were labeled as "Alb_id#04" through "Alb_id#93", while images from the "ID Card of Finland" folder followed a similar naming convention as "Fin_id#04" through "Fin_id#93". Additionally, images such as "Aze_passport#04", "Esp_id#10", "Est_id#19", "Grc_passport#23", "Lva_passport#32", "Rus_internalpassport#44", "Sub_passport#67", and "Svk_id#79" were selected from their respective folders. Each image was uniquely identified by its corresponding folder and image number, ensuring clarity and organization.

Figures 2, 3, and 4 showcase the results obtained from the application when processing these input images. The application demonstrated consistent and notably high performance in accurately predicting the labels of the input images. These results underscore the effectiveness and reliability of the machine learning model, as well as the efficiency of the frontend React application and the backend Flask service. Together, they form a robust system capable of accurately classifying passport images with high accuracy and reliability.

Final Notes

- Comments have been meticulously provided preceding each section of the code, offering valuable insights into its functionality.



Figure 5: Predictions of randomly selected images from the "ID Card of Albania" folder



Figure 6: Predictions of randomly selected images from the "ID Card of Finland" folder



Figure 7: Predictions of randomly selected images from the dataset

- All services, including the trained machine learning model, the user interface developed in React, and the backend, are engineered for optimal efficiency. Upon running the application, predictions are swiftly generated as users upload their desired images.
- The model exhibits remarkable accuracy, attributable to its high test accuracy rate of 86%. This accuracy is derived from splitting the dataset into 70%, 10%, and 20% for training, validation, and testing, respectively. When considering nearly perfect accuracy for both training and validation data, the overall model accuracy on the dataset is notably high. However, there remain some images that are misclassified, as indicated in the results section. To assess the true accuracy of the model (86%), it is recommended to utilize unseen images from the dataset. In such cases, the model is expected to maintain approximately 86% accuracy, meaning one to two images out of ten may be misclassified.