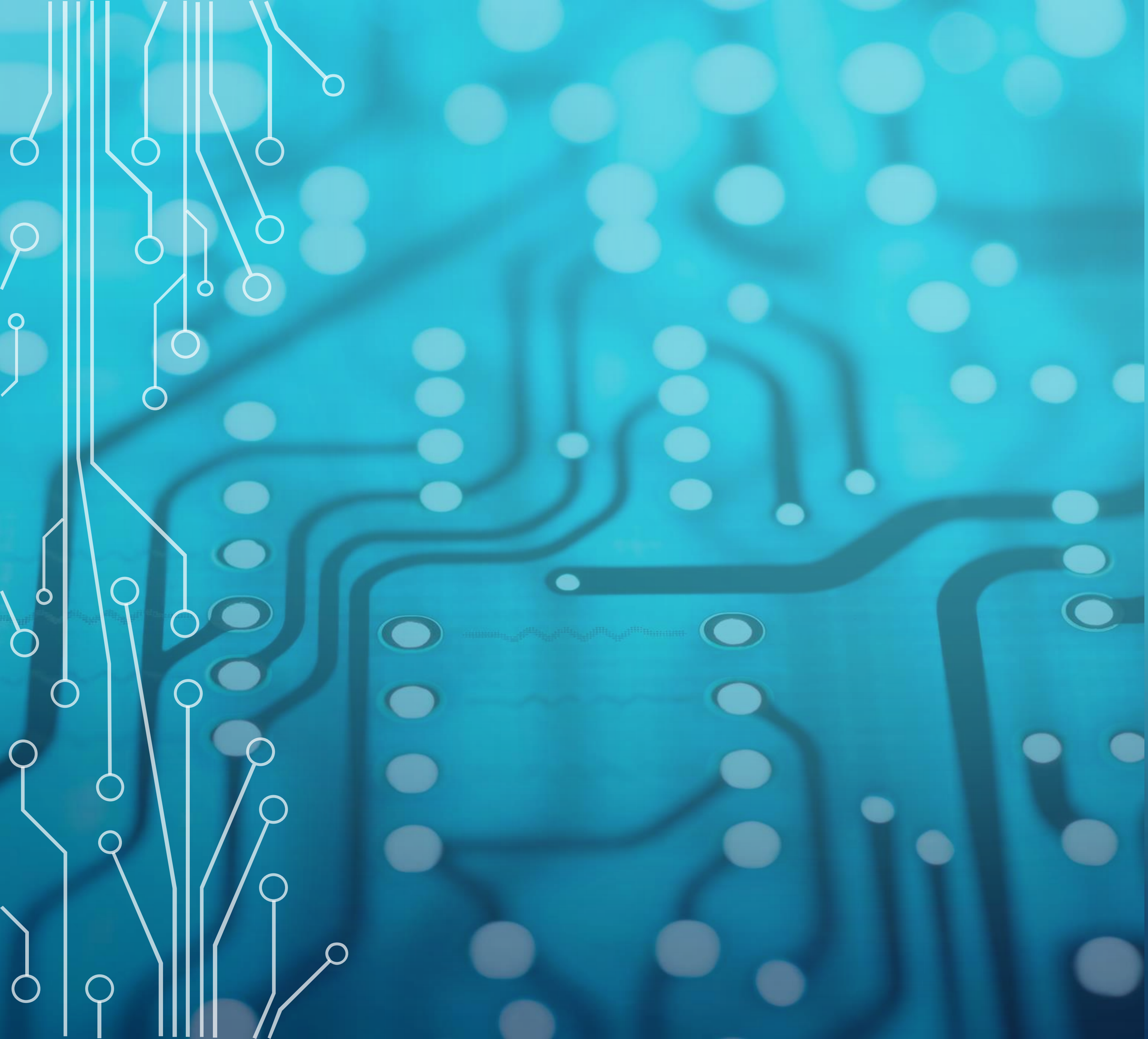
MODEL TO PRODUCTION

IMAGE CLASSIFICATION FOR A REFUND DEPARTMENT

ERIK NEL

HTTPS://GITHUB.COM/ERIK-02/ML-IMAGE-CLASSIFICATION-FLASK-API



PRESENTATION LAYOUT

- System design layout explanation
- Challenges of implementing a predictive model as a service.
- Constraints of implementing a predictive model as a service.
- Requirements for data acquisition, storage, and processing.
- Monitoring components for the reliable execution of the predictive model.

SYSTEM DESIGN

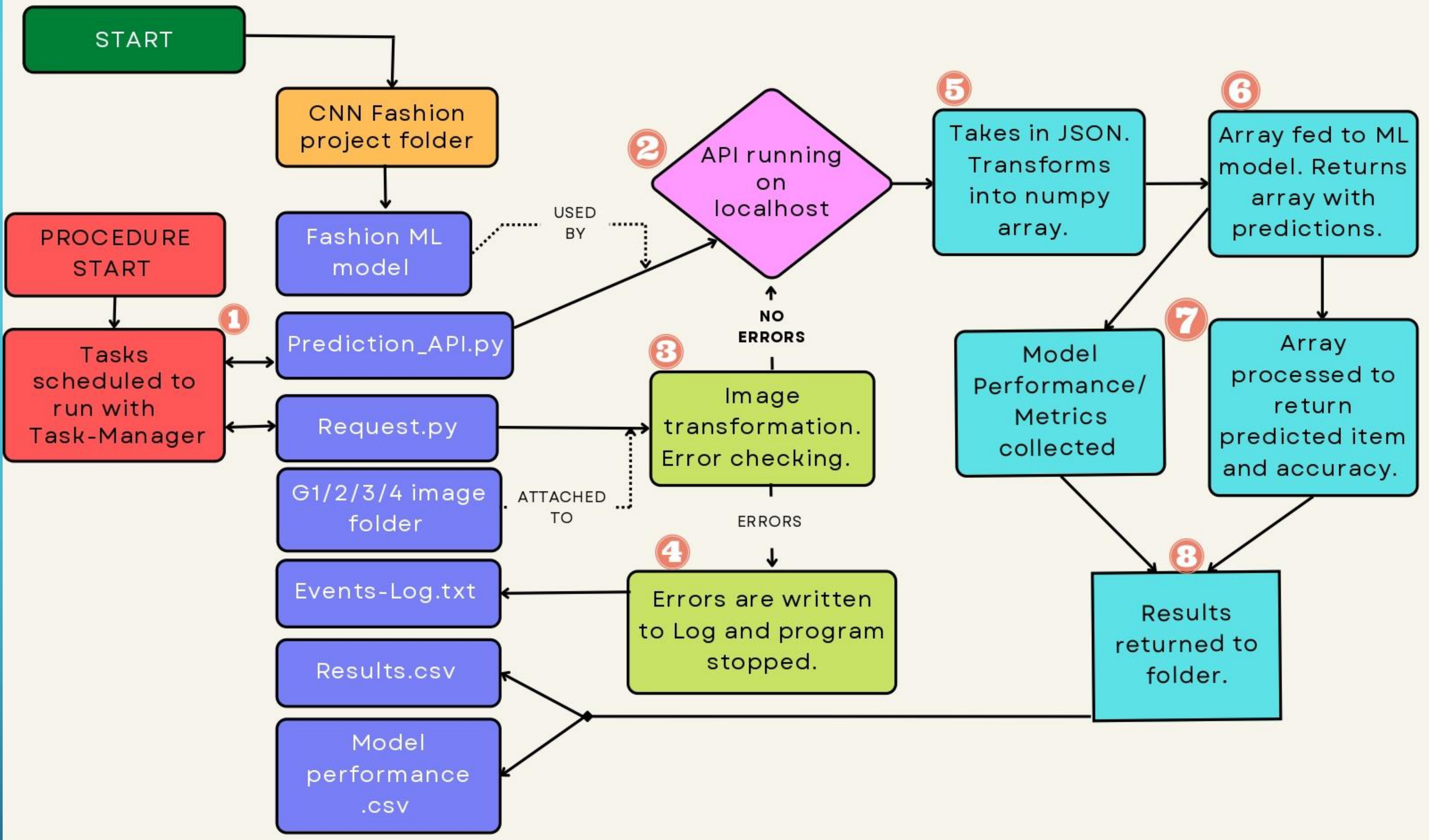

START
CNN Fashion project folder
PROCEDURE START
Fashion ML model
USED BY
1
Tasks scheduled to run with Task-Manager
Prediction_API.py
Request.py
G1/2/3/4 image folder
ATTACHED TO
Events-Log.txt
Results.csv
Model performance .csv
2
API running on localhost
NO ERRORS
3
Image transformation. Error checking.
ERRORS
4
Errors are written to Log and program stopped.
5
Takes in JSON. Transforms into numpy array.
6
Array fed to ML model. Returns array with predictions.
Model Performance/ Metrics collected
7
Array processed to return predicted item and accuracy.
8
Results returned to folder.

Image Transformation.

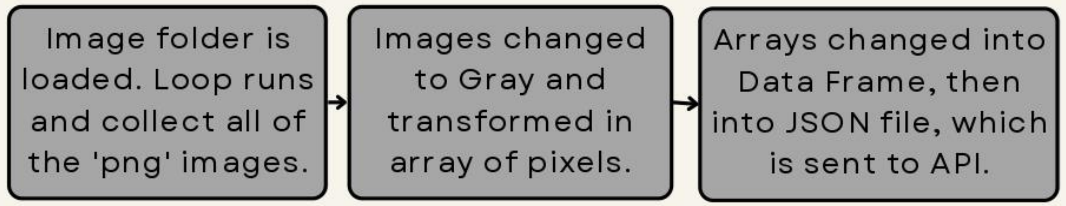
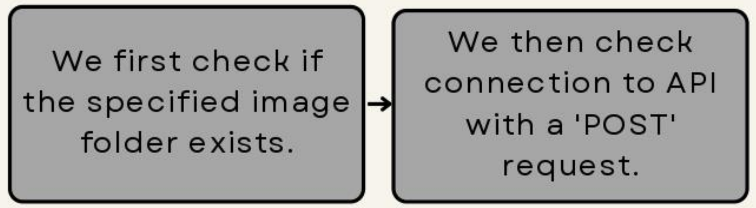


Image folder is loaded. Loop runs and collect all of the 'png' images. → Images changed to Gray and transformed in array of pixels. → Arrays changed into Data Frame, then into JSON file, which is sent to API.

Error Checking.



We first check if the specified image folder exists. → We then check connection to API with a 'POST' request.

TASKS SCHEDULED TO RUN AND MAKE API AVAILABLE ON LOCALHOST.

- The first task that is scheduled is to run the script that creates our API on the localhost, to which predictions will be sent.
- After 5 minutes, the next task is scheduled, which will send the request to the API.
- After 30 minutes, the API task will be automatically closed and rerun the next day at 6 pm.
- In order to create our API, the script makes use of another folder called ‘fashion’ which contains our fashion ML model.
- This is how the API can make predictions by using our previously created ML model.
- The folder ‘ML model setup’ contains code that created our CNN model.

IMAGE TRANSFORMATION AND ERROR CHECKING

- When we run our ‘request’ script, it uses the folder that contains the images to be predicted. In our case we use ‘G1’, containing 100 images.
- The images are then opened one by one and transformed into a NumPy array, which is then sent to the API.
- If there were to be any errors, such as the image folder not existing or the API not set up, our program will catch this.
- When an error is caught, instead of crashing the program it is written to the ‘Events-log.txt’ file.

API COLLECTS ARRAY SENT FROM REQUEST AND RETURNS PREDICTION

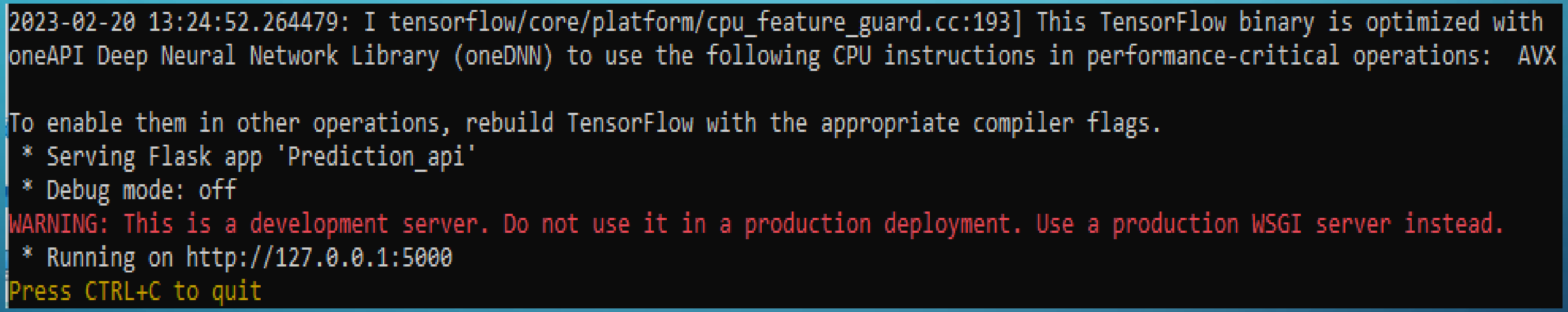
- The API will be looking for any data that it is able to collect. In this case it collects a JSON file that contains our array of image pixel data.
- JSON transformed into an understandable array as it was on client's request side.
- These arrays are then fed into our ML model after it has undergone the necessary transformations.
- The ML model then returns the class prediction, ranging from 1-10 as well as the prediction certainty in the form of a percentage value.

COLLECTING MODEL METRICS AND RETURNING RESULTS

- After we have run our ML model on the image data, we also collect certain metrics to help us keep track of our model’s performance.
- These metrics include Mean Accuracy, Process Time and Time per Prediction.
- All of these prediction results are then returned to the user. We can see that we have a file named ‘Results.csv’. This file contains our prediction results along with the accuracy.
- Our other metrics, mentioned on the left, are then stored and returned to the API/ ML model administrator.

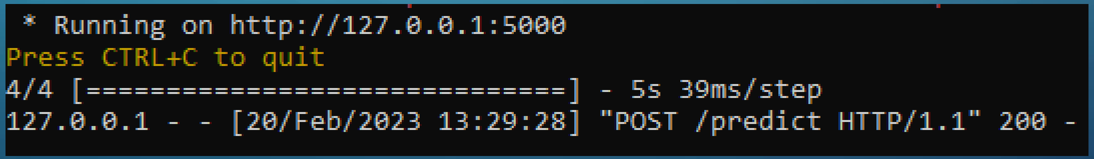
API INTERFACE LOOK.

- This is what the interface of our API looks like when it has been started up and when a request has been sent and processed successfully.



```
2023-02-20 13:24:52.264479: I tensorflow/core/platform/cpu_feature_guard.cc:193] This TensorFlow binary is optimized with
oneAPI Deep Neural Network Library (oneDNN) to use the following CPU instructions in performance-critical operations:  AVX

To enable them in other operations, rebuild TensorFlow with the appropriate compiler flags.
 * Serving Flask app 'Prediction_api'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
```



```
 * Running on http://127.0.0.1:5000
Press CTRL+C to quit
4/4 [==============================] - 5s 39ms/step
127.0.0.1 - - [20/Feb/2023 13:29:28] "POST /predict HTTP/1.1" 200 -
```

THE PROJECT

- This project is a simple image classifier for an online shopping platform that sells sustainably made clothes.
- The goal of the project is to develop a Machine Learning model that can take in image data, and return the predicted clothing item as well as the prediction accuracy.
- The challenge is to then implement the model as a service that can be accessed through an API.
- The image data was collected from Kaggle and is based on the MNIST Fashion Dataset.

CHALLENGES OF INTEGRATING A PREDICTIVE MODEL AS A SERVICE .

- 1. API dependencies. The dependencies are everything that our API application needs to be able to perform correctly. These include all our imported python modules, as well as our ML model. For the API to be deployed, whether on a business server or a localhost, all of these dependencies must be included for our application to run correctly. For simplicity, the required python modules can be installed using the 'Requirements.txt' file.
- My ML model is a CNN image classifier. It takes in pixel values and returns the predicted clothing item, along with the certainty of the prediction. The file 'Model code setup' contains all the training and configuration of the model.
- For the API to be able to perform the request being sent and return predictions, the API must have the correct ML model. I used the Keras module to import my ML model, which is the folder 'fashion model'. Without this ml model loaded into the API, it will be unable to give predictions about the images that was sent.

CHALLENGES OF INTEGRATING A PREDICTIVE MODEL AS A SERVICE .

- 2. File size being sent. Flask has a 16-megabyte payload limit. Meaning the 'payload' or images in our case cannot be larger than 16 megabytes. The images that I used is around 6.42 kb in size. This means that we are limited and can only send around 2500 images at a time. Initially this may sound like a lot, but since we are doing batch processing, we may be required to send more at a time.

- My solution is to transform the images into their pixel values on the client side, using the 'cv2' library. This allows us to do some pre-processing as well, since our ML model can only take an array of pixel values as input. These images are essentially transformed into a pandas data frame. This data frame of pixel values, is less than half of the original image's size.

- For example, our first folder of images, contains 101 images, where the total size in ‘png’ format is 619kb. When transformed into a pandas data frame, it is now only 304kb. Meaning that we can send much more data at a time.

CHALLENGES OF INTEGRATING A PREDICTIVE MODEL AS A SERVICE .

- 3. API server not working/ request being sent unable to connect to API service. In order to make requests to the API and return our predictions, we need to be sure that our API is up and running. Since I used windows task scheduler, it is more difficult to know whether the API is up and running on my localhost. When our API is hosted on a service it might be easier to know that our API is in fact up and running.
- The solution to this was to create a check when trying to send our data to the API. This check tries to connect to the API service 3 times. each try is spaced out in 1-minute intervals. Each time that the check fails a detailed error message is written to our 'Events-Log.txt' file.
- There is also no need to shut down the server when it is up and running. This will ensure that our prediction service is always up and running, resulting in faster computation times.

CONSTRAINTS OF IMPLEMENTING A PREDICTIVE MODEL AS A SERVICE.

- 1. Computing power. In the world of ML and specifically computer vision, these algorithms tend to be quite expensive on computing power. If the images requires a lot of pre-processing to be done, this can also add to the computations. I used a simple CNN model without making a lot of changes to the inner workings.
- The prediction accuracy of the model is at 92% and although it can be made higher, doing so will result in the need of much more computation power. In order to reduce some of the computational costs, the image processing is done on the client-side, while only the predictions are made server-side.
- The more data that is sent to the API, the more time our model will need to predict the results. At the end of each request, certain metrics must also be captured and saved to a specific file. Although it may not be much, there is still relevant computational costs needed to keep in mind.

CONSTRAINTS OF IMPLEMENTING A PREDICTIVE MODEL AS A SERVICE.

- 2. File directories. As mentioned earlier, when we are reading in the ML model, it is necessary to specify the location where the model is stored. If we were to change our model's name, storage location or directory, we will need to edit our program and change this path.
- An easy solution is to keep our API program in the same folder or subdirectory as where the model is stored. That way we won’t have to change the location in which our model is stored.

REQUIREMENTS FOR DATA ACQUISITION, STORAGE, AND PROCESSING.

1. Acquisition: To train the ML model, I acquired the necessary data from Kaggle. I used the popular MNIST Fashion dataset. The data can also be found in the link. The data consists of 2 csv files, one for training and one for validation. The images used to test the API, can also be found in the link. The images that are used in the 'Request.py' file are stored in folders.

2. Storage: For my example, it is necessary to have the local path of where the images are stored on the computer, in order for our program to find it. It is advisable to keep the 'Request.py' file in the same folder as our image folders being sent to the API. It is of course also possible to use an online storage platform such as Google cloud or One-Drive.

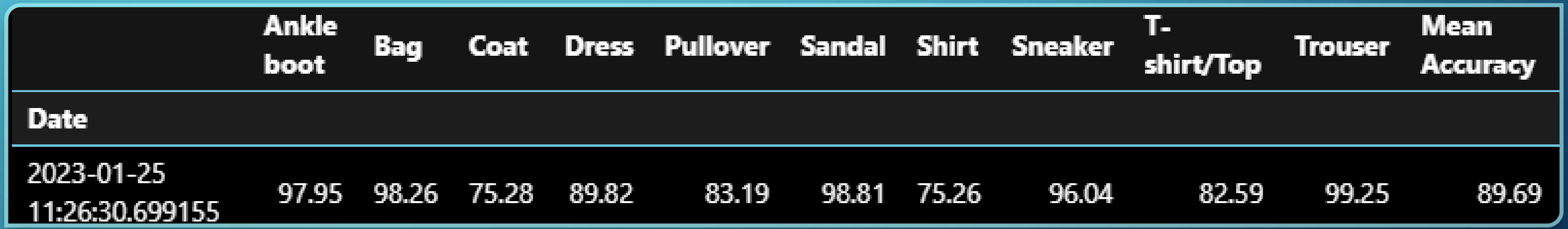
3. Processing: The code opens the folders and runs through each image, first converting it to a 'Gray-scale' colour and then converting it to pixel data as it is read. The images are then resized to a simple 28x28 size. This means that each image consists of 784 pixels. Each individual image's pixels are appended to a NumPy array and later transformed into a ‘pandas’ data frame. It is this data frame that is sent to our API where the ML model can make predictions.

MONITORING COMPONENTS FOR THE RELIABLE EXECUTION OF THE PREDICTIVE MODEL.

MONITORING COMPONENTS FOR MY ML MODEL CAN BE SPLIT INTO 2 GROUPS, MODEL PERFORMANCE AND MODEL METRICS.

- 1. Model performance: Our model performance metrics aim to keep track of how well our model is performing in production. Ideally, the aim is to see how accurately our model is predicting the outcome. Although this can be quite hard to monitor in our case. Since the model is making predictions on the clothing items of unseen images, we would need a way to confirm or deny whether the prediction made was right or wrong. Therefore, we would need a human intervention. One which we do not have.

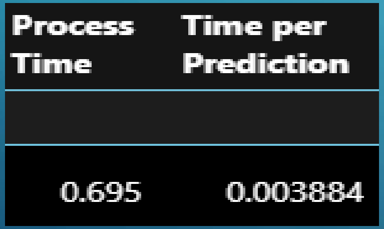
- Rather than obtaining the accuracy of the predictions we collect other useful metrics. First, we collect the mean, or average predicted accuracy over all the classes, which is our different clothing items. We then also collect the mean predicted accuracy of each individual class as well.



	Ankle boot	Bag	Coat	Dress	Pullover	Sandal	Shirt	Sneaker	T-shirt/Top	Trouser	Mean Accuracy
Date											
2023-01-25 11:26:30.699155	97.95	98.26	75.28	89.82	83.19	98.81	75.26	96.04	82.59	99.25	89.69

- We can then set a threshold for which we want our predicted accuracy to be, based on the predicted results obtained in training. Over the long term we can monitor these metrics and retrain our model when the prediction accuracy falls below that threshold.

MONITORING COMPONENTS FOR THE RELIABLE EXECUTION OF THE PREDICTIVE MODEL.



Process Time	Time per Prediction
0.695	0.003884

- 2. Model metrics: The model metrics focusses on the so called 'Health' of the model. We always aim to achieve the best performance, which includes the time needed to run the full request sent to the API, as well as the time per prediction.
- It is useful to have the time per prediction because it can show us whether our model is too complex or by how much we can improve it. The more complex our model is, the more computational costs are involved.
- We can also look at how much time our entire process is taking and compare that to the time used to make predictions. This will show us whether our pre- and postprocessing is using a lot of unnecessary time or not.

THE END!