1.0.0.0. **INTRODUCTION**

1.1.0.0. Setting the context

1.1.1.0. This report is to be read in conjunction with the Google Colab notebook for deployment & productionalization of the Loan Defaulting Tendency Predictor app, trained on the Home Credit loan defaulter prediction problem hosted on Kaggle here.

1.1.2.0. An understanding of the complete problem context and high level summary of the datasets used can be sought from here.

1.1.3.0. The datasets used in this phase are the ones already processed based on the insights and feature engineering in earlier EDA phase, which can be referred to, here.

1.1.4.0. The model referred to as the best model and the dataset with selected features is most suitable towards maximising the model performance; facts evidenced in the previous phase which can be read here.

1.1.5.0. The deployed app can be viewed from here.

1.2.0.0. A quick refresher about Home Credit's motivation for predicting potential defaulters

1.2.1.0. Though there are a lot of people seeking loans from banks and lending institutions, only a few of them get approved. This is primarily because of insufficient or non-existent credit histories of the applicant. Such a population is taken advantage of by untrustworthy lenders. In order to make sure that these applicants have a positive loan taking experience, Home Credit uses Data Analytics to predict the applicants' loan repayment abilities, trying to ensure that the clients capable of loan repayment do not have their applications rejected.

1.3.0.0. Summary of deployment process

1.3.1.0. The LightGBM model is the best model and PyCaret library was used for the whole process from data ingestion to the prediction stage on Colab notebook.

1.3.2.0. For the deployment phase, the entire data pipeline and model were recreated using sklearn.

1.3.3.0. All the necessary codes, input data and auxiliary files were pushed to a repository on GitHub.

1.3.4.0. Streamlit is used owing to its simplicity and intuitive UI.

1.3.5.0. The app is hosted on Heroku free tier service and deployed.

2.0.0.0. **DEPLOYMENT PROCESS IN DEPTH**

2.1.0.0. The PyCaret pipeline & its quirks

2.1.1.0. The best model & pipeline developed in the previous phase was used for deployment using FastAPI on Heroku.

2.1.2.0. The PyCaret model is substantially large (approximately 180mb). Though I could load the pickled model in Colab & get the predictions on defaulting tendency for the entire test dataset, deployment on Heroku threw an error. The error also persisted upon using Streamlit.

2.1.3.0. A possible cause might be the PyCaret pipeline or the usage of GitLFS while pushing the large files to Git repo.

2.1.4.0. To debug considering GitLFS as a compatibility issue, to reduce the file sizes, the model was trained using 50% & 25% of the original training data. However, there was no improvement.

2.1.5.0. Thus, presuming that PyCaret has compatibility issues in deployment, considering the time constraint, I decided to recreate the entire model and data pipeline in sklearn.

2.2.0.0. Recreating the model & data pipeline in sklearn

2.2.1.0. The ipython notebook has the entire modeling & data pipeline created in the sklearn library. The pickled model and pipelines were relatively very light-sized which eliminated the need to use LFS to push these files to the repo.

2.2.2.0. A thing worth noting is the amount of coding involved in using sklearn compared to PyCaret.

2.2.3.0. Importantly, the deployment on Heroku was successful emphasising that PyCaret or the LFS had compatibility issues.

2.3.0.0. The detailed process of deployment

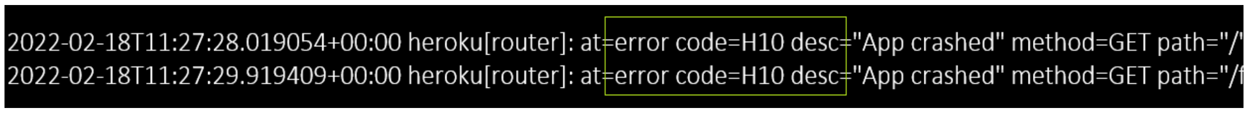
2.3.1.0. Before listing the steps involved in this iterative process, it is important to list the various deployment strategies executed as this has bearing on the whole cycle.

2.3.2.0. The following deployment options were carried out -
2.3.2.1. FastAPI + Heroku
2.3.2.2. FastAPI + AWS
2.3.2.3. Streamlit + Heroku
2.3.2.4. Streamlit + Azure
2.3.2.5. Streamlit + AWS

2.3.3.0. Initially, the model in the PyCaret framework was deployed using FastAPI on Heroku. For this, all the code files and data sets along with the auxiliary files need to be pushed to the GitHub repository.

2.3.4.0. Owing to the large file size of the datasets as well as the PyCaret's model and pipeline size, pushing to Git through CLI or desktop was not possible and Git LFS [Large File Storage] was used to push these large files to the repo. Size restrictions for GitHub file upload & CLI/Desktop are 25mb & 100mb respectively whereas the files were around 300mb.

2.3.5.0. Usage of the LFS and PyCaret apparently has compatibility issues with either Streamlit or Heroku itself as the deployed app crashed with H10 error.



```
2022-02-18T11:27:28.019054+00:00 heroku[router]: at=error code=H10 desc="App crashed" method=GET path="/
2022-02-18T11:27:29.919409+00:00 heroku[router]: at=error code=H10 desc="App crashed" method=GET path="/f
```

[A screengrab of the copied log for H10 error highlighted]

2.3.6.0. Searching on the internet for the error in our environment context highlighted compatibility errors between LFS and possibly PyCaret.

2.3.7.0. Owing to the time constraint, a detailed debugging was skipped in favour of trying out options to reduce dependency on LFS & PyCaret.

2.3.8.0. Thus, in order to create a light-weight model as well as the total pipeline, sklearn was used to build the whole framework from scratch.

2.3.9.0. The resulting files [pickled files] were very lightweight which eliminated the need for LFS.

2.3.10.0. By implementing the dataflow pipeline in sklearn, PyCaret library was dropped too.

2.3.11.0. The whole pipeline and model creation using sklearn is coded on this Colab notebook and the outputs are pickled for usage.

2.4.0.0. <u>The actual deployment cycles</u>

2.4.1.0. Having fixed the data pipeline and the model, the first deployment combination was FastAPI + Heroku.

2.4.1.1. FastAPI uses uvicorn owing to its ASGI implementation making it pretty fast.

2.4.1.2. This was very easy to set up. The UI is by OpenAPI (previously Swagger) which is pretty barebones.

2.4.1.3. Using Jinja, making a more customised UI may have been possible but time constraint compelled me to try out other options.

2.4.1.4. FastAPI is very easy to work with and for my case, the documentation needed is fairly elaborate. I deployed a few toy example models and they were extremely fast and behaving as intended. However, the UI (OpenAPI/Swagger) was too plain for me and I could not use Jinja effectively to customise the UI which made me try out Streamlit.

2.4.2.0. While using Streamlit, I tried out Azure, originally with the PyCaret pipeline. Setting up Azure though fairly easy, takes a lot of time for setting up the box as well as after connecting to Git repo.

2.4.2.1. The PyCaret model on Azure was not successfully deployed throwing up memory exceed error. The overall time to rebuild made me check the AWS platform.

2.4.3.0. FastAPI + AWS / Streamlit + AWS was tried out initially with the PyCaret model and deployment failed due to the RAM consumption. The sklearn model was deployed using Streamlit on AWS and it was successfully deployed.

2.4.3.1. Biggest advantage of AWS is the fact that large files can be easily pushed to the remote box using FTP programs like WinSCP.

2.4.3.2. AWS has elaborate documentation and online resources which make setting up ec2 instances extremely easy for people with coding experience.

2.4.3.3. Deployment on AWS though well documented, especially on external platforms, is relatively involved. I needed two additional softwares - Putty/PuttyGen & WinSCP for SSHing into and doing file transfers with the box respectively.

2.4.3.4. Code's RAM usage needs to be quite optimised as a deployment that worked on Heroku failed on AWS due to RAM consumption. Thus, I kept the AWS implementation on backburners and considered Heroku as primary.

2.4.4.0. A summary of the pros & cons of each option I used while deploying my model is given below -

Note - This table is compiled based on my experience while deploying the PyCaret &/or sklearn model and my experience of coding.

Sl. no.	**Platform/ Software**	**Pros**	**Cons**
1	FastAPI	✅ Easy to set up and get running ✅ Actually fast [ASGI implementation]	❌ Limited examples especially for Frontend/ UI customization
2	Streamlit	✅ Easy to set up and get running ✅ Good community support for quick front-end customization	None

3	AWS	✅ Elaborate documentation and resources/examples ✅ FTP to transfer large files avoids LFS dependence	❌ Free-tier box provides relatively limited compute resources, need very optimised code for using box fully ❌ Non-coding beginners can be intimidated by steps in getting the box running ❌ Additional software for SSH & FTP required for deployment ❌ Process of keys/ private keys and SSH may be little intimidating
4	Azure	✅ Beginner-friendly, especially windows users ✅ Elaborate session & error logs ✅ No additional software needed to deploy app	❌ Fairly time-consuming process for initial set up as well as during deployment ❌ Lot of tools like PowerBI integrated in dashboard, usage of which has a steep learning curve for beginners
5	Heroku	✅ Beginner-friendly & easy to set up and get running, no server configuration ✅ No additional software needed to deploy app ✅ Good logs and version control	❌ Procfile content need fair knowledge of CS ❌ Dependence on GitHub [& LFS for large files] sometimes throws errors

2.5.0.0. <u>The finalised deployment platform</u>

2.5.1.0. After trying out the mentioned combinations of platforms and services, I opted for Streamlit+Heroku as the primary method of deployment for the following reasons -

2.5.1.1. Streamlit allowed me to customise the app UI much better than FastAPI to the extent I could.

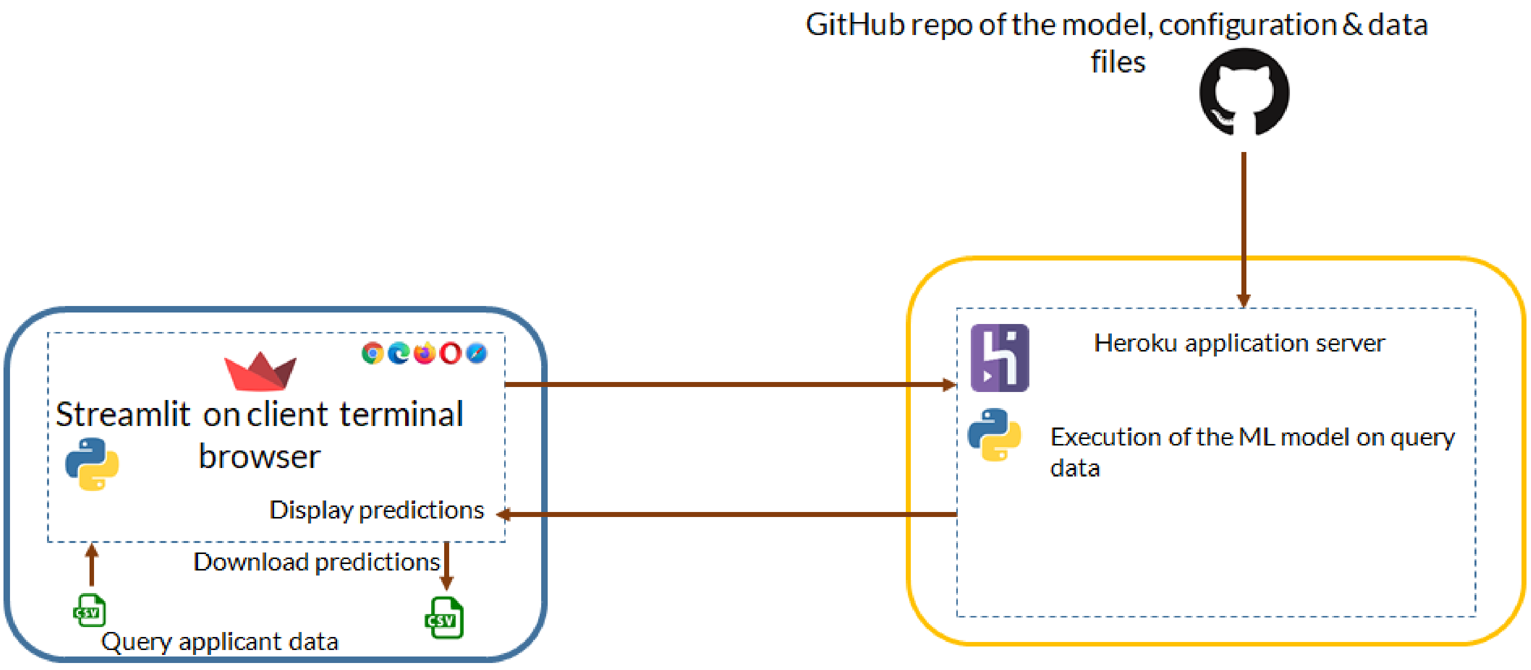
2.5.1.2. Heroku required the least amount of time for iterative deployment and the entire repository being on GitHub, I could modify, build & redeploy from anywhere.

2.6.0.0. <u>Basic architecture of the app system</u>

2.6.1.0. The user interacts with the app via any browser on their local PC running the Streamlit client by uploading the query csv file containing the applicant data.

2.6.2.0. The model hosted on the remote Heroku box computes the predictions and sends back the results which are displayed on the user's browser as well as can be downloaded.

2.6.3.0. Upon linking the GitHub repo with the Heroku site for the first time, the files are pulled into the Heroku box.


GitHub repo of the model, configuration & data files
Heroku application server
Execution of the ML model on query data
Streamlit on client terminal browser
Display predictions
Download predictions
Query applicant data

Simplified view of the model system architecture [img source - self]

2.7.0.0. <u>Highlights of the deployed app</u>
2.7.1.0. App engagement
2.7.1.1. The app accepts the Home credit applicant details in a CSV file as is in the test dataset.
2.7.1.2. A downloadable template is provided for the user to enter data into.
2.7.1.3. Individual form fields are not provided owing to the large number of fields which will result in an unpleasant UX.
2.7.1.4. The output of the model predictions is displayed on the screen as an interactable dataframe as well as a downloadable CSV file appended to the original query set.
2.7.1.5. Importantly, following error handling methods are implemented -
2.7.1.6. The uploaded csv is checked for correctness w.r.t. the actual feature names required in template and in case of mismatch, displays a message stating the same.
2.7.1.7. When a new, unseen categorical variable is encountered in the query data, handling is done by ignoring it which is implemented by setting ***'handle_unknown'*** parameter to ***'ignore'***. This ignores the unseen category values and proceeds ahead.

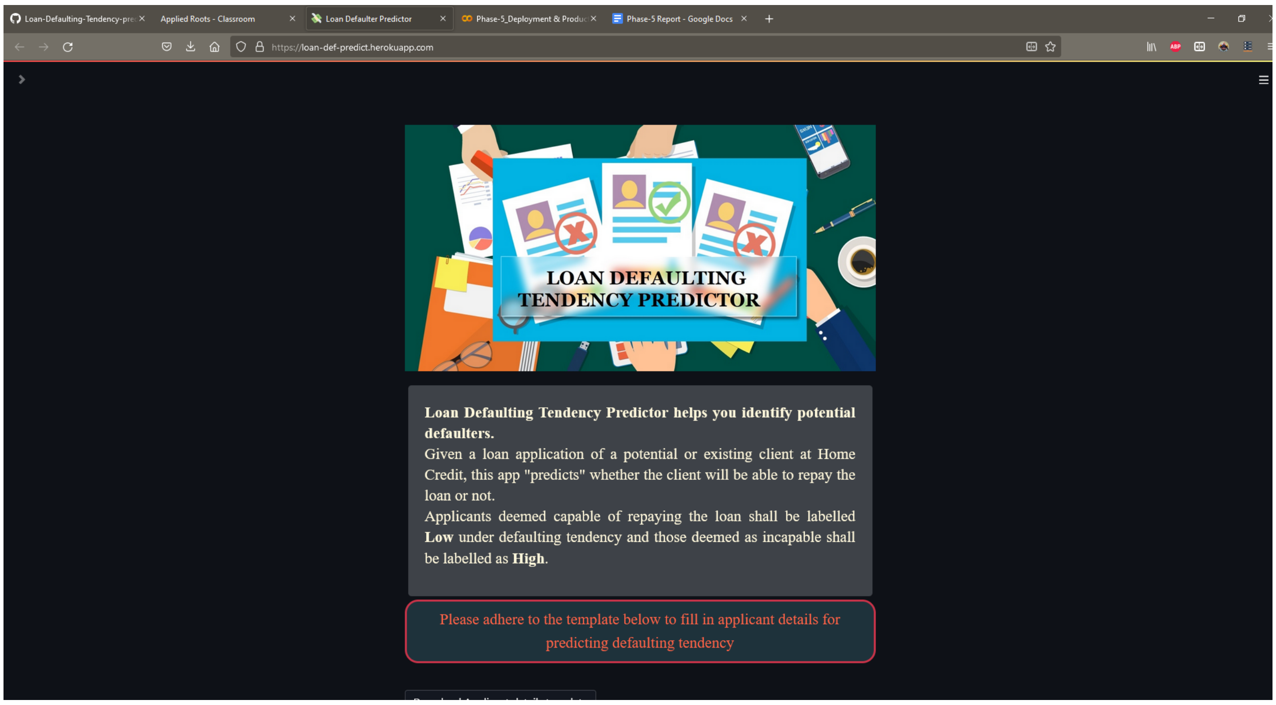
2.7.2.0. Scalability, Throughput, Latency and real-world case
2.7.2.1. The app was fed the Home Credit raw test dataset consisting of around 50k applicant records with a file size of approximately 26mb.
2.7.2.2. The app, after upload [depending on internet connectivity took around 5 -30 seconds] does the entire data processing and predicts the defaulting tendency for the applicants in less than 40 seconds.
2.7.2.3. Considering the real-world scenario, the latency is not a strict requirement and is acceptable.
2.7.2.4. With context to throughput, as the app can be run frequently per day or even per application, the throughput volumes are not a limiting case.

2.7.3.0. Visible limitations and scope for improvement/innovation

2.7.3.1. Going through the logic of current execution, the query data is converted to a Pandas dataframe and compared with existing Bureau & Previous Home Credit data (also a Pandas dataframe). This implementation might be checked with SQL db for optimised and scalable performance which ‘might’ improve the system latency or reduce memory requirement.

2.7.3.2. Going ahead, the debugging of the issue with PyCaret may also be done so that the low code tool can also be used.

2.7.4.0. **App interface**

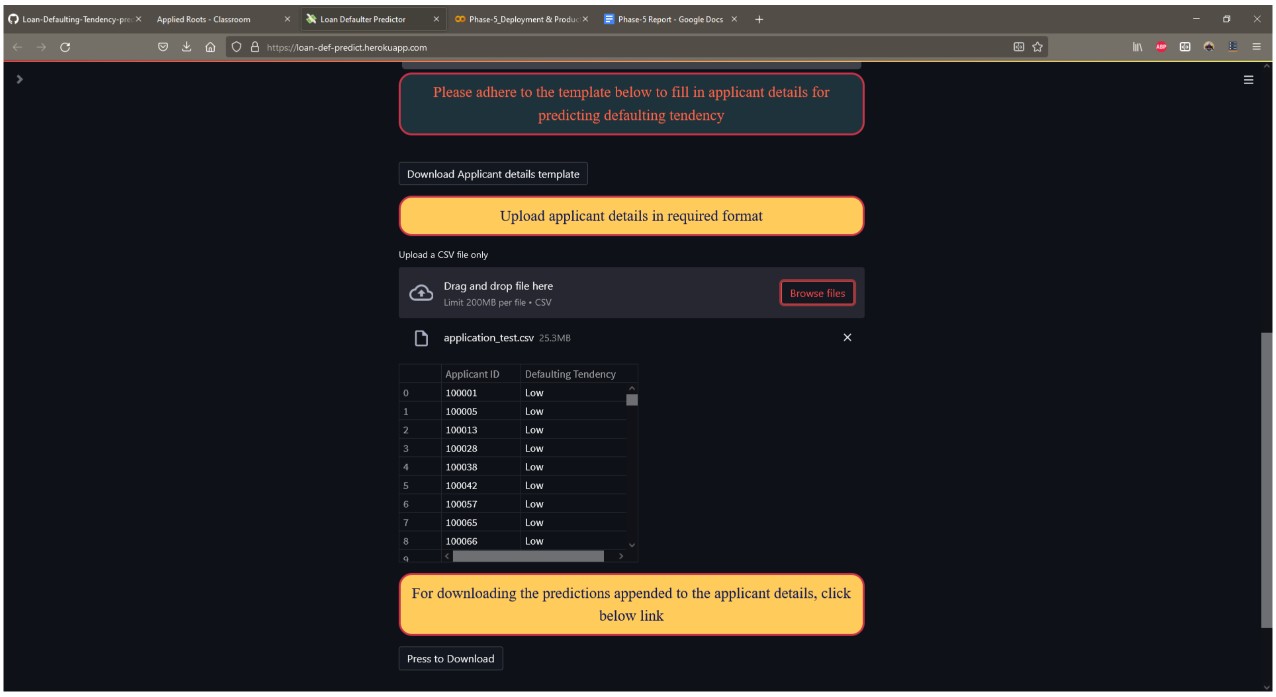


Loan-Defaulting-Tendency-pre
Applied Roots - Classroom
Loan Defaulter Predictor
Phase-5_Deployment & Produc
Phase-5 Report - Google Docs
https://loan-def-predict.herokuapp.com
LOAN DEFAULTING TENDENCY PREDICTOR

Loan Defaulting Tendency Predictor helps you identify potential defaulters.
Given a loan application of a potential or existing client at Home Credit, this app "predicts" whether the client will be able to repay the loan or not.
Applicants deemed capable of repaying the loan shall be labelled **Low** under defaulting tendency and those deemed as incapable shall be labelled as **High**.

Please adhere to the template below to fill in applicant details for predicting defaulting tendency

App home screen



Loan-Defaulting-Tendency-pre
Applied Roots - Classroom
Loan Defaulter Predictor
Phase-5_Deployment & Produc
Phase-5 Report - Google Docs
https://loan-def-predict.herokuapp.com
Download Applicant details template
Browse files
Press to Download

Please adhere to the template below to fill in applicant details for predicting defaulting tendency

Upload applicant details in required format

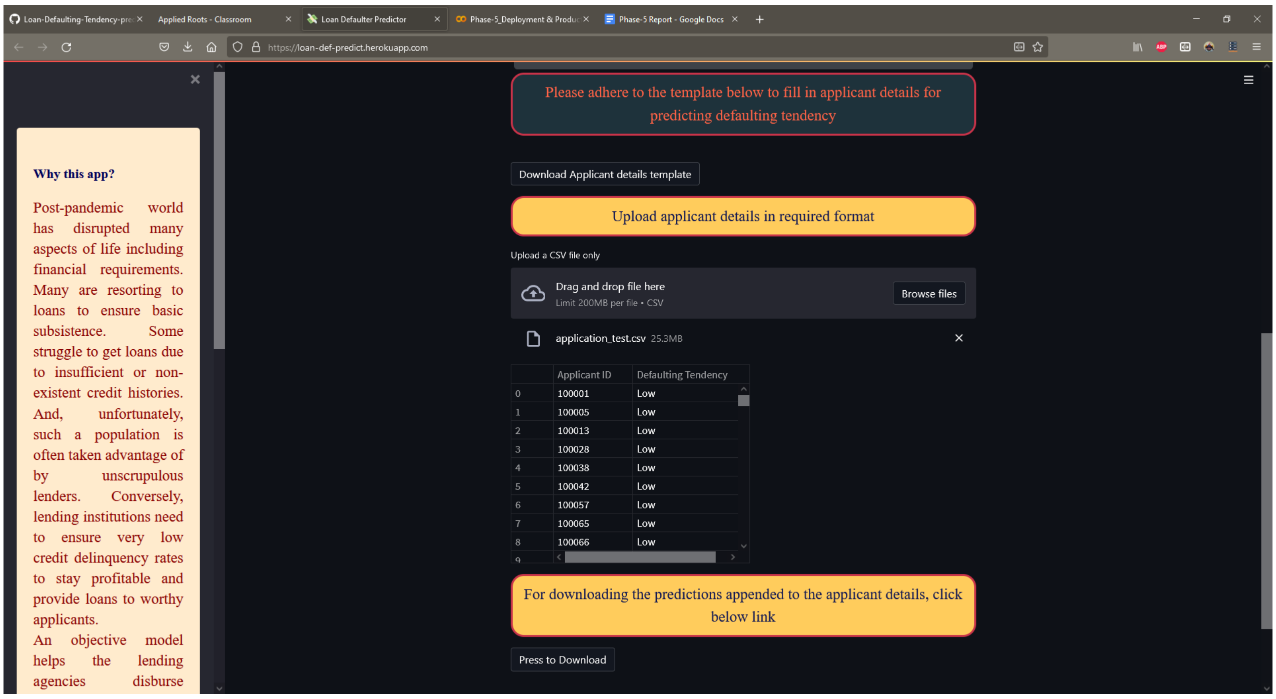
Upload a CSV file only

Drag and drop file here
Limit 200MB per file • CSV

application_test.csv 25.3MB

	Applicant ID	Defaulting Tendency
0	100001	Low
1	100005	Low
2	100013	Low
3	100028	Low
4	100038	Low
5	100042	Low
6	100057	Low
7	100065	Low
8	100066	Low
9		

For downloading the predictions appended to the applicant details, click below link


Loan-Defaulting-Tendency-pre
Applied Roots - Classroom
Loan Defaulter Predictor
Phase-5_Deployment & Produc
Phase-5 Report - Google Docs
https://loan-def-predict.herokuapp.com

Why this app?

Post-pandemic world has disrupted many aspects of life including financial requirements. Many are resorting to loans to ensure basic subsistence. Some struggle to get loans due to insufficient or non-existent credit histories. And, unfortunately, such a population is often taken advantage of by unscrupulous lenders. Conversely, lending institutions need to ensure very low credit delinquency rates to stay profitable and provide loans to worthy applicants.
An objective model helps the lending agencies disburse

Please adhere to the template below to fill in applicant details for predicting defaulting tendency

Download Applicant details template

Upload applicant details in required format

Upload a CSV file only

Drag and drop file here
Limit 200MB per file • CSV

Browse files

application_test.csv 25.3MB

	Applicant ID	Defaulting Tendency
0	100001	Low
1	100005	Low
2	100013	Low
3	100028	Low
4	100038	Low
5	100042	Low
6	100057	Low
7	100065	Low
8	100066	Low

For downloading the predictions appended to the applicant details, click below link

Press to Download

App interface with prediction & sidebar for introduction

3.0.0.0. CONCLUSION

3.1.1.0. The model and data pipeline using sklearn was successfully deployed on Heroku using the Streamlit framework. The deployed app can be viewed from here.

3.1.2.0. Latency requirements being non-stringent, the time taken to predict the defaulting tendency for the test dataset is acceptable.

3.1.3.0. As the prediction system can be run frequently, even per applicant as requirement is not real-time, the system throughput is not a cause for concern as the app hosted on the free tier Heroku box handled predictions for around 50K data points in one batch.