الجامعه المصريه للتعلم الالكترونى الأهليه

National Egyptian E-Learning University

Faculty of Computer & Information Technology

Blood Donation Locator App

By:

Ahmed Tarek Radwan	**(20-00882)**
Mario Maher Melek	**(20-00359)**
Mohammed Soliman Hassan	**(20-00262)**
Ahmed Abd Elnasser Ali	**(20-00246)**
Daniel Adel Helmy	**(20-00734)**
Maria Asaad Halem	**(20-01339)**
Marina Adel Shawky	**(20-01275)**

Under Supervision of:

DR. Mayar Ali
Lecturer in Computer and Information Technology
Egyptian E-Learning University

T.A Mohamed Mustafa
Teacher Assistant in Computer and Information Technology
Egyptian E-Learning University

Sohag 2024

Acknowledgement

First and foremost, praise and thanks be to God Almighty for His blessings all the time. Our years in college and our graduation project to complete this stage of our lives Successfully.

We have put efforts into this project. However, this would not have been possible without Support and kind assistance from many individuals and organizations.

Sincere thanks to all of them. Thanks to the Egyptian E-Learning University, especially the Sohag Center, for helping us reach to This level of awareness.

We would like to express our deep and sincere gratitude to the project supervisor, Dr / Mayar Ali for giving us the opportunity to lead us and provide invaluable support Guidance throughout this project. It has been a great privilege and honor to work and study under his supervision His directions. We are so grateful for what he has given us.

We are especially indebted to say many thanks to Eng/ Mohamed Mostafa for guidance and Constant supervision, as well as his patience, friendship, sympathy and great sensitivity kidding. His dynamism, vision, sincerity and motivation have deeply inspired us.

Finally, we can only express our gratitude to everyone who helped us during the period Graduation Project.

Abstract

There is no doubt that one of the most important issues that needs everyone's attention is blood donation and finding a blood donor as soon as possible, and we will focus in this project on solving this problem Saving lives:

Donating blood is crucial because it saves lives. Patients undergoing surgery, cancer treatment, or with serious injuries and chronic diseases often need blood transfusions to survive.

Emergencies: In emergency situations, such as accidents, natural disasters, or disease outbreaks, the demand for blood can rise dramatically. Having a readily available blood supply can be the difference between life and death for many individuals.

Chronic conditions: People with conditions such as anemia, hemophilia, and sickle cell disease often needs regular blood transfusions to maintain their health. Donating blood ensures that these patients receive the treatment they need.

Medical procedures: Many medical procedures, including surgeries, often require blood transfusions. A reliable blood supply ensures that hospitals can perform these procedures safely.

Table of Contents

Table of figures

LIST OF ACRONYMS/ABBREVIATIONS

- OCR - Optical Character Recognition
- CBC - Complete Blood Count
- SVC - Support Vector Classifier
- XgBoost - Extreme Gradient Boosting
- CV - Cross-Validation
- AI - Artificial Intelligence
- BMI - Body Mass Index
- ML - Machine Learning
- RF - Random Forest
- WBC - White Blood Cells
- RBC - Red Blood Cells
- Hemoglobin - Hemoglobin
- Platelet - Platelets
- QR code - Quick Response code
- API - Application Programming Interface
- URL - Uniform Resource Locator
- NLP - Natural Language Processing

Chapter 1: Introduction

1.1 History

The establishment of the first blood bank in Chicago in 1937 marked a pivotal moment in the history of organized blood collection and storage. Since then, technological advances have continually enhanced safety standards for donated blood. Today, technology has revolutionized the blood donation process through the development of mobile applications. These apps connect donors, blood banks, and hospitals, facilitating seamless communication and coordination. They provide user-friendly interfaces for individuals to easily register as blood donors and find nearby donation centers. Additionally, these apps offer real-time updates and notifications about blood drives and urgent blood needs, keeping donors informed and actively engaged.

Technological innovations have led to advanced blood screening techniques, ensuring the safety and reliability of donated blood. Technology improves access to blood donors and streamlines the distribution process, ultimately saving lives through timely blood provision. Modern technology also enhances the blood donation process with data analysis and artificial intelligence to improve blood stock management and predict future needs. These technologies analyze patterns and trends in blood usage, aiding in informed decision-making about blood collection and storage.

Advanced electronic tracking technologies ensure the safe and swift transfer of blood from donors to recipients, reducing errors and increasing transparency. Social media and digital platforms have made awareness campaigns about blood donation more effective. These channels enable organizations to reach a wide audience and launch targeted campaigns, increasing potential donors and encouraging regular donations. Modern applications motivate donors with features like awarding badges, digital rewards for reaching donation milestones, and organizing contests and group initiatives that foster community spirit.

Technology has also made the donation process more convenient with new techniques to reduce pain and increase donor comfort. Modern medical devices provide a smoother and safer donation experience, encouraging people to donate more frequently. In conclusion, technology has not only improved the blood donation process but has also transformed how we understand and practice donation.

1.2 Motivation

The donation process faces several challenges, impacting its effectiveness and lifesaving potential. One prominent issue is the occurrence of cases where donated blood goes unused due to insufficient quantities available when needed urgently.

This scarcity can lead to critical situations where patients are unable to receive the necessary transfusions in time, jeopardizing their health and, in some cases, resulting in tragic outcomes.

Additionally, individuals with rare blood types face a heightened risk, as finding compatible blood becomes more challenging. Without timely access to suitable blood donations, these individuals may face life-threatening situations, increasing the urgency of addressing blood donation disparities and ensuring equitable access to lifesaving resources.

Moreover, logistical challenges and inefficiencies in the blood donation and distribution system exacerbate these issues. Poor coordination between donation centers, hospitals, and patients often leads to delays and wastage of valuable blood supplies. Ensuring that the blood donation process is streamlined and efficient is crucial for maximizing the availability of blood when and where it is needed the most.

Chapter 2: Problem Phases

2.1 Problem Statement

Challenges in Finding Specific Blood Types and Their Impact on Lives:
Finding specific blood types, especially rare ones, can be both challenging and time-consuming, leading to significant implications for healthcare and patient survival.

Quick Access to Hospitals:
Lack of Knowledge About Hospital Locations for the User.

Effort and Time Required to Determine Blood Donation Possibility:
The time and effort expended by the medical staff to examine patients to Determining whether a patient can donate blood involves significant.

the problem that a person suffers from:
Sometimes the user suffers from a blood condition that prevents him from donating and the result of the possibility of donating for him is abnormal.

Communicating and urged to donate:
When there is a deficiency in a specific blood type, this represents a danger for patients survival and want to inform users registered on the application and urged them to donate.

The time and effort spent using traditional methods to maintain donation data:
Can't find blood type data leads to risk of Incompatible Blood Transfusion, Inability to Donate Blood, Missed Opportunities for Medical Intervention.

Keep quantities blood type and information about it:

Saving and return data about blood type to facilitate the process of searching for blood type.

2.2 Problem solution:

A mobile application serves both the user and hospitals.

First service for user or hospitals using logic enables the user to find the nearest hospital or blood bank that contains this blood type he is searching for and it will appear List of three nearest hospitals and the amount of blood Type in each one.

Provide map view After user choose hospital it will take him to map view which enables user to show hospital location and tracks it.

machine learning model helps the user identify a potential donation to the hospital that has the least amount of blood type.

User enters some data based on medical analysis. And if the test donation is healthy Show hospitals or blood banks where a person can donate and it is available with a Google maps.

saving the Time and effort expanded by the medical staff by equipped with a CBC test which provide the result of the analysis to determine whether the person can donate or not.

A chatbot using artificial intelligence It analyzes the message through which the patient complains about it and then predicts the disease that prevents him from donating. He can provide some important advice on dealing with this disease.

Use notifications to send a request for the required blood type at the right time to save lives to all the users registered in application contains information about the blood type, location, and contact information provided.

Create QR code This service provided to hospitals assists in creating barcodes to retrieve data about blood types.

utilizes barcode technology allows users to scan a barcode to add blood type information and can search by serial number about blood type.

Users (hospitals) can also remove a blood type from the system either by scanning itsbarcode or searching for it using its serial number but before the blood type becomes invalid (35 days).

2.3 Related Work

1- BloodLine - Blood Bank App (BDCreniputer Lab 2019):

- Post a donation request.
- Providing blood types through communication between donors and those who need blood types.

2- Red: Nehad younis 2019

- Providing blood types through communication between donors and those who need blood types

Shortcomings in BloodLine - Blood Bank App and Red App

- Time-Consuming Examination: this apps do not provide quick and easy tools for users (hospitals) to assess their eligibility to donate blood, leading to lengthy examination processes

- Complex medical Data: understanding and analyzing Complete Blood Count (CBC) reports can be complex and time-consuming.

- Lack predictive tools: identifying medical conditions that may prevent a person from donating.

- Inefficient process store: the blood donation process can be inefficient, leading to a less optimal experience for donors and collection inefficiencies for hospitals.

- Ineffective communication: between who need blood communication between blood banks and potential donors is often inefficient, leading to missed donation opportunities.

Chapter 3: Datasets

3.1 "Blooddonation Dataset "

What is "Blooddonation" Dataset?
The Locator Donation project utilizes the "blooddonation Dataset," A carefully curated dataset, which provides basic details about the valid values that prepare a user to donate, here are some important details about this dataset:

1. Size: blooddonation dataset comprises 3013 datapoints, each record in the dataset contains information about individual attributes that determine their eligibility to donate or not

2. Structure: The dataset is divided into 10 columns: " Blood_Pressure_Abnormality " , " Level_of_Hemoglobin ", " Age ", "BMI", "Sex", " Pregnancy ", " Smoking ", " Chronic_kidney_disease ", " Adrenal_and_thyroid_disorders "and " Result " The Label column contains the patient label associated with all of these values.

3. The dataset encompasses 3012 patient records, each characterized by 9 distinct values. This diversity of values offers a comprehensive overview of prevalent health conditions among the patients represented in the dataset.

4. The blood donation dataset serves as the foundation for training and validating machine learning models. This process improves the accuracy and reliability of our donation locator system by accurately classifying whether the individual's values indicate suitability for donation or not.

	Blood_Pressure_Abnormality	Level_of_Hemoglobin	Age	BMI	Sex	Pregnancy	Smoking	Chronic_kidney_disease	Adrenal_and_thyroid_disorders	Result
0	1	11.28	34.0	23.0	1	1	0	1	1	0
1	0	9.75	54.0	33.0	1	0	0	0	0	0
2	1	10.79	70.0	49.0	0	0	0	1	0	0
3	0	11.00	71.0	50.0	0	0	0	1	0	0
4	1	14.17	52.0	19.0	0	0	0	0	0	0
5	0	11.64	23.0	48.0	0	0	1	0	0	0
6	1	11.69	43.0	41.0	1	1	0	1	1	0
7	0	12.70	48.0	20.0	0	0	0	0	0	0
8	0	10.88	72.0	44.0	0	0	0	0	0	0
9	1	14.56	40.0	44.0	0	0	0	0	0	0

Fig 1: Blooddonation" Dataset

3.2 “Symptom2Disease" Dataset

What is "Symptom2Disease" Dataset?
The chatbot project utilizes the "Symptom2Disease Dataset," a meticulously curated dataset, which provides essential details about the disease, here are some important details about this dataset:

1. Size: The Symptom2Disease dataset comprises 1200 datapoints, each containing information about a specific symptom description and its corresponding disease label.

2. Structure: The dataset is divided into two columns: "label" and "text", where the "label" Column contains the disease label associated with each symptom description.

3. Disease Variety: The dataset includes 42 diseases with 50 symptom descriptions, providing a comprehensive overview of common health conditions.

Data Configuration:

1. Data Augmentation: We utilized data augmentation techniques like Paraphrase online and Paraphrase to expand the dataset size, resulting in 2500 datapoints, enhancing its diversity and granularity.

2. Quality Assurance: The data augmentation process involved rigorous quality assurance measures to ensure accuracy and relevance, preserving the semantic meaning and clinical relevance of original symptom descriptions.

3. Training and Validation: The Symptom2Disease Dataset is used to train and validate machine learning models, enhancing the efficacy and reliability of our healthcare chatbot by accurately classifying diseases based on user-input symptoms.

index	label	text
439	Pneumonia	I've recently been suffering with chills, lethargy, a cough, a high temperature, and difficulties breathing. I've been sweating profusely and generally feeling ill and weak. I've also had some quite thick and red phlegm.
308	Fungal infection	My skin has been acting up recently, becoming extremely itchy and rashes-prone. Additionally, there are certain spots that deviate from my natural skin tone in terms of hue. And now my skin has these lumps or bumps that weren't there before.
1184	diabetes	My emotions change, and I have difficulties focusing. At times, my mind might be cloudy and hazy, making it challenging for me to do even simple tasks.
580	Acne	A skin rash with several pus-filled pimples and blackheads has been bothering me lately. Additionally, my skin has been scurring a lot.
371	Common Cold	I keep sneezing, and my eyes don't quit dripping. It's incredibly difficult for me to breathe because it feels like there is something trapped in my throat. I often feel exhausted, and lately, I've had a lot of phlegm. Moreover, my lymph nodes are enlarged.
275	Dengue	I've been feeling sick and feel a strong need to vomit. There is a sharp ache behind my eyes, and swollen red dots all over my back.
526	Arthritis	I've been experiencing stiffness and weakness in my neck muscles recently. Since my joints have grown, it's hard for me to walk without getting stiff. Additionally, walking has been extremely uncomfortable.
417	Pneumonia	I'm having a really hard time catching my breath and I'm sweating a lot. I feel really sick and have a lot of mucus in my throat. My chest hurts and my heart is racing. The mucus I'm coughing up is dark and looks like rust.
820	Jaundice	I've been feeling extremely scratchy, sick, and worn out. In addition, I've lost weight and have a temperature. My urine is dark, and my skin has turned yellow. Additionally, I have been experiencing stomach pain.

Fig : 2 "Symptom2Disease" Dataset

3.3 "symptom_precaution" Dataset

What is " symptom_precaution " Dataset?

The chatbot utilizes the " symptom_precaution " a meticulously curated dataset, Which provides precise details on how to deal with the disease, here are some important details about this dataset:

1. Size: The symptom_precaution dataset comprises 42 datapoints, Each contains how to deal with the disease.

2. Structure: The dataset is divided into five columns: "Disease","Precaution_1", "Precaution_2""Precaution_3"and "Precaution_4", where the " Disease " The column contains the label of the disease associated with the actions that the patient should take in consideration

3. Disease Variety: The dataset includes 42 diseases and each disease have 4 precaution that help user to know how to deal with the disease.

The purpose

The dataset's primary objective is to offer actionable guidance to individuals diagnosed with particular diseases. Following the suggested precautions enables patients to effectively manage their conditions and alleviate symptoms. This dataset finds application in healthcare platforms, patient management systems, and health advisory services, where it tailors precautionary measures specific to different diseases, enhancing patient care and outcomes..

	Disease	Precaution_1	Precaution_2	Precaution_3	Precaution_4
0	Drug Reaction	stop irritation	consult nearest hospital	stop taking drug	follow up
1	Malaria	Consult nearest hospital	avoid oily food	avoid non veg food	keep mosquitos out
2	Allergy	apply calamine	cover area with bandage	NaN	use ice to compress itching
3	Hypothyroidism	reduce stress	exercise	eat healthy	get proper sleep
4	Psoriasis	wash hands with warm soapy water	stop bleeding using pressure	consult doctor	salt baths
5	GERD	avoid fatty spicy food	avoid lying down after eating	maintain healthy weight	exercise
6	Chronic cholestasis	cold baths	anti itch medicine	consult doctor	eat healthy
7	hepatitis A	Consult nearest hospital	wash hands through	avoid fatty spicy food	medication
8	Osteoarthristis	acetaminophen	consult nearest hospital	follow up	salt baths

Fig 3: "symptom_precaution" Dataset

Chapter 4 Implementation

4.1 Introduction of Blood Donation

To determine if a user is eligible to donate blood, we gather various health-related and demographic information from them. The information we collect includes:

- **Blood Pressure**: Blood pressure readings can indicate if the user has hypertension or hypotension, which can affect eligibility.
- **Hemoglobin Level**: Adequate hemoglobin levels are necessary to ensure safe blood donation.
- **Age**: There are age restrictions for blood donation, typically ranging from 18 to 65 years.
- **BMI (Body Mass Index)**: BMI helps assess the user's overall health and fitness for donation.
- **Sex**: Different criteria may apply based on the user's sex due to physiological differences.
- **Pregnancy**: Pregnant individuals are generally not eligible to donate blood.
- **Smoking Status**: Smoking can impact overall health and blood quality.
- **Chronic Diseases**: Chronic conditions such as diabetes or heart disease can affect eligibility.
- **Adrenal and Thyroid Disorders**: These disorders can influence the user's overall health and suitability for blood donation.

Using this information, we apply a supervised machine learning model to predict whether the user is eligible to donate blood. Supervised machine learning involves training a model on a labeled dataset, where the outcome (eligible or not eligible) is known. The model learns patterns and associations in the data, allowing it to make predictions on new, unseen data.

By leveraging supervised machine learning, we can effectively assess the eligibility of users for blood donation based on their health and demographic information. This approach ensures a systematic and data-driven decision-making process.

4.2 Grid Search

- Grid Search is a hyperparameter tuning technique used to find the optimal hyperparameters for a given machine learning model. It exhaustively searches through a specified parameter grid to determine the best combination of parameters that yield the highest performance for the model. Below is a detailed explanation of why Grid Search is used and how to implement it in Python using the GridSearchCV class from the scikit-learn library.

➢ **Why Use Grid Search?**

- Hyperparameters are parameters that are not learned by the model during training. They are set prior to the training process and can significantly impact the model's performance. Examples of hyperparameters include the regularization strength in logistic regression, the number of trees in a random forest, and the learning rate in gradient boosting.

Grid Search helps in:

- **Optimizing Model Performance**: By finding the best hyperparameter combination, Grid Search helps improve the model's accuracy and generalization capability.
- **Systematic and Comprehensive Search**: It exhaustively explores all possible combinations within a specified range, ensuring a thorough search for the best parameters.
- **Reliable Evaluation**: Grid Search uses cross-validation to assess the performance of each hyperparameter combination, providing a robust evaluation metric and reducing the risk of overfitting.
- **Automation of Hyperparameter Tuning**: It automates the process of hyperparameter tuning, saving time and ensuring consistency in the search process.

How Grid Search Works

1. **Define the parameter grid**: Specify the hyperparameters and their possible values in a dictionary form.
2. **Instantiate the model**: Choose the machine learning algorithm you want to tune.
3. **Set up the Grid Search**: Use GridSearchCV to combine the parameter grid, the model, and the cross-validation scheme.
4. **Fit the Grid Search**: Train the model on different combinations of hyperparameters.

5. **Find the best parameters**: Extract the best combination of hyperparameters that resulted in the highest performance.

To optimize the performance of our models, we utilized Grid Search across all models. Below, we will demonstrate the specific models we employed and
how Grid Search was applied to each.

4.3 Support Vector Machine (SVC)

- The Support Vector Classifier (SVC) is a supervised machine learning algorithm used for classification tasks. It aims to find the optimal hyperplane that separatesthe data into different classes with the maximum margin.

How It Works:

1. **Selecting Hyperplanes**:
   - SVC attempts to find the hyperplane that best separates the classes in the feature space. If the data is linearly separable, the hyperplane will divide the classes perfectly.
2. **Maximizing the Margin**:
   - The algorithm selects the hyperplane that maximizes the margin between the classes. The margin is the distance between the hyperplane and the nearest data points from each class, known as support vectors.

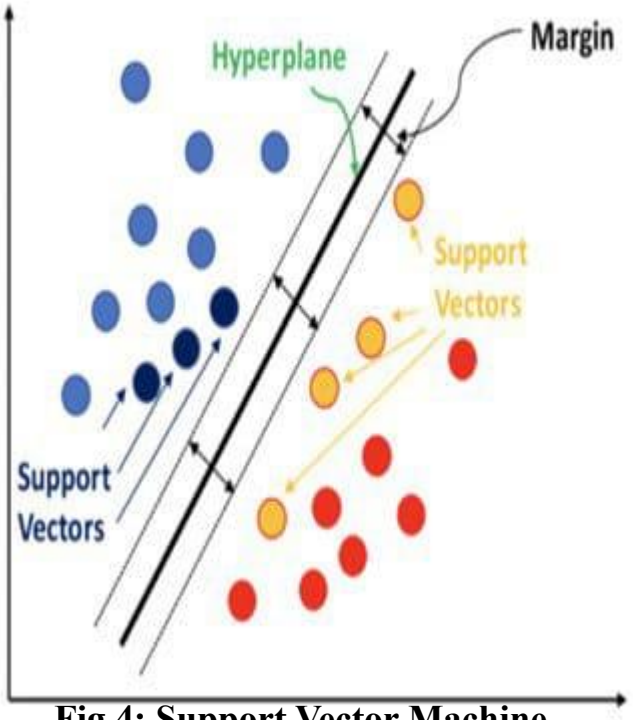


Fig 4: Support Vector Machine

3. **Handling Non-Linearly Separable Data**:
   - For data that is not linearly separable, SVC uses a technique called the kernel trick. It transforms the original feature space into a higher-dimensional space where a hyperplane can be used to separate the classes.
4. **Choosing the Kernel Function**:
   - Different kernel functions (e.g., linear, polynomial, radial basis function (RBF)) can be used to map the data into a higher-dimensional space. The choice of kernel affects the decision boundary and overall performance of the classifier.
5. **Regularization**:
   - The regularization parameter (C) controls the trade-off between achieving a low error on the training data and minimizing the margin's size. A smaller C value allows for a larger margin and some misclassifications, while a larger C value aims for a narrower margin with fewer misclassifications.

Now, let's delve into the implementation details of the support vector machine using our dataset. Below is the code we used, along with a detailed explanation of each step.

➢ Implementation

```
from sklearn.model_selection import GridSearchCV, train_test_split

svm_model = svm.SVC()
param_grid = {
    'C': [0.1, 1, 10],
    'kernel': ['linear', 'rbf'],
    'gamma': ['scale', 'auto']
}

# Perform grid search
grid_search = GridSearchCV(svm_model, param_grid, cv=4)
grid_search.fit(X_train, y_train)
best_params = grid_search.best_params_
best_svm_model = svm.SVC(**best_params)
best_svm_model.fit(X_train, y_train)
train_score = best_svm_model.score(X_train, y_train)
test_score = best_svm_model.score(X_test, y_test)
print(f'The Accuracy : {round(test_score*100, 2)} %')

print("Best Hyperparameters:", best_params)
```

```
The Accuracy : 98.34 %
Best Hyperparameters: {'C': 10, 'gamma': 'scale', 'kernel': 'linear'}
```

Fig 5: Support Vector Machine (SVC) Model

1. **Import necessary libraries**:
- GridSearchCV is used for performing grid search with cross-validation
- train_test_split is used to split the dataset into training and test sets.
- svm is the module for Support Vector Machine classifiers.

2. **Create an instance of the SVM model**:
- svm.SVC() creates an instance of the Support Vector Classifier.

3. **Define the parameter grid**
- C: Regularization parameter. A higher value of C tries to fit the training data better.
- kernel: Specifies the kernel type to be used in the algorithm. 'linear' and 'rbf' are the options considered.
- gamma: Kernel coefficient for 'rbf'. 'scale' and 'auto' are the options considered.

4. **Perform grid search with 4-fold cross-validation**:

- GridSearchCV performs an exhaustive search over the specified parameter grid.
- cv=4 specifies 4-fold cross-validation, which splits the training data into
- parts and trains the model 4 times, each time using a different part as the validation set

5. **Retrieve the best hyperparameters**:

- grid_search.best_params_ returns the best combination of hyperparameters found during the grid search.

6. **Create a new SVM model using the best hyperparameters**:

- svm.SVC(**best_params) creates a new SVM model with the best hyperparameters.
- best_svm_model.fit(X_train, y_train) trains the new SVM model on the entire training set.

7. **Print the accuracy on the test set and the best hyperparameters found** :

1) Prints the accuracy of the model on the test set.
2) Prints the best combination of hyperparameters found by the grid search.

4.4 XgBoost

- XGBoost is a supervised machine learning algorithm primarily used for regression and classification tasks. It belongs to the ensemble learning methods and is known for its speed and performance in various machine learning competitions.

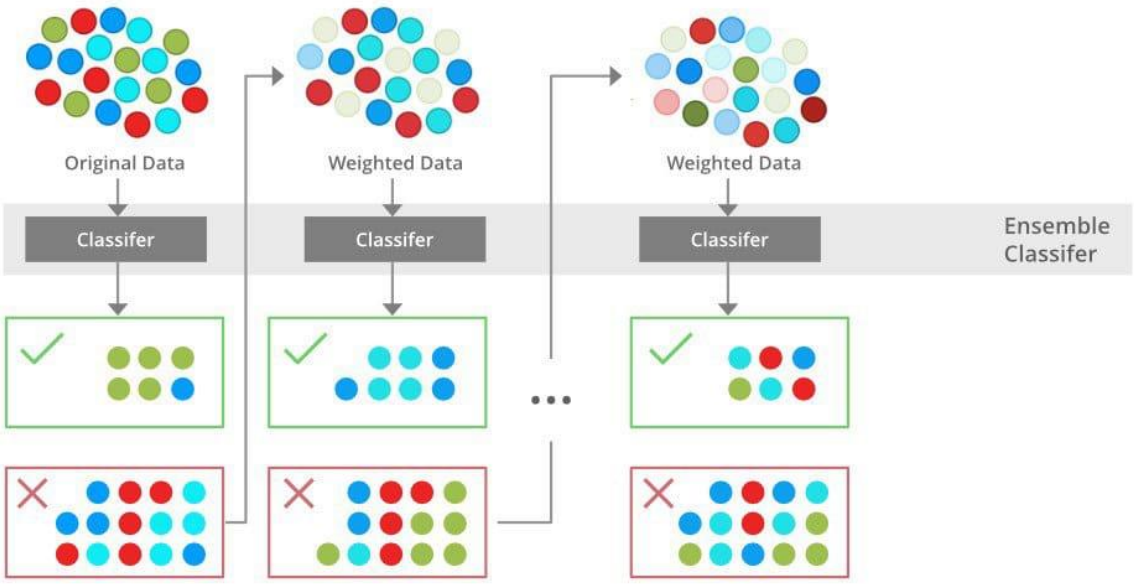


Fig 6: XgBoost

How It Works:

- **Boosting Technique:** XGBoost employs a boosting technique where multiple weak learners (decision trees) are combined to create a strong learner.
- **Gradient Boosting:** It uses a gradient boosting framework where each new tree corrects errors made by previously trained trees, leading to a more accurate model over iterations.
- **Decision Trees:** XGBoost's base learners are decision trees. Initially, a simple tree is built, and subsequent trees are added to correct errors, with each tree focusing on the mistakes of the previous ones.
- **Regularization:** Similar to SVC's regularization parameter (C), XGBoost has regularization parameters like `gamma`, `alpha`, and `lambda`. These parameters control the complexity of the trees, preventing overfitting and improving generalization.

- **Learning Rate:** XGBoost also has a learning rate parameter (`eta`), which scales the contribution of each tree. A lower learning rate makes the model more robust by shrinking the impact of each tree.
- **Feature Importance:** XGBoost provides insights into feature importance, showing which features are most influential in making predictions. This is beneficial for feature selection and understanding the data.
- **Parallel Processing:** XGBoost is designed for parallel processing, making it efficient for large datasets and speeding up training times compared to traditional gradient boosting algorithms.
- **Handling Missing Values:** XGBoost has built-in capabilities to handle missing values, reducing the need for preprocessing steps like imputation.
- **Scalability:** XGBoost is highly scalable and can handle large datasets with millions of samples and features efficiently.

❑ Now, let's delve into the implementation details of the Xgboost using our dataset. Below is the code we used, along with a detailed explanation of each step.

- **Implementation**

```
]: from sklearn.model_selection import GridSearchCV
   xgboost_model = XGBClassifier()
   param_grid = {
       'n_estimators': [50, 100, 200],
       'max_depth': [3, 5, 7],
       'learning_rate': [0.01, 0.1, 0.2],
   }
   grid_search = GridSearchCV(xgboost_model, param_grid, cv=3)
   grid_search.fit(X_train, y_train)
   best_params = grid_search.best_params_
   best_xgboost_model = XGBClassifier(**best_params)
   best_xgboost_model.fit(X_train, y_train)
   train_score = best_xgboost_model.score(X_train, y_train)
   test_score = best_xgboost_model.score(X_test, y_test)
   print(f'The Accuracy : {round(test_score*100, 2)} %')
   print("Best Hyperparameters:", best_params)

   The Accuracy : 99.67 %
   Best Hyperparameters: {'learning_rate': 0.1, 'max_depth': 7, 'n_estimators': 100}
```

Fig 7: XGBoost Model

1.Import necessary libraries:

- GridSearchCV is used for performing grid search with cross-validation.
- XGBClassifier is the XGBoost classifier from the xgboost library.

2.Create an instance of the XGBoost model:

- XGBClassifier() creates an instance of the XGBoost classifier.

3.Define the parameter grid

- n_estimators: The number of boosting rounds (trees).
- max_depth: The maximum depth of each tree. Increasing this value makes the model more complex.
- learning_rate: The rate at which the model learns. Lower values require more boosting rounds.

4.Perform grid search with 3-fold cross-validation:

- GridSearchCV performs an exhaustive search over the specified parameter grid.
- cv=3 specifies 3-fold cross-validation, which splits the training data into 3 parts and trains the model 3 times, each time using a different part as

5.Retrieve the best hyperparameters:

- grid_search.best_params_ returns the best combination of hyperparameters found during the grid search.

6. **Create a new XGBoost model using the best hyperparameters**:

- XGBClassifier(**best_params) creates a new XGBoost model with the best hyperparameters.
- best_xgboost_model.fit(X_train, y_train) trains the new XGBoost model on the entire training set.

7.Print the accuracy on the test set and the best hyperparameters found :

- Prints the accuracy of the model on the test set.
- Prints the best combination of hyperparameters found by the grid search.

4.5 LogisticRegression

- Logistic Regression is a supervised machine learning algorithm used for binary classification tasks. Despite its name, it's a regression algorithm that predicts the probability of a binary outcome (0 or 1, Yes or No) based on input features.

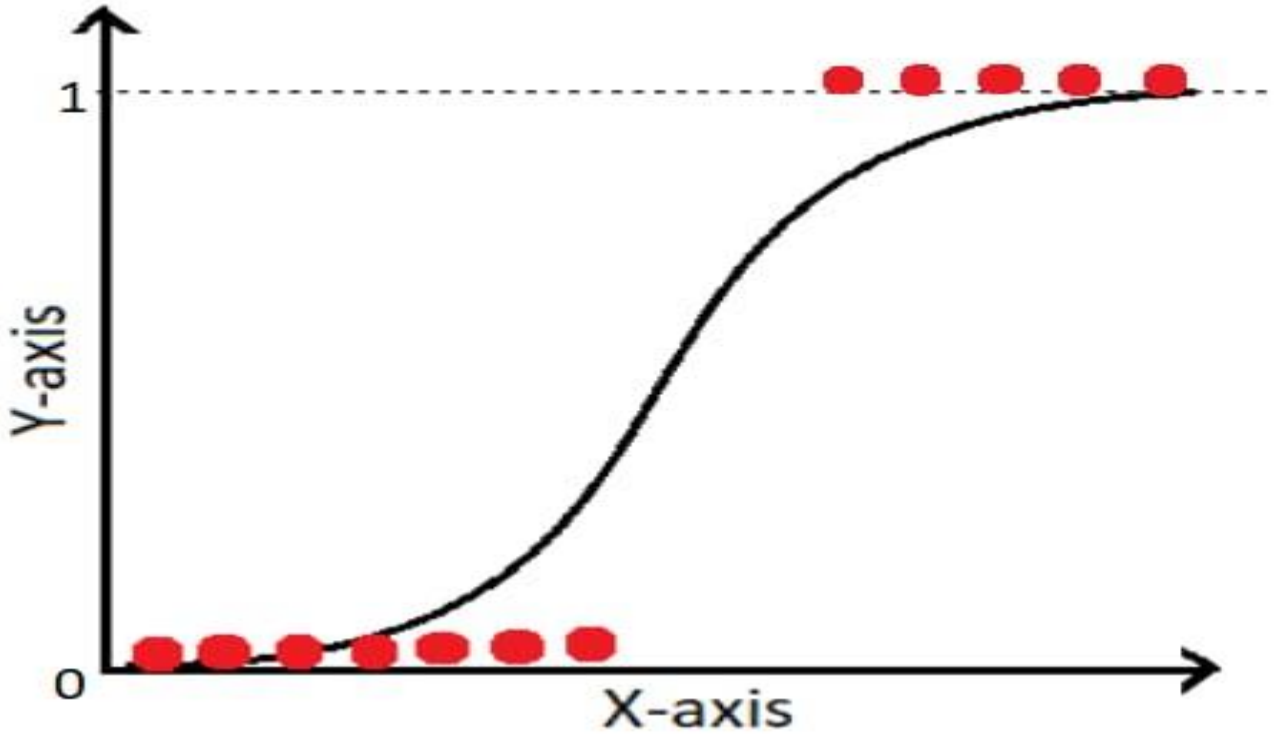


Fig 8: LogisticRegression

How It Works:

- **Sigmoid Function:** Logistic Regression uses the sigmoid function to map predicted values to probabilities between 0 and 1. The sigmoid function transforms the output of the linear combination of input features into a probability score.

- **Linear Decision Boundary:** The algorithm creates a linear decision boundary that separates the data points belonging to different classes. For binary classification, this boundary is a straight line in the feature space.

- **Probability Threshold:** A probability threshold (usually 0.5) is used to classify data points into classes. If the predicted probability is above the threshold, the data point is classified as 1; otherwise, it's classified as 0.

- **Training Process:** Logistic Regression is trained using optimization techniques like Gradient Descent or its variants. The model learns the optimal coefficients (weights) for each feature to minimize the error between predicted probabilities and actual class labels.

- **Regularization:** Similar to other algorithms, Logistic Regression also has regularization techniques like L1 (Lasso) and L2 (Ridge) regularization. These techniques help prevent overfitting by penalizing large coefficients.

- **Interpretability:** Logistic Regression is interpretable as it provides coefficients for each feature, indicating the feature's impact on the predicted probability. Positive coefficients increase the probability of the positive class, while negative coefficients decrease it.
- **Assumptions:** Logistic Regression assumes that the relationship between input features and the log-odds of the outcome is linear. It also assumes independence of observations and absence of multicollinearity among features.

- **Binary and Multiclass Classification:** While primarily used for binary classification, Logistic Regression can be extended to handle multiclass classification tasks using techniques like One-vs-Rest (OvR) or One-vs-One (OvO).

Now, let's delve into the implementation details of the Logistic Regression using our dataset. Below is the code we used, along with a detailed explanation of each step.

➢ Implementation

```
logisticRegr = LogisticRegression()
param_grid = {
    'penalty': ['l1', 'l2'],
    'C': [0.1, 1, 10],
}
grid_search = GridSearchCV(logisticRegr, param_grid, cv=3)
grid_search.fit(X_train, y_train)
best_params = grid_search.best_params_
best_logisticRegr = LogisticRegression(**best_params)
best_logisticRegr.fit(X_train, y_train)
predictions = best_logisticRegr.predict(X_test)
accuracy = accuracy_score(y_test, predictions)

print("Accuracy: {:.2f}%".format(accuracy * 100))
print("Best Hyperparameters:", best_params)
```

Fig 9: LogisticRegression model

1. **Import necessary libraries**:
    a. GridSearchCV is used for performing grid search with cross-validation.
    b. LogisticRegression is the logistic regression model from the sklearn.linear_model module.
    c. accuracy_score is used to calculate the accuracy of the model's predictions
2. **Create an instance of the Logistic Regression model**:
    a. LogisticRegression() creates an instance of the Logistic Regression classifier.
3. **Define the parameter grid**
    a. penalty: Specifies the norm used in the penalization. 'l1' and 'l2' are the options considered.
    b. C: Inverse of regularization strength; smaller values specify stronger regularization..
4. **Perform grid search with 3-fold cross-validation**:
    a. GridSearchCV performs an exhaustive search over the specified parameter grid.
    b. cv=3 specifies 3-fold cross-validation, which splits the training data into 3 parts and trains the model 3 times, each time using a different part as the validation set
5. **Retrieve the best hyperparameters**:
    a. grid_search.best_params_ returns the best combination of hyperparameters found during the grid search.
6. **Create a new Logistic Regression model using the best hyperparameters**:
    a. LogisticRegression(**best_params) creates a new Logistic Regression model with the best hyperparameters.
    b. best_logisticRegr.fit(X_train, y_train) trains the new Logistic Regression model on the entire training set.
7. **Print the accuracy on the test set and the best hyperparameters found** :
    a. Prints the accuracy of the model on the test set.
    b. Prints the best combination of hyperparameters found by the grid search.

4.6 DecisionTreeClassifier

- The Decision Tree Classifier is a supervised machine learning algorithm used for classification tasks. It partitions the dataset based on feature values to make predictions, following a tree-like model of decisions.

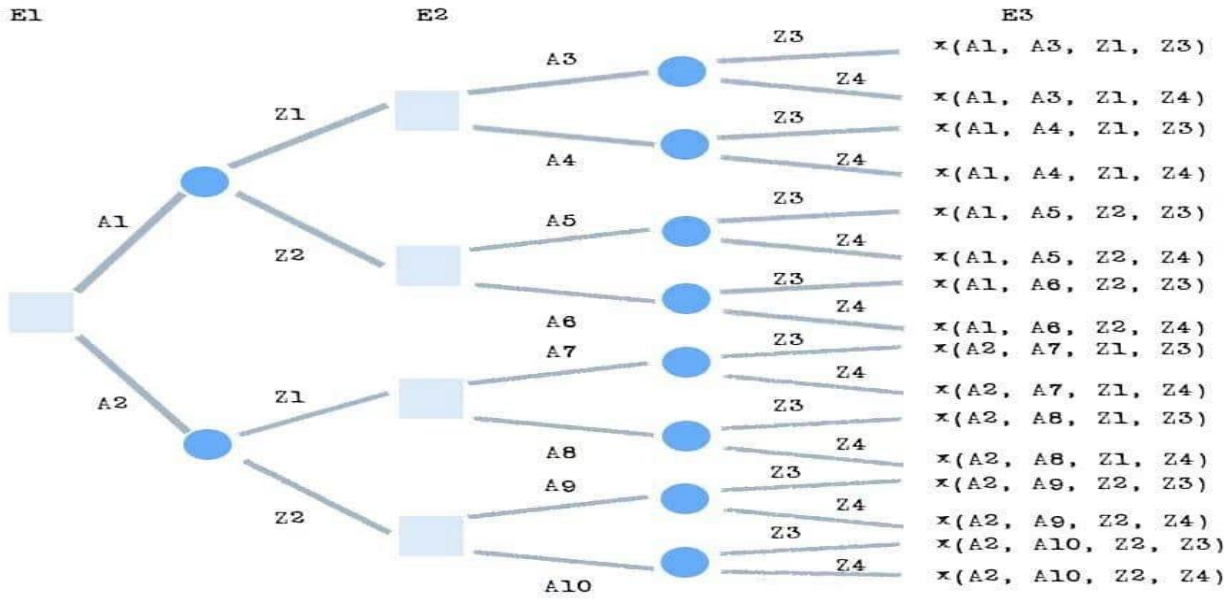


Fig 10: DecisionTreeClassifier

How It Works:

1. **Choosing the Best Feature to Split the Dataset**:
   a. The algorithm evaluates all features and selects the one that best separates the data according to a chosen criterion (e.g., Gini impurity or information gain).
2. **Dividing Data Based on Chosen Feature Values**:
   a. Once the best feature is selected, the dataset is divided into subsets based on the feature's values.
3. **Recursively Splitting Subsets**:
   a. The process of selecting the best feature and splitting the data is repeated recursively for each subset until stopping conditions are met (e.g., maximum depth of the tree, minimum number of samples per leaf, or no further improvement in classification).
4. **Building the Decision Tree**:
   a. The decision tree is constructed with branches representing decisions based on feature values and leaf nodes representing the final classification outcomes.
5. **Prediction**:

a. To predict the target class for new input data, the algorithm traverses the decision tree from the root to a leaf node, following the path dictated by the feature values of the input data.

- Now, let's delve into the implementation details of the Decision Tree Classifier using our dataset. Below is the code we used, along with a detailed explanation of each step.

Implementation

```
decision_tree_model = DecisionTreeClassifier()
param_grid = {
    'criterion': ['gini', 'entropy'],
    'max_depth': [None, 5, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
}
grid_search = GridSearchCV(decision_tree_model, param_grid, cv=3)
grid_search.fit(X_train, y_train)
best_params = grid_search.best_params_
best_decision_tree_model = DecisionTreeClassifier(**best_params)
best_decision_tree_model.fit(X_train, y_train)
y_pred_dt = best_decision_tree_model.predict(X_test)
accuracy_dt = accuracy_score(y_test, y_pred_dt)

print(f"Decision Tree Accuracy: {accuracy_dt:.2f}")
print("Best Hyperparameters:", best_params)
```

```
Decision Tree Accuracy: 0.99
Best Hyperparameters: {'criterion': 'entropy', 'max_depth': 20, 'min_samples_leaf': 1, 'min_samples_split': 10}
```

Fig 11: DecisionTreeClassifier model

1) **Import necessary libraries**:
   - GridSearchCV is used for performing grid search with cross-validation.
   - DecisionTreeClassifier is the decision tree model from the sklearn.tree module.
   - accuracy_score is used to calculate the accuracy of the model's predictions.

2) **Create an instance of Decision Tree model**:
   - DecisionTreeClassifier() creates an instance of the Decision Tree classifier.

3) **Define the parameter grid**:
   - criterion: Function to measure the quality of a split ('gini' for Gini impurity, 'entropy' for information gain).
   - max_depth: Maximum depth of the tree. If None, nodes are expanded until all leaves contain less than min_samples_split samples.
   - min_samples_split: Minimum number of samples required to split an internal node.
   - min_samples_leaf: Minimum number of samples required to be at a leaf node.

4) **Perform grid search with 3-fold cross-validation**:
   - GridSearchCV performs an exhaustive search over the specified parameter grid.
   - cv=3 specifies 3-fold cross-validation, which splits the training data into 3 parts and trains the model 3 times, each time using a different part as the validation set.

5) **Create a new Decision Tree model using the best hyperparameters**:
   - DecisionTreeClassifier(**best_params) creates a new Decision Tree model with the best hyperparameters.
   - best_decision_tree_model.fit(X_train, y_train) trains the new Decision Tree model on the entire training set..

6) **Print the accuracy on the test set and the best hyperparameters found** :
   - Prints the accuracy of the model on the test set.
   - Prints the best combination of hyperparameters found by the grid search.

4.7 RandomForestRegressor

Random Forest Regressor is an ensemble learning algorithm used for regression tasks. It's an extension of the Random Forest algorithm, which combines multiple decision trees to improve predictive accuracy and reduce overfitting.

How It Works:

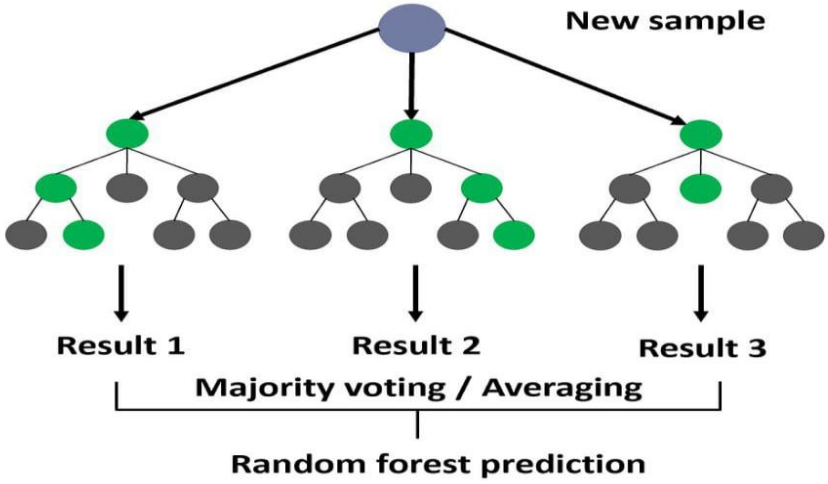


Fig 12: RandomForestRegressor

- **Decision Trees:** Random Forest Regressor is built on a collection of decision trees, where each tree makes predictions independently. These trees are trained on random subsets of the data (bootstrap samples) and random subsets of features.

- **Ensemble Learning:** The algorithm aggregates predictions from multiple decision trees to make the final prediction. For regression tasks, the predictions from individual trees are averaged to obtain the ensemble prediction.

- **Random Subsampling:** Random Forest Regressor uses random subsampling of data points and features to introduce diversity among the trees. This randomness helps reduce overfitting and improves the model's generalization performance.

- **Feature Importance:** The algorithm provides insights into feature importance by evaluating how much each feature contributes to reducing the variance in predictions across the ensemble of trees. This information helps in feature selection and understanding the data.

- **Bootstrap Aggregating (Bagging):** Random Forest Regressor employs a technique called bagging, where each tree is trained on a random subset of the training data with replacement. This technique further enhances the model's robustness and stability.

- **Hyperparameters:** Random Forest Regressor has hyperparameters like the number of trees (`n_estimators`), maximum depth of trees (`max_depth`), minimum samples required to split a node (`min_samples_split`), and others. Tuning these hyperparameters can improve the model's performance and prevent overfitting.

- **Out-of-Bag Error:** The algorithm uses out-of-bag (OOB) samples, which are data points not included in the bootstrap sample used to train each tree. These samples are used to estimate the model's performance without the need for a separate validation set.

- **Scalability:** Random Forest Regressor can handle large datasets with high dimensionality efficiently. It is parallelizable, allowing for faster training on multi-core processors or distributed computing frameworks

Now, let's delve into the implementation details of the Random Forest using our dataset. Below is the code we used, along with a detailed explanation of each step.

➢ Implementation

```
model_r = RandomForestRegressor()
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
}
grid_search = GridSearchCV(model_r, param_grid, cv=3)
grid_search.fit(X_train, y_train)
best_params = grid_search.best_params_
best_model_r = RandomForestRegressor(**best_params)
best_model_r.fit(X_train, y_train)
accuracy_train = best_model_r.score(X_test, y_test)
y_pred = best_model_r.predict(X_test)
accuracy = r2_score(y_test, y_pred)

print(f"Accuracy of RandomForestRegressor is: {round(accuracy * 100, 2)} %")
print("Best Hyperparameters:", best_params)
```

Fig 13: RandomForestRegressor model

1 **Import necessary libraries**:

- GridSearchCV is used for performing grid search with cross-validation.
- RandomForestRegressor is the random forest regressor model from the sklearn.ensemble module.
- r2_score is used to calculate the R^2 score of the model's predictions.

2 **Create an instance of Random Forest Regressor model**:

- RandomForestRegressor() creates an instance of the Random Forest Regressor.

3 **Define the parameter grid**:

- n_estimators: Number of trees in the forest.
- max_depth: Maximum depth of the tree. If None, nodes are expanded until all leaves are pure or contain less than min_samples_split samples.
- min_samples_split: Minimum number of samples required to split an internal node

4 **Perform grid search with 3-fold cross-validation**:

- GridSearchCV performs an exhaustive search over the specified parameter grid.
- cv=3 specifies 3-fold cross-validation, which splits the training data into 3 parts and trains the model 3 times, each time using a different part as the validation set.

5 Make predictions and Calculate the R^2 score:

- best_model_r.predict(X_test) generates predictions on the test data.
- r2_score(y_test, y_pred) calculates the R^2 score of the model's predictions on the test set.

6 Create a new Decision Tree model using the best hyperparameters:

- RandomForestRegressor(**best_params) creates a new Random Forest Regressor model with the best hyperparameters.
- best_model_r.fit(X_train, y_train) trains the new Random Forest Regressor model on the entire training set

7 Print the accuracy on the test set and the best hyperparameters found :

- Prints the accuracy of the model on the test set.
- Prints the best combination of hyperparameters found by the grid search.

4.8 Accuracy

- After evaluating the performance of five different machine learning models based on their accuracy metrics, we have selected XGBoost as the model to integrate with our application.

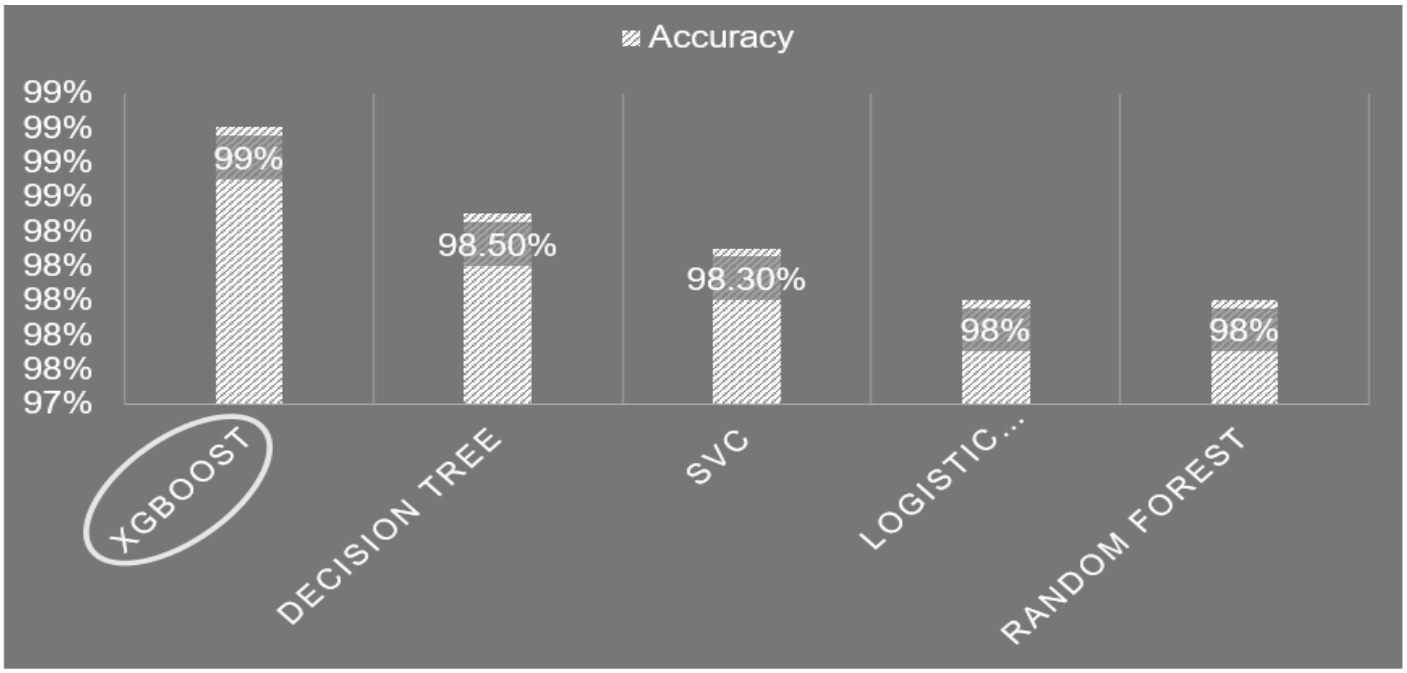


Fig 14: Accuracy

4.9 Optical Character Recognition (OCR) system

Definition: OCR is a technology that converts various types of documents, such as scanned paper, documents, PDF files, or images captured by a digital camera, into editable and searchable data.

Functionality in the app: In this app, Optical Character Recognition (OCR) technology is used to analyze images of CBC test reports uploaded by users. This involves scanning the image, recognizing text and numeric values, and converting them into digital data that can be further processed.[1]

Analysis of uploaded CBC test images: Uploading is easy to use Users can upload images of their CBC test reports directly through the app. This can be done by taking a photo with their phone camera or uploading an existing photo from their device.

Saving effort and time: This feature eliminates the need to manually enter data, saving users significant time and effort. Instead of manually entering blood test results, users simply take a photo and let the app do the rest.

Determine eligibility for blood donation: Automatic Analysis: Once the CBC report image is uploaded, OCR technology extracts the relevant blood parameters, such as hemoglobin levels, white blood cell count, platelet count, etc.

Eligibility Criteria: The application then compares these extracted values with the criteria set for blood donation eligibility. For example, it is checked whether the hemoglobin level is within the safe range for donating blood

1-

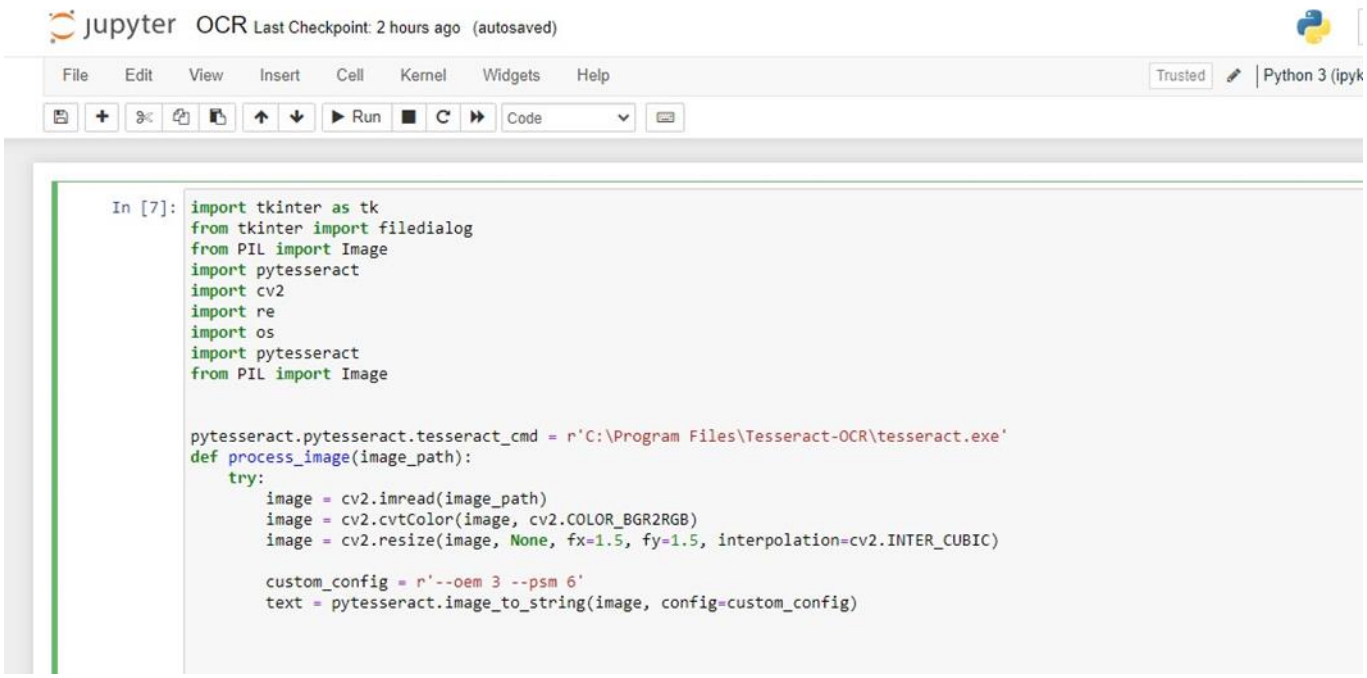


Fig 15: OCR code (1)

1. **Importing Necessary Libraries**:
   - tkinter and filedialog: Used for creating a graphical user interface for opening file dialogs.
   - PIL (from Pillow library): Used for image processing.
   - pytesseract: Python interface for Tesseract OCR.
   - cv2 (from OpenCV library): Used for image processing (reading, color conversion, resizing).
   - re: Imports regular expressions (not used in the current code).
   - os: Imports operating system functionalities (not used in the current code).

2. Specifying the Tesseract Path:
   - pytesseract.pytesseract.tesseract_cmd = r'C:\Program Files\Tesseract-OCR\tesseract.exe' Specifies the path to the Tesseract executable on the system.

3. **Function process_image:**

- **Input**: image_path (path to the image).

- **Image Processing Steps**:
  - image = cv2.imread(image_path): Reads the image from the specified path.
  - image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB): Converts the image from BGR color space to RGB.
  - Image=cv2.resize(image,None,fx=1.5,fy=1.5,interpolation=cv2.INTER_CUBIC): Resizes the image by increasing its dimensions by 1.5 times using cubic interpolation.

- **Configuring Tesseract**:
  - custom_config = r'--oem 3 --psm 6': Custom configuration for Tesseract, where:
    - --oem 3: Uses the LSTM OCR engine.
    - --psm 6: Treats the image as a single uniform block of text.

- **Extracting Text**:
  - text = pytesseract.image_to_string(image, config=custom_config):
  - Extracts text from the image using Tesseract with the specified configuration.

- **Error Handling**:
  - If an error occurs during image processing, it is printed and None is returned.

These are the main steps in the code. It reads an image, processe it to extract text using Tesseract OCR, and returns the extracted text.

2-

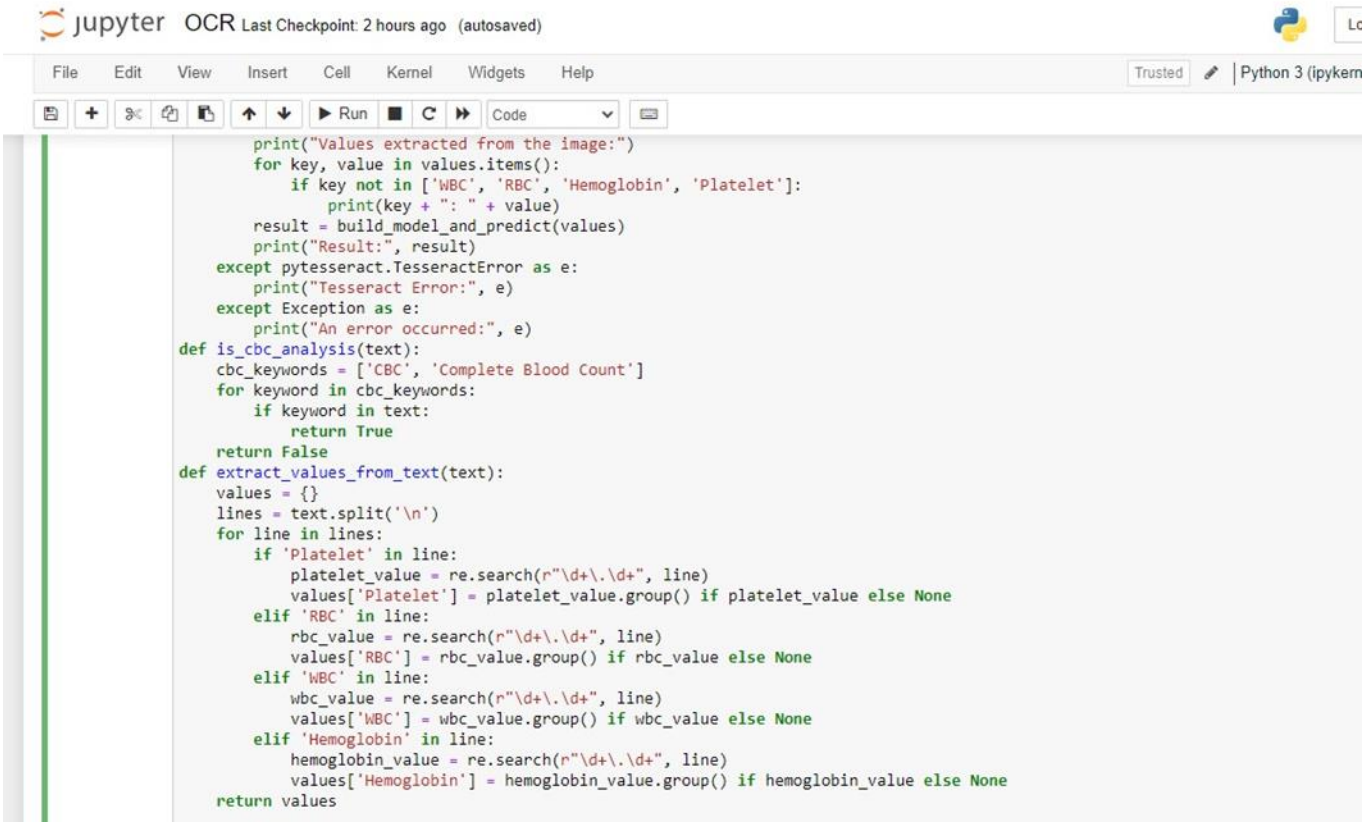


Fig 16: OCR code (2)

The code shown in the image demonstrates how to extract values from an image containing a CBC analysis.

A CBC analysis is a blood test that measures the number of white blood cells, red blood cells, platelets, and hemoglobin

- is_cbc_analysis(text): This function checks if the text contains a CBC analysis.
- extract_values_from_text(text): This function extracts the values from the text.
- build_model_and_predict(values): This function builds a model that predicts the outcome of the
- CBC analysis.

Statements:

- print("Values extracted from the image:"): This statement prints the message "Values extracted from the image:".
- for key, value in values.items():: This loop iterates over all the values in the dictionary values.
- if key not in ['WBC', 'RBC", "Hemoglobin', 'Platelet']:: This condition checks if the key is not present in the list ['WBC', 'RBC", "Hemoglobin', 'Platelet'].
- print(key + ":" + value): This statement prints the key and value.
- result = build_model_and_predict(values): This line stores the result from the build_model_and_predict function.
- print("Result:", result): This statement prints the result.
- except pytesseract.TesseractError as e:: This exception catches a Tesseract error.print("Tesseract Error:", e): This statement prints the Tesseract error.
- except Exception as e:: This exception catches any other error.
- print("An error occurred:", e): This statement prints any other error.

Function Explanations

- **is_cbc_analysis(text)**

  This function checks if the text contains a CBC analysis. To do this, the function searches keywords are found, the function returns True. Otherwise, it returns False.

- **extract_values_from_text(text)**

  This function extracts the values from the text. To do this, the function first splits the text into lines. Then,

  the function iterates over each line and searches for specific patterns. For instance, the function searches for

  the pattern "\d+.\d+" to extract numeric values. If a value is found, the function stores it in the dictionary values.

3-

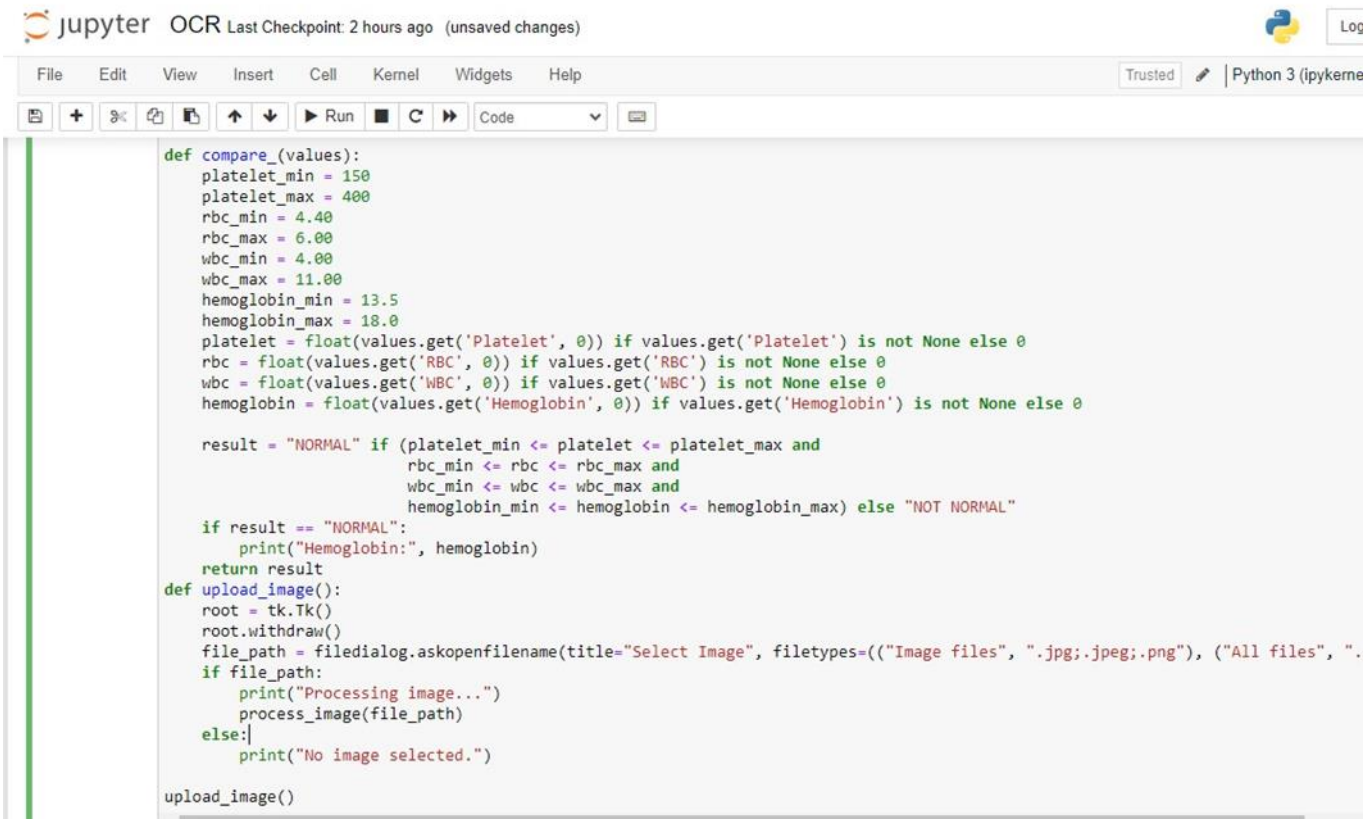


Fig 17: OCR code (3)

Extracts values from a text-based CBC analysis report (likely from a screenshot).

Potentially predicts the results of the CBC analysis (although the prediction part is not fully shown in the image).

Functions:

is_cbc_analysis(text): This function likely checks if the provided text contains

keywords or phrases indicative of a CBC analysis report.

extract_values_from_text(text): This function extracts numerical values from the text,

possibly corresponding to white blood cell (WBC) count, red blood cell (RBC) count, platelets, and hemoglobin levels.

build_model_and_predict(values) (not fully shown): This function seems to build a model

(machine learning) based on reference data and then uses it to predict the outcome (normal or abnormal) based on the extracted values.

Extracting and Printing Values:

The code iterates through a dictionary named values (which likely stores the extracted CBC values) and prints each key-value pair.

Error Handling:

The code includes error handling for potential exceptions that might occur during text extraction using

Pytesseract (an OCR library) or other parts of the code.

4- Results

1) The normal result

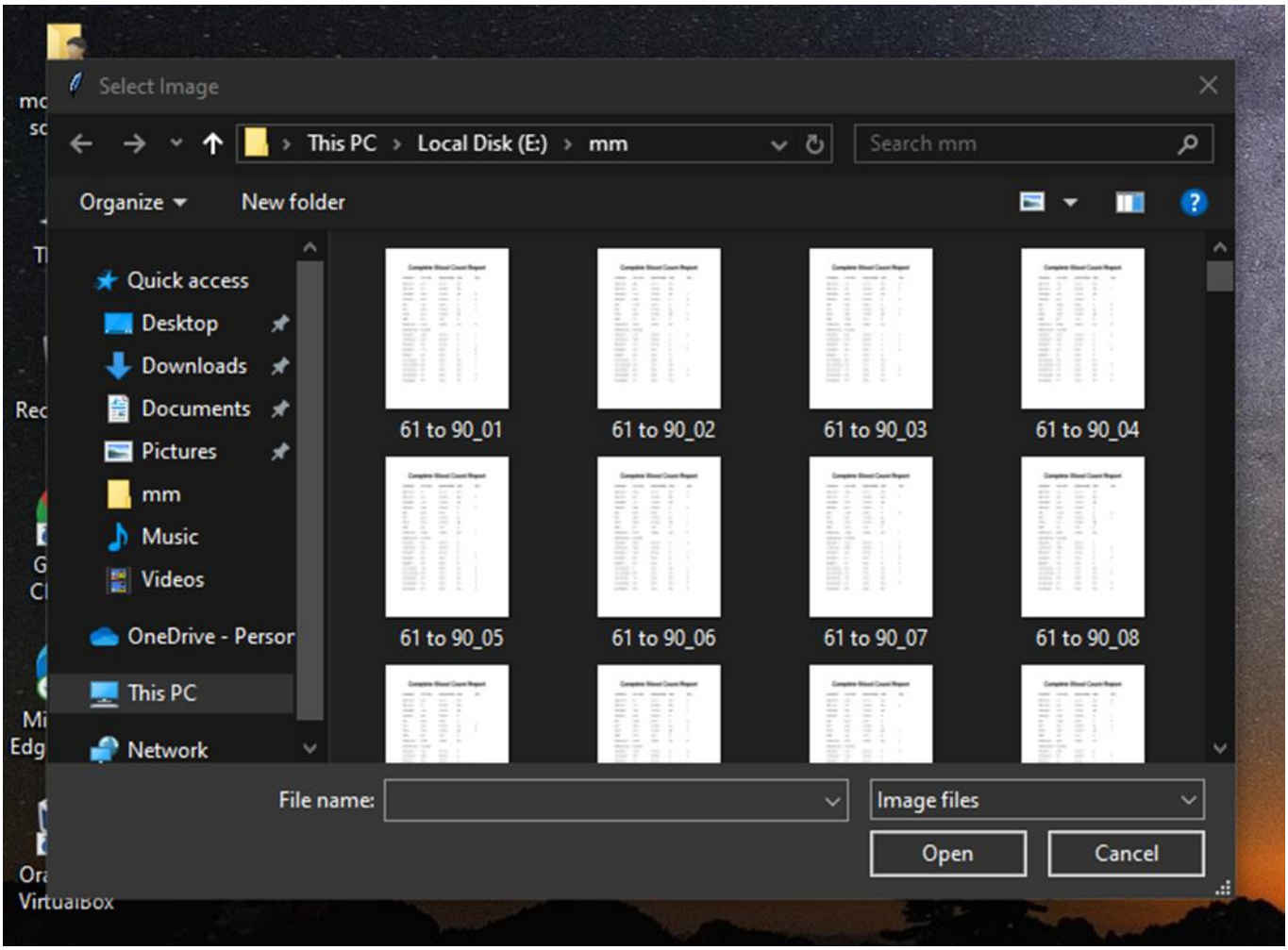


Fig 18: choose the CBC image

Processing the Result

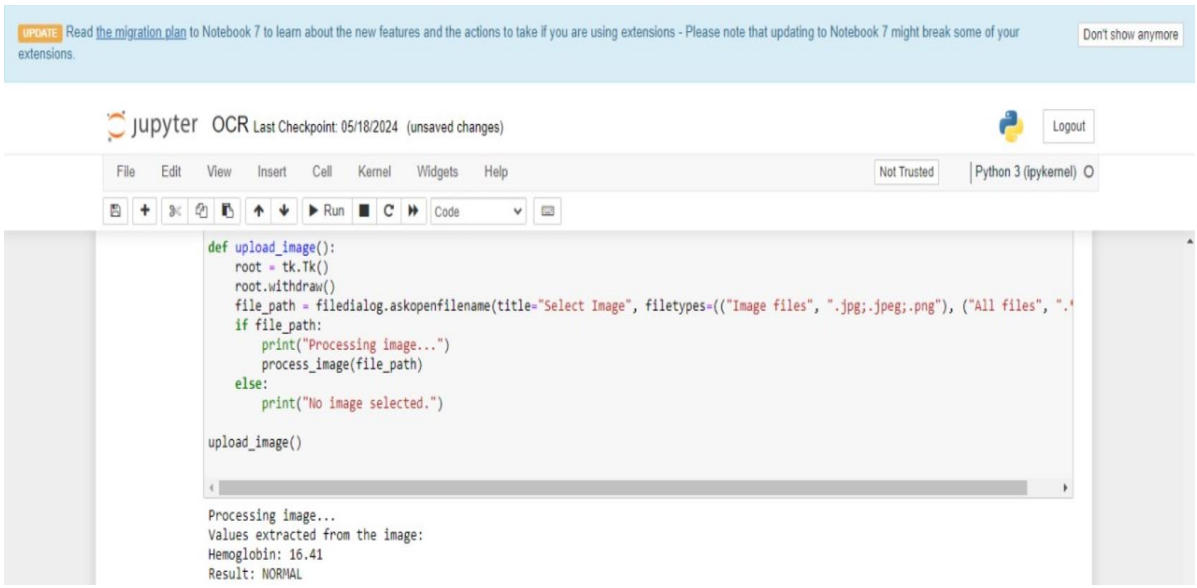


Fig 19: Processing the Result (1)

This is the normal result we back the hemoglobin value to continue in the first model.

2- In the abnormal

We will upload image and see the result

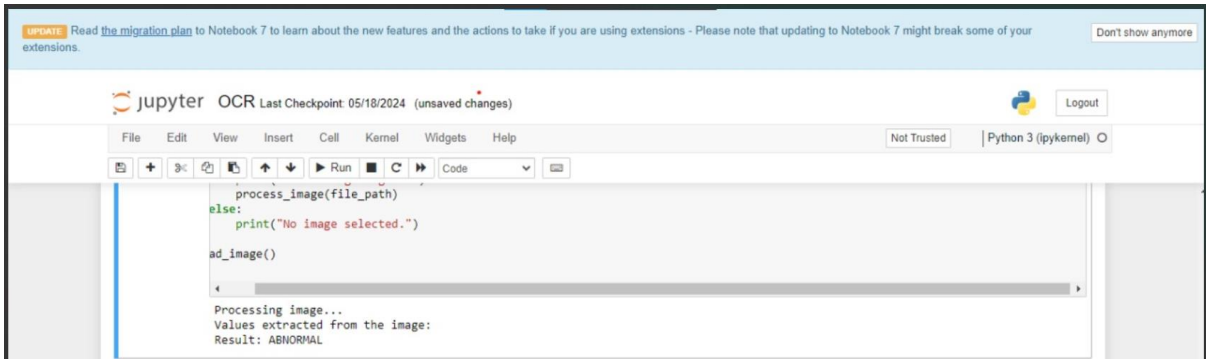


Fig 20: Processing the Result (2)

In this we can't back any value of the hemoglobin because in this issue the first model will print the user can't donate

4.10 chatbot

- After determining whether the user is eligible to donate, if they are not eligible, the application will redirect them to the chatbot. The user can express their feelings and concerns, and the chatbot will assist them in identifying their health issues and recommend appropriate actions or precautions. [9]

- Using natural language processing (NLP) enhances the communication between the user and the chatbot in our application. It allows the chatbot to understand and interpret user input more effectively, leading to a more natural and seamless interaction experience.

Natural Language Processing (NLP):

- is a subfield of artificial intelligence (AI) that focuses on the interaction between computers and humans through natural language. The goal of NLP is to enable computers to understand, interpret, and generate human language in a way that is both meaningful and useful.[10]

➢ First we will apply the NLP in the first dataset (for disease)

```
df['label'] = df['label'].str.lower().str.strip()
df['text'] = df['text'].str.lower().str.strip()
```

```
import string
df['label'] = df['label'].apply(lambda x: x.translate(str.maketrans('', '', string.punctuation)))
df['text'] = df['text'].apply(lambda x: x.translate(str.maketrans('', '', string.punctuation)))
```

```
from nltk.tokenize import word_tokenize
df['text'] = df['text'].apply(word_tokenize)
```

```
from nltk.corpus import stopwords
```

```
stop_words = set(stopwords.words('english'))
df['text'] = df['text'].apply(lambda x: [word for word in x if word not in stop_words])
```

```
from nltk.stem import WordNetLemmatizer
```

```
lemmatizer = WordNetLemmatizer()
df['text'] = df['text'].apply(lambda x: [lemmatizer.lemmatize(word) for word in x])
```

Fig 21: NLP

Convert all text in the 'label' and 'text' columns to lowercase and remove leading/trailing whitespace.

```
df['label'] = df['label'].str.lower().str.strip()
df['text'] = df['text'].str.lower().str.strip()
```

Fig 22: Str.lower

str.lower(): Converts the text to lowercase to ensure uniformity.
str.strip(): Removes any leading or trailing whitespace
Remove punctuation from the text in the 'label' and 'text' columns.

```
import string
df['label'] = df['label'].apply(lambda x: x.translate(str.maketrans('', '', string.punctuation)))
df['text'] = df['text'].apply(lambda x: x.translate(str.maketrans('', '', string.punctuation)))
```

Fig 23: String.punctuation

string.punctuation: A string containing all punctuation characters.
str.maketrans('', '', string.punctuation): Creates a translation table that maps each punctuation character to **None**.
translate(...): Applies the translation table to remove punctuation.

i Tokenize the text in the 'text' column into individual words

```
from nltk.tokenize import word_tokenize
df['text'] = df['text'].apply(word_tokenize)
```

Fig 24: Word_tokenize

word_tokenize: A function from the NLTK library that splits a string into a list of words (tokens).
Remove stopwords from the tokenized text in the 'text' column.

```
from nltk.corpus import stopwords

stop_words = set(stopwords.words('english'))
df['text'] = df['text'].apply(lambda x: [word for word in x if word not in stop_words])
```

Fig 25: Stop_words

stopwords.words('english'): Retrieves a list of common English stopwords from the NLTK library. set(...): Converts the list of stopwords to a set for faster lookup.
lambda x: [word for word in x if word not in stop_words]: A lambda function that filters out stopwords from the list of tokens.

Lemmatize the words in the tokenized text in the 'text' column

```
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
df['text'] = df['text'].apply(lambda x: [lemmatizer.lemmatize(word) for word in x])
```

Fig 26: Lemmatizer

WordNetLemmatizer(): Initializes a lemmatizer from the NLTK library.
lambda x: [lemmatizer.lemmatize(word) for word in x]: A lambda function that applies lemmatization to each word in the list of tokens, converting them to their base or dictionary form.

Filter out misspelled words from the tokenized text in the 'text' column.

```
import enchant

spell_checker = enchant.Dict("en_US")
df['text'] = df['text'].apply(lambda x: [word for word in x if spell_checker.check(word)])
```

Fig 27: Enchant

enchant.Dict("en_US"): Initializes a spell checker for US English using the Enchant library.
lambda x: [word for word in x if spell_checker.check(word)]: A lambda function that filters out words that are not recognized as correctly spelled by the spell checker.

Convert the preprocessed text data into numerical features using TF-IDF vectorization.

```
from sklearn.feature_extraction.text import TfidfVectorizer

tfidf_vectorizer = TfidfVectorizer(max_features=1000)
X = tfidf_vectorizer.fit_transform(df['text'].astype(str))
y = df['label']
```

Fig 28: TfidfVectorizer

TfidfVectorizer(max_features=1000): Initializes a TF-IDF vectorizer that converts a collection of raw
documents to a matrix of TF-IDF features, considering only the top 1000 features based on term frequency.
fit_transform(df['text'].astype(str)): Fits the TF-IDF vectorizer to the text data and transforms the
text into a TF-IDF feature matrix. The **astype(str)** method ensures that the text data is in string format.
X: The resulting TF-IDF feature matrix, representing the numerical features of the text data.
y: The 'label' column, which contains the target labels for the text data.

Convert categorical labels into numerical values

```
from sklearn.preprocessing import LabelEncoder

label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)
```

Fig 29: Str.lower

LabelEncoder(): Initializes a label encoder from the scikit-learn library.
fit_transform(y): Fits the label encoder to the target labels (**y**) and transforms them into numerical values.
y_encoded: The resulting array of encoded labels, where each unique label in **y** is assigned a unique integer
value.

➢ **second we apply the NLP in second dataset (for precaution)**

1- Convert all text data in the specified columns to lowercase and remove leading/trailing whitespace.

```
df_second['Disease'] = df_second['Disease'].str.lower().str.strip()
df_second['Precaution_1'] = df_second['Precaution_1'].str.lower().str.strip()
df_second['Precaution_2'] = df_second['Precaution_2'].str.lower().str.strip()
df_second['Precaution_3'] = df_second['Precaution_3'].str.lower().str.strip()
df_second['Precaution_4'] = df_second['Precaution_4'].str.lower().str.strip()
```

Fig 30: String.punctuation

str.lower(): Converts the text to lowercase to ensure uniformity across the dataset.
str.strip(): Removes any leading or trailing whitespace to clean up the text data.

2- Remove punctuation from all the text data in the specified columns

```
df_second['Disease'] = df_second['Disease'].apply(lambda x: x.translate(str.maketrans('', '', string.punctuation)))
df_second['Precaution_1'] = df_second['Precaution_1'].apply(lambda x: x.translate(str.maketrans('', '', string.punctuation)))
df_second['Precaution_2'] = df_second['Precaution_2'].apply(lambda x: x.translate(str.maketrans('', '', string.punctuation)))

df_second['Precaution_3'] = df_second['Precaution_3'].apply(lambda x: x.translate(str.maketrans('', '', string.punctuation)))
df_second['Precaution_4'] = df_second['Precaution_4'].apply(lambda x: x.translate(str.maketrans('', '', string.punctuation)))
```

Figu 31: Neural network model

string.punctuation: A string containing all punctuation characters.
str.maketrans('', '', string.punctuation): Creates a translation table that maps each punctuation character to **None**.
translate(...): Applies the translation table to remove punctuation.

Integration with previous Dataset ;This dataset complements the symptom description dataset by offering practical steps that individuals can take after recognizing the symptoms and being diagnosed with a particular disease. When combined, these datasets can be used to build comprehensive health advisory systems that not only identify potential diseases based on symptoms but also suggest appropriate actions to manage and treat the conditions effectively.

So We build simple neural network model to train our dataset

```
from sklearn.model_selection import train_test_split
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.2, random_state=42)

model = Sequential([
    Dense(128, activation='relu', input_shape=(X_train.shape[1],)),
    Dense(64, activation='relu'),
    Dense(len(label_encoder.classes_), activation='softmax')
])


model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])


model.fit(X_train, y_train, epochs=10, batch_size=32, validation_split=0.1)


test_loss, test_accuracy = model.evaluate(X_test, y_test)
print(f"Test Accuracy: {test_accuracy}")
```

Fig 32: chatbot response

1) **Import Necessary Libraries**:
   - train_test_split: Splits the data into training and testing sets.
   - Sequential: Initializes a linear stack of layers.
   - Dense: Fully connected layer in a neural network.

2) **Split the Data**:
   - X and y_encoded are the feature and label datasets, respectively.
   - train_test_split: Splits the data into training (80%) and testing (20%) sets.
   - random_state=42: Ensures reproducibility of the split.

3) **Define the Model**:

- Sequential: Creates a sequential model.
- Dense: Adds fully connected layers to the model.
- The first layer has 128 units and uses the ReLU activation function. It also specifies the input shape.
- The second layer has 64 units and uses the ReLU activation function.
- The output layer has units equal to the number of classes in the target variable and uses the softmax activation function for multi-class classification.

4) **Compile the Model**:

- optimizer='adam': Uses the Adam optimizer.
- loss='sparse_categorical_crossentropy': Uses sparse categorical cross-entropy loss function.
- metrics=['accuracy']: Tracks accuracy during training and evaluation.

5) **Train the Model**:

- epochs=10: Trains the model for 10 epochs.
- batch_size=32: Uses a batch size of 32.
- validation_split=0.1: Uses 10% of the training data for validation during training

1. Function: chatbot response

```
from sklearn.metrics.pairwise import cosine_similarity
def chatbot_response(user_input):
    user_input = user_input.lower().strip()

    if user_input == "hi" or user_input == "hello":
        return "Hello! How can I assist you today?"

    elif user_input == "bye" or user_input == "goodbye":
        return "Goodbye! Have a great day!"

    elif user_input == "how are you":
        return "I'm just a computer program, so I don't have feelings, but I'm here to assist you!"

    elif user_input == "thank you" or user_input == "thanks":
        return "You're welcome! If you have any more questions, feel free to ask."

    else:

        user_input_vector = tfidf_vectorizer.transform([user_input])
        similarity_scores = cosine_similarity(user_input_vector, X)
        matched_index = np.argmax(similarity_scores)
```

Fig 33: Response Generation

- chatbot using the **scikit-learn** library for text processing and machine learning. The chatbot interacts with users, takes their input, matches it to potential diseases, and provides relevant actions/precautions if requested. Here's a detailed breakdown of how the code works:

- We use **cosine similarity** from **sklearn.metrics.pairwise** to measure the similarity between the user input and the disease descriptions.

- This function processes the user's input and generates an appropriate response. It handles various user inputs, including greetings, farewells, and queries about the chatbot's status.

- **Standard Responses:** The function checks if the input is a common greeting ("hi", "hello"), a farewell ("bye", "goodbye"), or a courtesy ("thank you", "thanks").

- **Disease Matching:** For other inputs, the function transforms the user input into a vector using a pre-fitted **tfidf_vectorizer**, then computes cosine similarity scores between the user input and a dataset (X) of disease descriptions.

```
if similarity_scores[0][matched_index] > 0.5:
    matched_disease = y[matched_index]


    actions_available = matched_disease in df_second['Disease'].values
    if actions_available:
        return f"The matched disease for your description is: {matched_disease}.
        Do you want to know what actions/precautions you should take? (yes/no)"

    else:
        return f"The matched disease for your description is: {matched_disease}.
        However, no actions/precautions are available for this disease. Do you want to know what actions/precautions you should take?

else:
    return "Sorry, the disease corresponding to your description was not found s."
```

Figu 34: Interaction(1)

Response Generation: If a matching disease is found with a similarity score above a threshold (0.5), the function returns the matched disease and asks if the user wants to know the precautions. If no match is found, it informs the user.

4.11 Chatbot Interaction

```
response = chatbot_response(user_input)
print("Chatbot:", response)

if "actions/precautions" in response.lower():
    user_response = input("You: ")

    if user_response.lower().strip() == "yes":

        matched_row = df_second[df_second['Disease'] == matched_disease]



        else:
            print("No actions or precautions found for", matched_disease)

    elif user_response.lower().strip() == "no":
        print("Chatbot: Okay, if you have any other questions, feel free to ask.")

    else:
        print("Chatbot: I'm sorry, I didn't understand that. Please answer with 'yes' or 'no'.")
```

Figure 35: Interaction(2)

- The main interaction loop continuously prompts the user for input until the user types "bye".
- **Action Inquiry:** If the initial response indicates the presence of disease information, it asks the user if they want to know the actions/precautions.
- **Providing Precautions:** If the user says "yes",
  it fetches and displays the relevant precautions from the dataset. If the user says "no", the conversation can continue or end based on user input.
- **Exiting the Chat:** If the user types "bye", the chatbot exits the loop and prints a goodbye message.

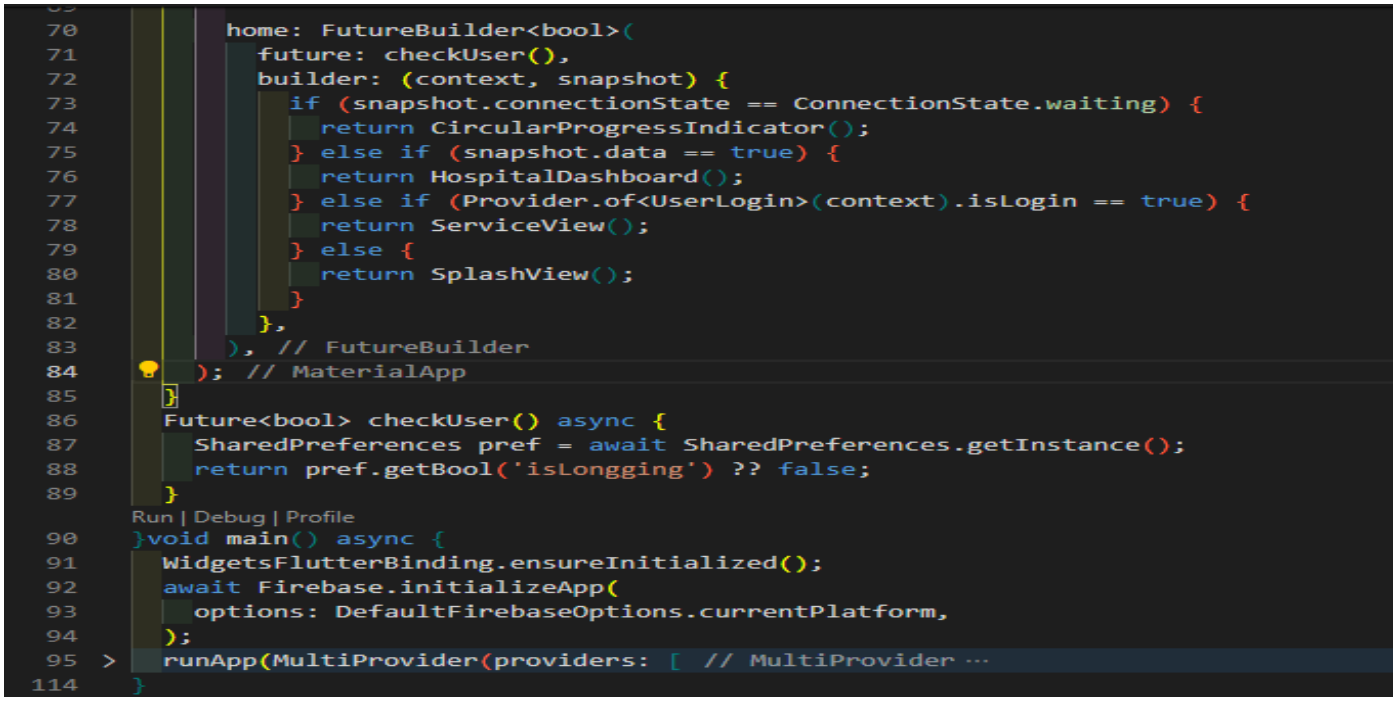
4.12 Flutter Application

Fig 36: Intro to Flutter

- Upon launching the app, a new user is welcomed by a colorful and engaging onboarding experience. This introductory process is designed to make a positive first impression and set the tone for the user's journey with the app. The vibrant visuals and interactive elements aim to captivate the user's attention, making them feel excited and eager to explore further.[3]

- During this onboarding phase, it is crucial to communicate the app's core values and mission clearly and effectively. This helps new users understand what the app stands for, its purpose, and how it can benefit them. By highlighting these aspects, we ensure that users grasp the app's unique selling points and feel aligned with its goals from the very beginning.

- The onboarding messages should be crafted thoughtfully to convey the app's vision and objectives. They should explain the fundamental principles that guide the app's development and usage, such as innovation, user-centric design, and commitment to quality. Additionally, the mission statement should outline the broader impact the app aims to achieve, whether it's enhancing productivity, fostering community, or providing entertainment.



```
70        home: FutureBuilder<bool>(
71          future: checkUser(),
72          builder: (context, snapshot) {
73            if (snapshot.connectionState == ConnectionState.waiting) {
74              return CircularProgressIndicator();
75            } else if (snapshot.data == true) {
76              return HospitalDashboard();
77            } else if (Provider.of<UserLogin>(context).isLogin == true) {
78              return ServiceView();
79            } else {
80              return SplashView();
81            }
82          },
83        ), // FutureBuilder
84      ); // MaterialApp
85    }
86    Future<bool> checkUser() async {
87      SharedPreferences pref = await SharedPreferences.getInstance();
88      return pref.getBool('isLongging') ?? false;
89    }
      Run | Debug | Profile
90    }void main() async {
91      WidgetsFlutterBinding.ensureInitialized();
92      await Firebase.initializeApp(
93        options: DefaultFirebaseOptions.currentPlatform,
94      );
95  >   runApp(MultiProvider(providers: [ // MultiProvider ···
114   }
```

Fig 37: Check user Login

- This code is part of a Flutter application that manages user navigation based on their login status. Here's

  a detailed breakdown of what each part does:

  The app starts and the FutureBuilder initiates the checkUser() function to determine the user's login status.
  While checkUser() is running, a loading indicator is displayed.
  Once checkUser() completes:

  If the user is logged in (checkUser() returns true), the HospitalDashboard is shown.
  If the user is not logged in (checkUser() returns false), the app checks the isLogin status from the UserLogin provider.
  If isLogin is true, ServiceView is shown.
  If isLogin is false, SplashView is shown.

Assumptions:

HospitalDashboard, ServiceView, and SplashView are Flutter widgets representing different parts of the app.
Provider.of<UserLogin>(context).isLogin accesses the UserLogin provider to check if the user is currently logged in.

SharedPreferences is used for storing simple data persistently across app launches.

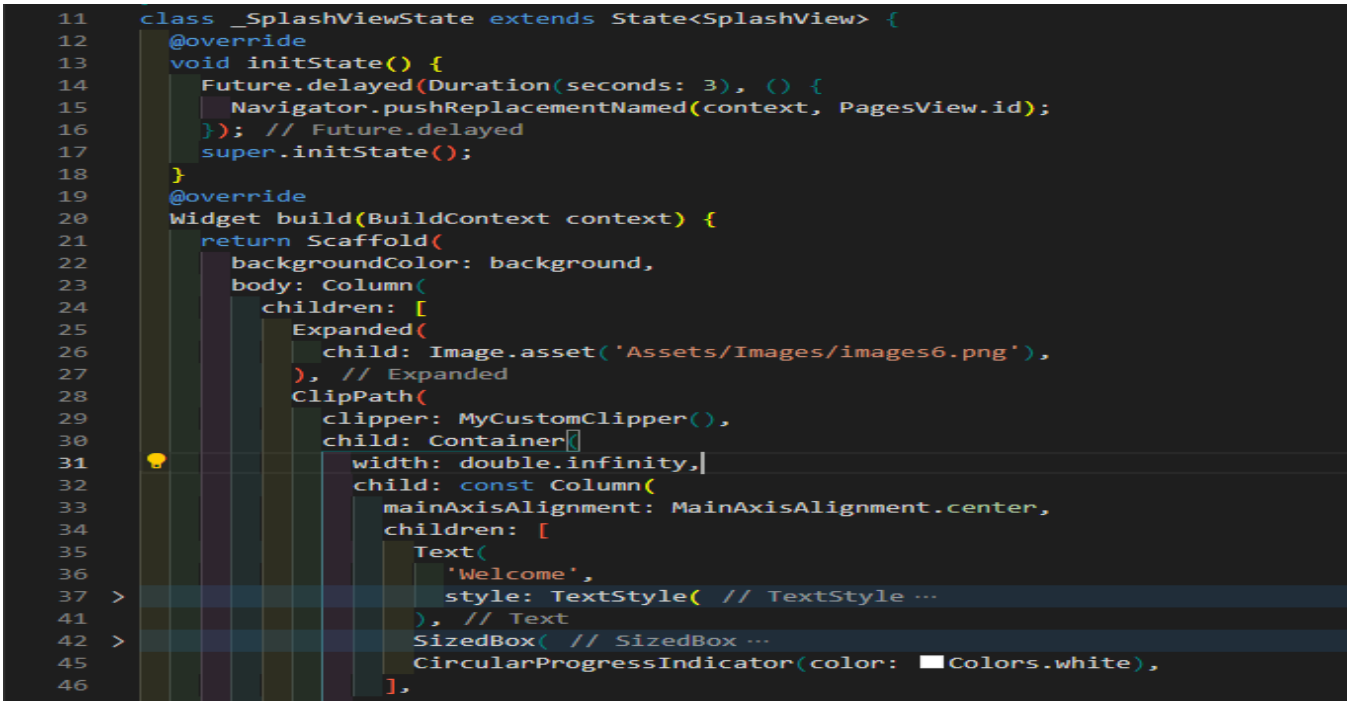


Fig 38: Splash screen

SplashView Widget

Definition

SplashView extends StatefulWidget, which means it has a mutable state that can change over time.
It has a static string id for easy reference, likely used for navigation within the app.

initState Method

initState is called once when the state is initialized.
Inside initState, a delay is introduced using Future.delayed(Duration(seconds: 3), () { ... }).
This waits for 3 seconds before executing the provided callback function.
After 3 seconds, Navigator.pushReplacementNamed(context, PagesView.id) is called.
This replaces the current route with the route identified by PagesView.id, navigating to a new screen.

4.13 Application Authentication

Register View

- handles user registration, including form fields for email and password, form validation to ensure all required fields are filled,
- And feedback mechanisms like snack bars to alert users about errors such as an email already in use or a weak password. [2]

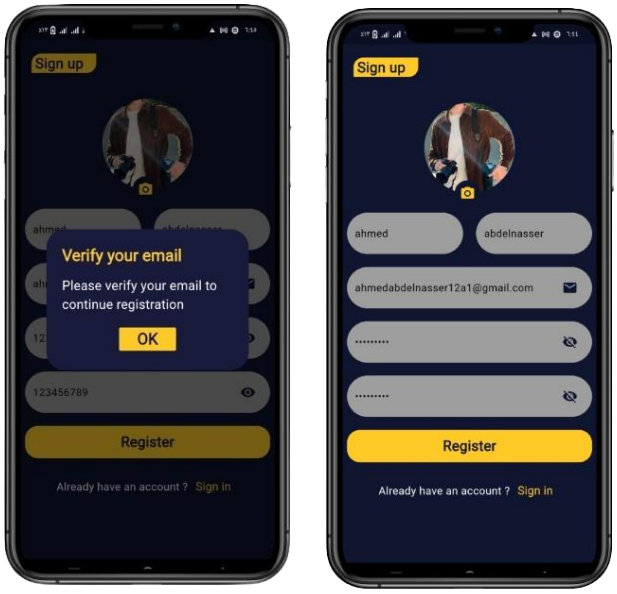


Fig 39: Register Screen

- After successful registration and email verification, the user is navigated to the login view. This ensures a smooth registration process with clear feedback and validation for the user.

```dart
main.dart 2 ● | Splash_Screen.dart ● | RegisterView.dart ●
lib > view > RegisterView.dart > _RegisterViewState > addUser
class _RegisterViewState extends State<RegisterView> {

  Future<void> addUser() {
    return users
        .add({
          'frist_name': '${fristName}',
          'last_name': '${lastName}',
          'url': url,
          'email': email,
          'bloodType': _textEditingController.text,
          'id': FirebaseAuth.instance.currentUser!.uid,
        })
        .then((value) => print("User Added"))
        .catchError((error) => print("Failed to add user: $error"));
  }
  Future<void> _registerWithEmailAndPassword(
      String email, String password) async {
    try {
      final credential =
          await FirebaseAuth.instance.createUserWithEmailAndPassword(
        email: email,
        password: password,
      );
      addUser();
      await credential.user!.sendEmailVerification();
    } on FirebaseAuthException catch (e) {
      if (e.code == 'weak-password') {
        await Future.delayed(Duration(seconds: 2));
        ScaffoldMessenger.of(context as BuildContext).showSnackBar(···
      } else if (e.code == 'email-already-in-use') {
        await Future.delayed(Duration(seconds: 2));
        ScaffoldMessenger.of(context as BuildContext).showSnackBar(···
```

Fig 40: Add user & register Function

addUser()

This function adds user data to a database, presumably Firestore.
It adds a new document to the 'users' collection with fields such as first name, last name, URL, email,
blood type, and user ID.
It returns a `Future<void>` to indicate completion, which is resolved when the data is successfully added.
If an error occurs during the addition process, it catches the error and prints a message indicating the failure.

registerWithEmailAndPassword(String email, String password)
This function attempts to register a new user with the provided email and password using Firebase Authentication.
It uses `createUserWithEmailAndPassword()` method provided by FirebaseAuth to register the user.
Upon successful registration, it calls `addUser()` to add additional user data to the database.
After registering the user, it sends an email verification request to the user.
It handles potential errors using a try-catch block:

If the password provided is considered weak, it displays a snack bar message indicating that the password is too weak.
If the email is already in use, it displays a snack bar message indicating that the email already exists.
Other potential errors are caught and printed.

4.14 Firebase Setup

Creating a Firebase Project:
We set up the Firebase project in the Firebase console.
We configure project settings according to our app's requirements.
Adding Our Flutter App to the Firebase Project:
We register the Flutter app with the Firebase project in the console.
We follow prompts to integrate the app with Firebase.
Adding Firebase SDK to Our Flutter Project:
We edit the pubspec.yaml file to include Firebase dependencies.
We run flutter pub get to install the dependencies.[4]

```
dependencies:
  firebase_core: ^1.0.0
  firebase_auth: ^3.0.0
  cloud_firestore: ^3.1.0
```

Fig 41: Firebase packages

Initializing Firebase in Our Flutter App:
We import Firebase packages in the app's Dart files.
We initialize Firebase in the app, typically in the main() function or initState().

```
import 'package:firebase_core/firebase_core.dart';

void main() async {
  WidgetsFlutterBinding.ensureInitialized();
  await Firebase.initializeApp();
  runApp(MyApp());
}
```

Fig 42: main function

we turned to Firebase Authentication, and it was a game-changer for us. First off, setting it up was a breeze.
We went to the Firebase Console, enabled the authentication methods we wanted to use (like email/password and Google sign-in), and that was it

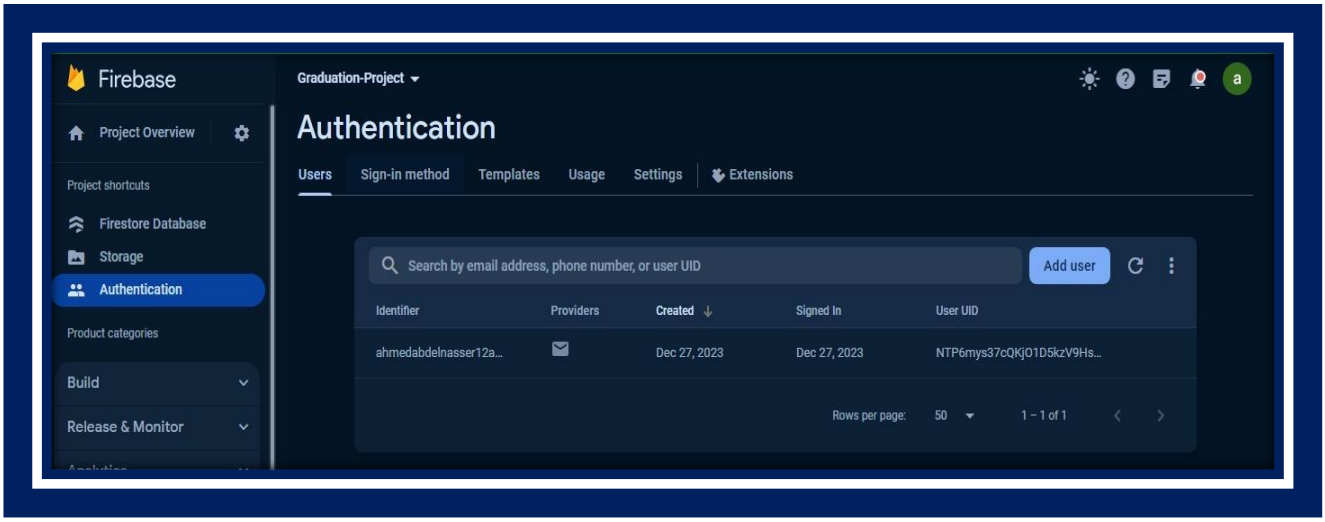


Fig 43: Firebase authentication

We go to the Firestore Database section in the Firebase Console.
We decide to create a collection named users.
Each user will have a unique document within this collection.
The document will have fields such as Name, Email, id, and url_image

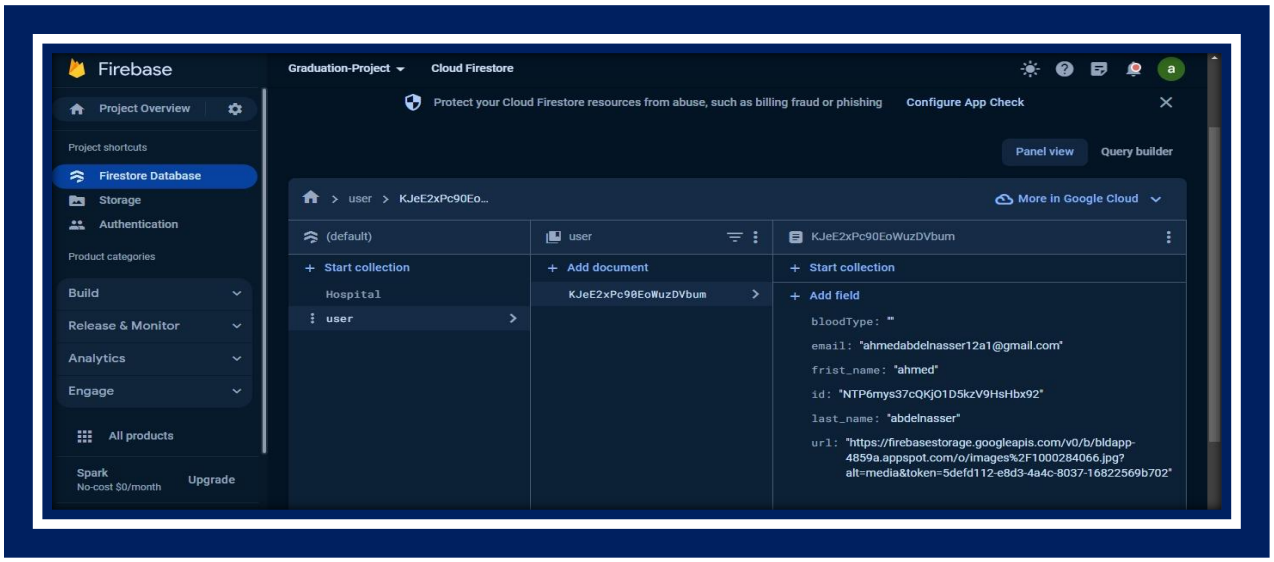


Fig 44: Firebase database (user)

While opening service view automatically appears location permission which required to access user location to use application services.

4.15 User Service

Donate Blood:
The app needs to know the user's location to suggest nearby blood donation centers.
It helps in scheduling appointments and providing directions to the closest donation centers.
Find Blood Type:
Users searching for specific blood types need to find donors or blood banks near their location.
The app can match users with nearby donors or resources to ensure timely assistance.

Fig 45: User Service

Location Permission

```dart
Future<Position> determinePosition() async {
  bool serviceEnabled;
  LocationPermission permission;

  serviceEnabled = await Geolocator.isLocationServiceEnabled();
  if (!serviceEnabled) {}

  permission = await Geolocator.checkPermission();
  permission = await Geolocator.requestPermission();

  if (permission == LocationPermission.denied) {}

  if (permission == LocationPermission.deniedForever) {
    ScaffoldMessenger.of(context).showSnackBar(snackbar);
  }
  Position position = await Geolocator.getCurrentPosition(
      desiredAccuracy: LocationAccuracy.low);
  print(position);
  return await Geolocator.getCurrentPosition();
}
```

Fig 46: Location Permission

Define Function: Create determinePosition with a BuildContext parameter.
Check Location Services: Ensure location services are enabled; return error if not.
Check and Request Permissions:
Handle Denials:
Return error if permission is denied again.
Show SnackBar and return error if permanently denied.

Drawer :

Header Section:
Background image. Overlay text and profile picture.
Profile picture from URL or default placeholder.
User's name and email displayed below the picture.
Menu Items:
Blood Donate: Navigates to OCR view.
Find Blood Type: Navigates to search view, closes drawer.
Permissions: Opens app settings.
Theme Switch: Toggles light/dark theme.
Notifications: Navigates to notifications view.
Sign Out: Shows confirmation dialog, then signs out and redirects to login.
Close Drawer: Closes the drawer.

Fig 47: Drawer

Find Blood Type

- ❑ This service is provided by the application, which enables the user to find the blood type he is searching for.
- ❑ Service logic appears the hospitals which Join the application with providing its data according to the closest distance to the user in his area.
- ❑ After the user searches, it will appear List of three nearest hospitals and the amount of blood Type in each one

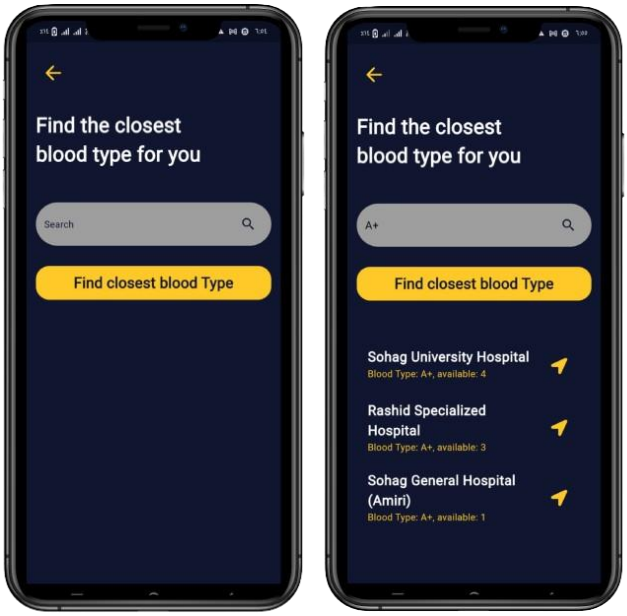


Fig 48: Find blood type

```
double calculateDistance(
  double startLatitude,
  double startLongitude,
  double endLatitude,
  double endLongitude,
) {
  const int earthRadius = 6371;

  double lat1 = startLatitude * (3.1415 / 180);
  double lon1 = startLongitude * (3.1415 / 180);
  double lat2 = endLatitude * (3.1415 / 180);
  double lon2 = endLongitude * (3.1415 / 180);

  double dlat = lat2 - lat1;
  double dlon = lon2 - lon1;

  double a =
      pow(sin(dlat / 2), 2) + cos(lat1) * cos(lat2) * pow(sin(dlon / 2), 2);

  double c = 2 * atan2(sqrt(a), sqrt(1 - a));

  double distance = earthRadius * c;
  return distance;
}
```

```
hospitals = allHospitals
    .where((hospital) =>
        hospital.bloodTypes.containsKey(searchBloodType) &&
        hospital.bloodTypes[searchBloodType]! > 0)
    .toList();

if (currentPosition != null) {
  hospitals.sort((a, b) {
    double distanceToA = calculateDistance(
      currentPosition!.latitude,
      currentPosition!.longitude,
      a.location.latitude,
      a.location.longitude,
    );
    double distanceToB = calculateDistance(
      currentPosition!.latitude,
      currentPosition!.longitude,
      b.location.latitude,
      b.location.longitude,
    );
    return distanceToA.compareTo(distanceToB);
  });
}

setState(() {});
}
```

Fig 49: Ascending hospital logic (1)

Service Logic:

Filter Hospitals:
The service first filters out hospitals that do not have the required blood type or have zero units of it.
This results in a list of hospitals that can potentially fulfill the user's need.
Sort by Proximity:
If the user's current location is known, the filtered hospitals are sorted by their distance to the user.

This ensures the user sees the nearest hospitals first.

Distance Calculation:
The distance between the user's location and each hospital is calculated using the Haversine formula.

This formula accounts for the spherical shape of the Earth, providing an accurate distance measurement in kilometers.

Distance Calculation Details:
Convert Coordinates:
The latitude and longitude of both the user and the hospital are converted from degrees to radians.

Calculate Differences:
The differences in latitude and longitude between the user and the hospital are computed.

Apply Haversine Formula:
The formula calculates the shortest path over the Earth's surface, giving the distance between two points.

User Interaction:
Search for Blood Type:
The user inputs the desired blood type.
The application then processes the request, filters the hospitals, sorts them by distance, and updates the display.

Display Results:
The application shows a list of the three nearest hospitals with the required blood type and the available amount at each location.

Alert Dialogs for Location Issues
Location Services Disabled.
Location Permission Denied.
,Uer writes a wrong blood type

Storing Hospital Data in Firestore Database .
Hospital Contact:
The hospital contacts the application, likely through phone ,whatssap ,linkedln

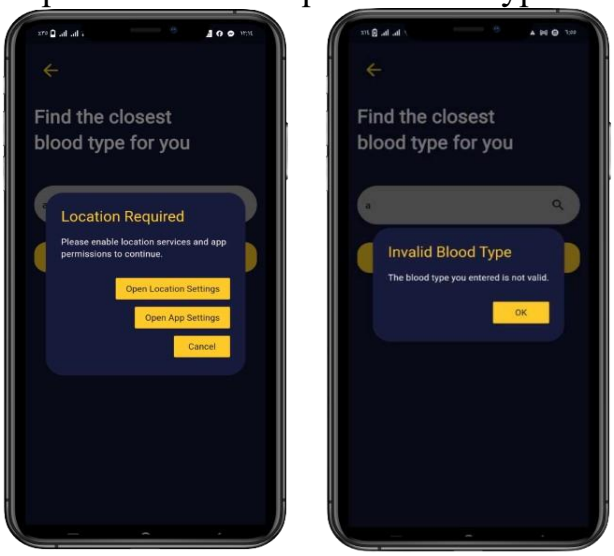


Fig 50: Find blood type permission

Data Collection:
The application collects necessary information from the hospital, such as:
Name of the hospital.
Image/logo of the hospital.
Location coordinates or address of the hospital.

Firestore Database:
The application uses Firestore, a NoSQL cloud database provided by Firebase, to store the hospital data.

Document Creation:
A new document is created in the Firestore database's "hospitals" collection for each hospital that contacts the application.

Data Storage:
Hospital information, such as name, image URL, and location details, is stored as fields within the document.
Firestore collection: "hospitals"
Document (for each hospital)
Field: "name" (hospital name)
Field: "image" (URL to hospital image/logo)
Field: "location" (latitude and longitude or address)

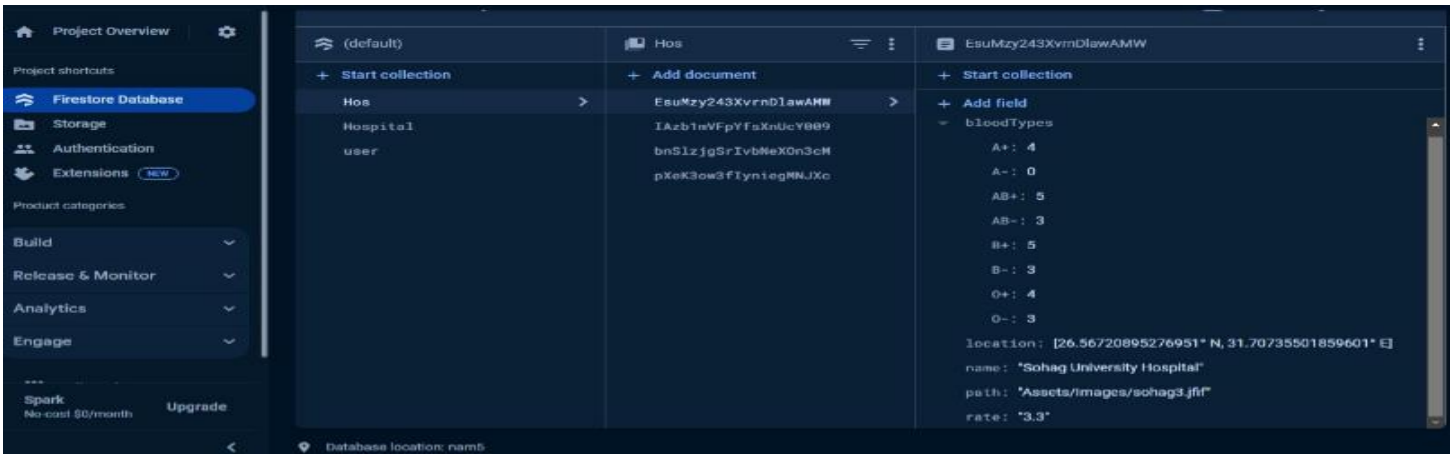


Fig 51: Firebase database (hospital)

Donate blood type
This service that we provide through the application is to facilitate the donation process and to enable the user to donate to the hospital that has the least amount of blood type.
Using Machine learning that determines the user's ability to donate or not
User enters some data based on medical analysis

4.16 CBC Donation Test

App simplifies donor eligibility check using CBC test results, guiding users through the donation process effectively.
using Ai (OCR)

Steps:
Upload OCR Image:
User uploads CBC test results via OCR image.

Analysis Dialog:
App analyzes results and displays eligibility status.

Feedback:
If eligible, user proceeds to fill out the donation form.
If not eligible, user is informed they may not donate at this time.

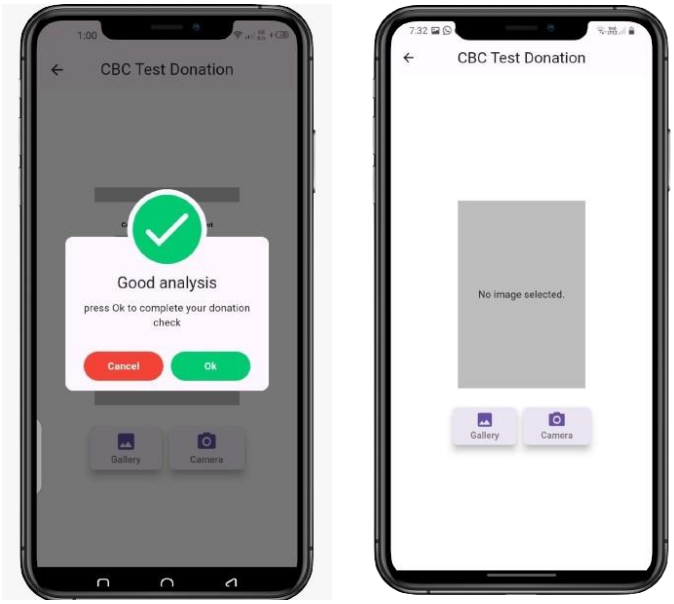


Fig 52: CBC analysis

```
Codeium: Refactor | Explain | Generate Function Comment | ×
Future<void> _uploadImage() async {
  if (_image == null) {
    setState(() {
      _result = 'Please select an image first';
    });
    return;
  }

  var url = Uri.parse('https://blood-ocr.onrender.com');
  var request = http.MultipartRequest('POST', url)
    ..files.add(await http.MultipartFile.fromPath('image', _image!.path));

  var streamedResponse = await request.send();
  var response = await http.Response.fromStream(streamedResponse);
  if (response.statusCode == 200) {
    var jsonResponse = json.decode(response.body);
    setState(() {
      _result = jsonResponse['result'];
    });
  } else {
    setState(() {
      _result = 'Failed to upload image';
    });
  }
}
```

Fig 53: upload image funcion

The function _uploadImage() uploads an image file to a server.
It checks if an image is selected; if not, it notifies the user to select one.

If an image is selected, it creates an HTTP POST request with the image attached.
The request is sent to a specific URL.
After receiving a response from the server:
If the response status is OK (200), it updates the UI with the result extracted from the response.
If the response status is not OK, it notifies the user of a failed upload.

<u>After verifying the integrity of the CBC analysis, it goes to a form that is created using artificial intelligence To ensure the safety of the donor person</u>

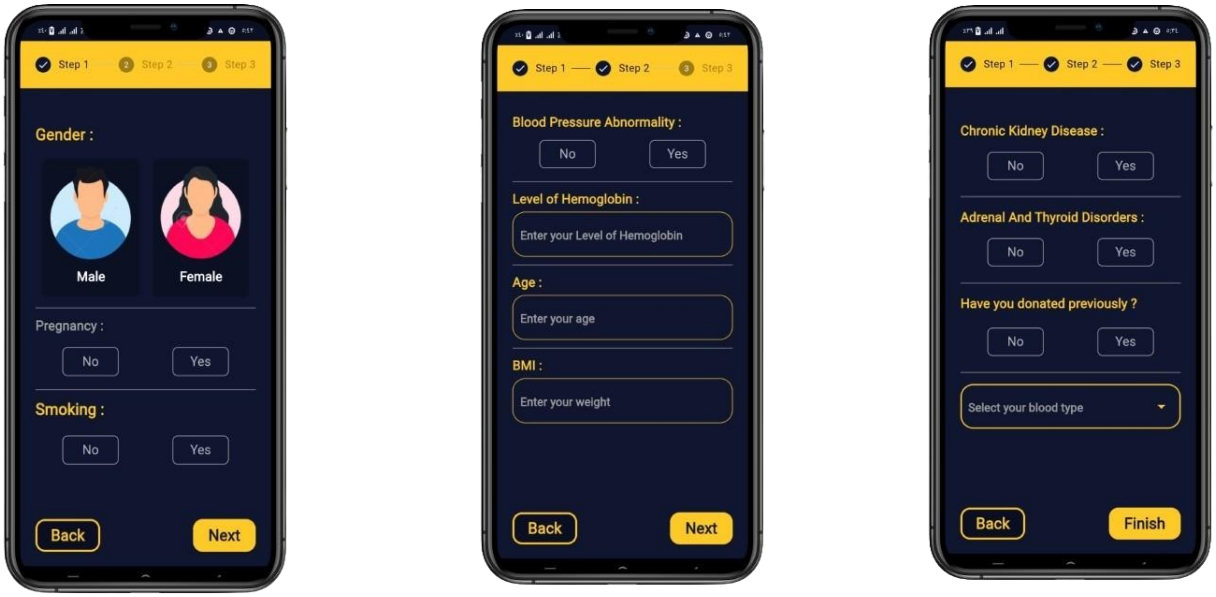


Fig 54: Donation screen

Gender:
Women who are pregnant may be temporarily ineligible to donate blood due to potential health risks associated with pregnancy.

Smoking:
Smoking does not automatically disqualify a person from donating blood. However, heavy smokers may have elevated levels of carbon monoxide in their blood, affecting the quality of the donated blood.

Blood Pressure:
Donors must have blood pressure within an acceptable range to donate safely.
High or low blood pressure may disqualify a person from donating.

Hemoglobin Level:
Hemoglobin is a protein in red blood cells that carries oxygen. Donors must have a minimum hemoglobin level to ensure they can safely donate blood without risk of anemia.

Age:
Donors must meet the minimum age requirement, typically 16 or 18 years old, depending on local regulations. There may also be an upper age limit for donation.

BMI (Body Mass Index):
Donors must have a healthy BMI within a specified range to donate blood.
A BMI that is too low or too high may disqualify a person from donation.

Chronic Kidney Disease:
Chronic kidney disease may affect a person's ability to donate blood due to potential complications or health risks.

Adrenal and Thyroid Disorders:
Certain adrenal and thyroid disorders may impact a person's eligibility to donate blood, depending on the severity and management of the condition.

Previous Donations:
Previous blood donations may affect eligibility, as frequent donations can lead to temporary or permanent deferral periods to allow the donor's body to replenish blood stores.

Blood Type:
Certain blood types are in higher demand for specific blood products. Donors of all blood types are needed, but some restrictions may apply based on the needs of blood banks and hospitals.

```
if (key.currentState!.validate()) {
  var provider = Provider.of<FormProvider>(context, listen: false);

  int i = await Services().makePrediction(
      bloodPressure: provider.selectPressure,
      levelHemoglobin: widget.HemoglobinLevel,
      age: age!,
      bmi: bmi!,
      gender: provider.selectedGender,
      pregancy: provider.selectedPregnancy,
      smoking: provider.selectedSmoking,
      chronicKidney: provider.selectedChronicKidneyDisease,
      adrenalAndThyroidDisorders:
          provider.selectedAdrenalAndThyroidDisorders,
      bloodType: _textEditingController.text);
```

Fig 55: Model prediction function

Form Validation: key.currentState!.validate() checks if the form fields associated with the Form widget, referenced by key, are valid. If they are, it proceeds with the next steps; otherwise, it halts execution.
Provider Usage: Provider.of<FormProvider>(context, listen: false) retrieves the instance of FormProvider
from the widget tree. This provider likely holds state information relevant to the form.
Service Invocation: Services().makePrediction(...) invokes a method named makePrediction from a class or object called Services. This method likely sends data to a backend service for processing.
Data Parameters: It passes various parameters to the makePrediction method :

bloodPressure: Likely the selected blood pressure value.
levelHemoglobin: Hemoglobin level (likely obtained from a widget property).
age: The age value.
BMI: Body Mass Index (BMI) value.
gender: Selected gender.
pregnancy: Pregnancy status.
smoking: Smoking status.
chronicKidney: Status of chronic kidney disease.
adrenalAndThyroidDisorders: Status of adrenal and thyroid disorders.
bloodType: Text obtained from a TextEditingController.

When user enter all field , this values passes to ai model to check the check donar ability

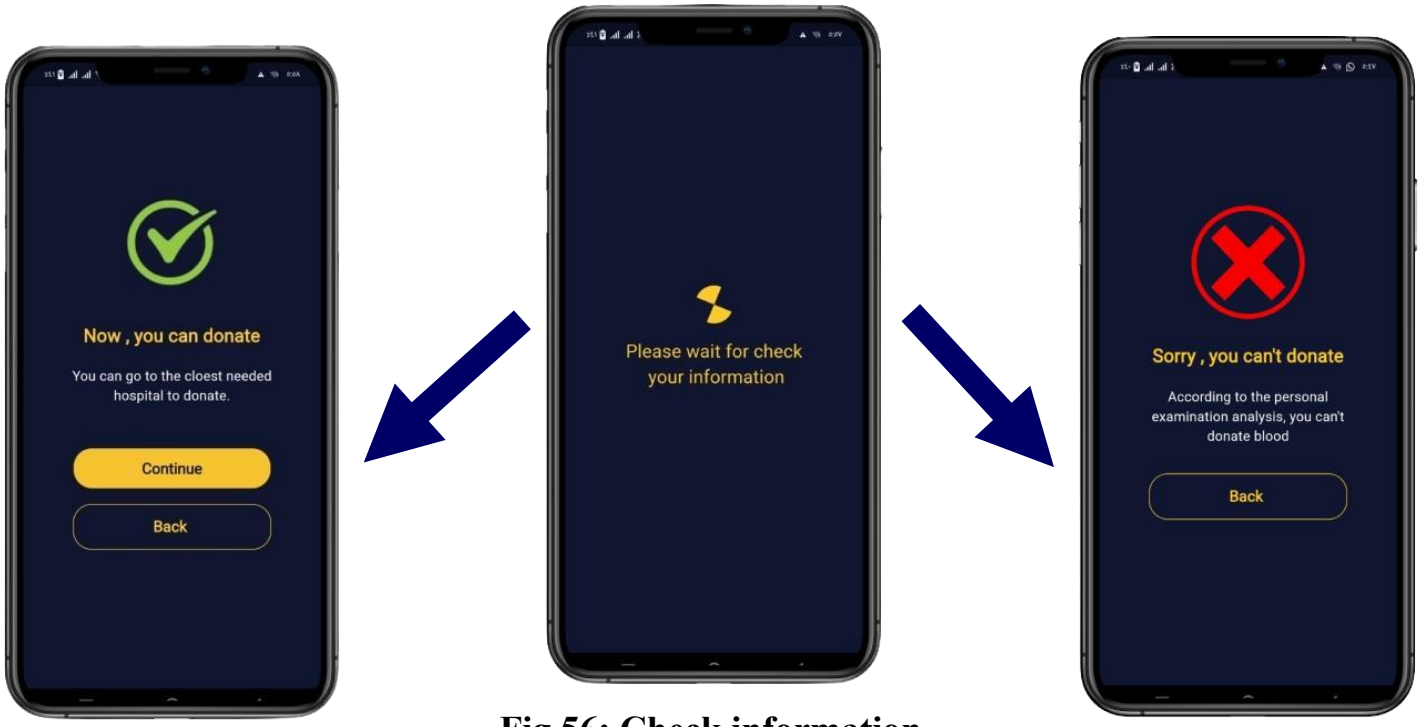


Fig 56: Check information

User data is examined using an artificial intelligence model, predicting the user's ability to donate blood. Then, the user's donation capability is displayed, and
the user can know the reason for not donating blood.

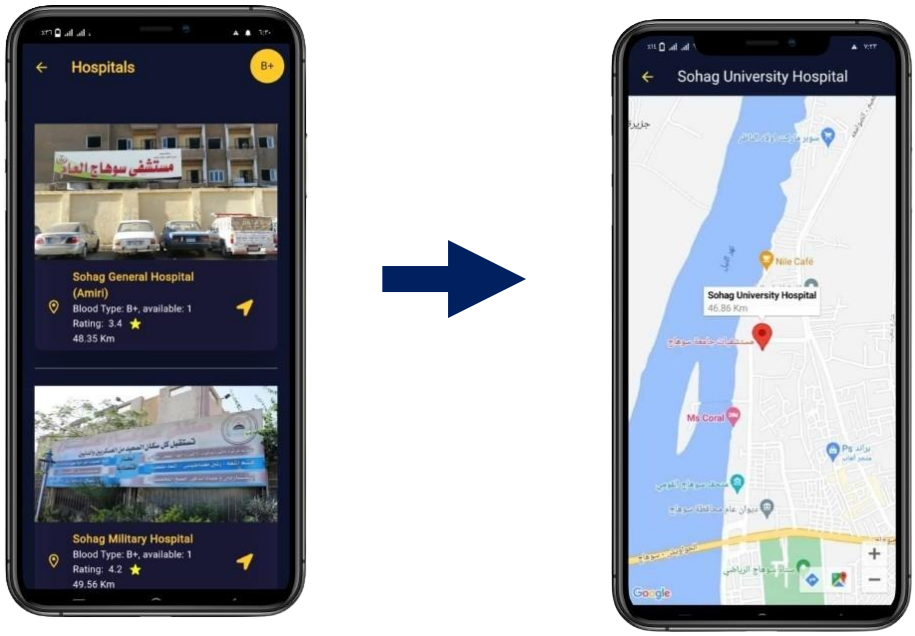


Fig 57: Hospital & Map Screen

User's Personal Location: The service first determines the user's personal location, either through explicit input or by accessing the device's location services.
Database of Hospitals: It then accesses a database or repository containing information about hospitals in the vicinity. This database includes details such as hospital names, addresses, and the blood types they are in need of.

Matching Blood Types: The service identifies hospitals that are in need of the same blood type as that of the user. This involves matching the user's blood type with the blood types required by hospitals listed in the database.

Proximity Calculation: Using the user's location and the geographical coordinates of each hospital in the database, the service calculates the distance between the user and each hospital.

Closest Hospital Selection: In cases where multiple hospitals require the same blood type, the service prioritizes the hospital that is closest to the user's location. This ensures that the user is presented with the most convenient donation option.

Display of Hospitals: Finally, the service displays a list of hospitals that match the user's blood type, sorted by proximity. The user can then choose from this list which hospital they prefer to donate to.

Integration with Google Maps: The service is integrated with the Google Maps API, which allows it to access geographical data and mapping functionalities provided by Google.

Map Display: Once the locations of all the hospitals are retrieved, the service dynamically generates a map interface, utilizing the Google Maps API. This map interface displays markers representing the locations of the hospitals.

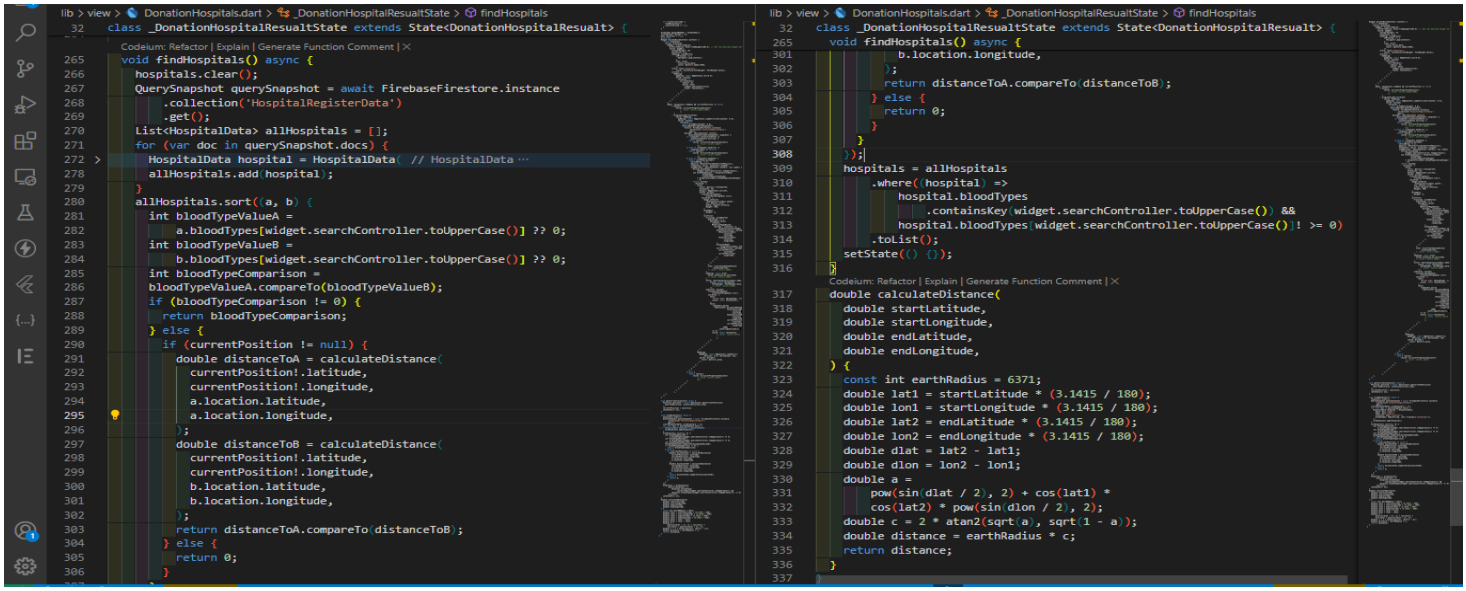


Fig 58: Ascending hospital logic (2)

getCurrentLocation() Function:

This function retrieves the current position (latitude and longitude) of the device using the Geolocator package. It sets the obtained position to the 'currentPosition' variable and triggers a UI update using setState().

findHospitals() Function:

Clears the 'hospitals' list to prepare for populating it with updated data.
Retrieves data from the Firestore database under the 'HospitalRegisterData' collection using FirebaseFirestore.
Iterates over each document in the QuerySnapshot, creating HospitalData objects and adding them to the 'allHospitals' list.
Sorts the 'allHospitals' list based on the blood type requirements and, if available, the proximity to the user's current location.
Filters the sorted list to include only hospitals that require the blood type specified by the 'searchController'.
Updates the 'hospitals' list with the filtered results and triggers a UI update using setState().

calculateDistance() Function:

Calculates the distance between two geographical points (latitude and longitude) using the Haversine formula.
Accepts start and end latitude and longitude coordinates as parameters.
Converts latitude and longitude values from degrees to radians.
Computes the Haversine formula to calculate the distance in kilometers.
Returns the calculated distance.

4.17 Hospital Services

Your services offer a streamlined solution for hospital employees, simplifying the management of blood donations and blood types. By providing a centralized platform,

staff can efficiently handle donor information and match blood types with patient needs. Moreover, your system enables swift communication through notifications, ensuring

prompt contact with donors when urgent donations are required. This efficiency saves time and resources, ultimately benefiting patient care by ensuring timely access to vital blood supplies.

After entering the secret key, the hospital gains access to the application's home page. Where display the hospital Name and its image Here, it encounters the following services:

Fig 59: Hospital Service

Find Blood Type: This feature allows hospital staff to search for specific blood types needed for patients. It provides a quick and convenient way to locate available blood supplies within the hospital's network.

Check Donor: Hospital staff can use this service to verify donor eligibility and suitability for blood donation. It enables them to ensure that donors meet the necessary criteria before accepting blood donations.

Create QR Blood Type: Hospitals have the ability to generate QR codes representing different blood types. These QR codes serve as a digital identifier for blood types, making it easier to manage and track blood supplies.

Add Blood Type by QR Code: This functionality enables hospitals to add new blood types to their inventory by scanning QR codes. It streamlines the process of updating blood type information, ensuring accuracy and efficiency.

Remove Blood Type by QR Code: Hospitals can also remove blood types from their inventory by scanning corresponding QR codes. This helps maintain an up-to-date record of available blood supplies and prevents outdated or incorrect information.

OCR (Optical Character Recognition): The application incorporates OCR technology, allowing hospital staff to extract text from images or documents. This feature can be used to digitize handwritten forms, medical records, or other documents related to blood donation and patient care.

Notification System: Hospitals can utilize the notification system to send alerts and updates to donors, volunteers, and staff members. This ensures timely communication and coordination, particularly during emergencies or urgent blood donation requests.

Create Qr Code

This service provided to hospitals assists in creating barcodes to retrieve data about blood types through the following steps:

1. **First, enter Donor Name:**
    - Input the name of the blood donor.
2. **Enter National ID:**
    - Input the national identification number of the donor.
3. **Option to Add More Data:**
    - There is an option to input additional data if required.
4. **Select Blood Type:**
    - Choose the blood type of the donor from the available options.
5. **Generate Unique Serial Number:**
    - Each QR code generated will have a unique serial number that differs for each blood type.

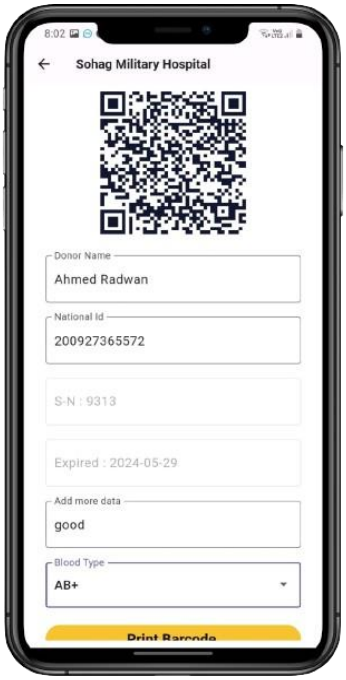


Fig 60: Create QR code

6. **Show Blood Type Expiry Date:**
    - The barcode will display the expiration date of the blood type.
7. **Button to Connect to Printer:**
    - A button is available to connect to a printer and print the QR code.

```
Container(
  width: 200,
  height: 200,
  child: BarcodeWidget(
    color: _theme.isDarkMode ? Colors.white : background,
    data:
        '${donar.donarName} ${donar.donarId} ${donar.serialNum} ${donar.bloodType} ${postId} ${
          donar.expiredDate} ${widget.hospitalName.replaceAll(' ', '-')} ${donar.moreDetails}',
    barcode: Barcode.qrCode(),
  ), // BarcodeWidget
), // Container
```

Fig 61: Barcode data

1. **Data:**
    - data: '${donar.donarName} ${donar.donarId} ${donar.serialNum} ${donar.bloodType} ${postId} ${donar.expiredDate} ${widget.hospitalName.replaceAll(' ', '-')} ${donar.moreDetails}':
        - Concatenates various pieces of donor and hospital information into a single string, which will be encoded into the QR code.
        - Includes:
            - donar.donarName: Donor's name.
            - donar.donarId: Donor's ID.
            - donar.serialNum: Donor's unique serial number.
            - donar.bloodType: Donor's blood type.
            - postId: Some additional post ID.
            - donar.expiredDate: Expiry date of the donor’s blood type.
            - widget.hospitalName.replaceAll(' ', '-'): Hospital name with spaces replaced by hyphens.
            - donar.moreDetails: Any additional details about the donor.

2. **Barcode Type:**
    - barcode: Barcode.qrCode(): Specifies that the type of barcode to generate is a QR code.

Add Blood Type

Fig 62: Add blood type screen

This service provided to hospitals assists in managing blood type information by scanning barcodes and searching through the following steps:

1. **Add Blood Type by Scanning Barcode:**
   - The service allows users to scan a barcode to add blood type information.
2. **Extract and Add Information:**
   - After scanning the barcode, the service extracts and adds the following details about the donor:
     - **Name**
     - **National ID**
     - **Blood Type**
     - **Blood Type Date**
     - **Blood Type Expiry Date**
     - **More Details**
3. **Add Serial Number:**
   - The serial number associated with the blood type is also added to the system.
4. **Search by Serial Number:**
   - The service provides a feature to search for blood type information using the serial number.
5. **Show Success Dialog:**
   - When the blood type information is successfully added, a dialog is displayed confirming that the blood type has been added.

```
Future<void> scanQrCode() async {
  try {
    FlutterBarcodeScanner.scanBarcode('#2A99CF', 'Cancel', true, ScanMode.QR)
        .then(
      (value) {
        List dataList = value.split(' ');
        String moreData = '';
        for (var i = 7; i < dataList.length; i++) {
          moreData = moreData + ' ' + dataList[i];
        }
        String name = dataList[0].replaceAll('-', ' ');
        String hospital = dataList[6].replaceAll('-', ' ');
        addData(
          donarName: name,
          donarId: dataList[1],
          serialNumber: dataList[2],
          bloodType: dataList[3],
          uId: dataList[4],
          expiredDate: dataList[5],
          hospitalName: hospital,
          moreDetails: moreData,
        );
      },
    );
  } catch (e) {
    print(e.toString());
  }
}
```

Fig 63: Add blood type function

1. **Function Definition:**
    - Future<void> scanQrCode() async {: Defines an asynchronous function named scanQrCode which returns a Future and can be awaited.
2. **Try-Catch Block:**
    - try { ... } catch (e) { ... }: Wraps the code inside a try-catch block to handle any potential errors that may occur during barcode scanning.

3. **Barcode Scanning:**
    - FlutterBarcodeScanner.scanBarcode(...).then((value) { ... });: Initiates the process of scanning a barcode using the FlutterBarcodeScanner package. It takes parameters including:
        - '#2A99CF': Color for the scan animation.
        - 'Cancel': Text for the cancel button.
        - true: Option to show the flashlight toggle button.
        - ScanMode.QR: Specifies that the scanner should detect QR codes.
        - The then function is called when the scanning is completed, providing the scanned value.

4. **Data Extraction:**
    - The scanned value is split into a list of strings using the space (' ') delimiter: List dataList = value.split(' ');.
    - The extracted data includes:
        - Donor name, national ID, serial number, blood type, UID, blood type expiry date, hospital name, and additional details.
        - The moreData variable aggregates any additional details beyon the predefined data fields.

5. **String Manipulation:**
    - The extracted strings are processed to remove any hyphens ('-') and combine additional details if present.

6. **Data Addition:**
    - The extracted data is used to call the addData function, passing the relevant parameters such as donor name, national ID, blood type, etc.

7. **Error Handling:**
    - Any errors that occur during barcode scanning or data processing are caught and printed to the console for debugging purposes.

Check if Blood Type Exists:

- Before adding the data extracted from the QR code, check if the blood type already exists in the system.

Show Dialog if Blood Type Exists:

- If the blood type already exists, display a dialog informing the user that the blood type is already in the system.

Fig 64: Scan the same blood type (when add)

Remove Blood Type

Remove Blood Type by Scanning Barcode or Search by Serial Number:

- Users can remove a blood type from the system either by scanning its barcode or searching for it using its serial number.

Date Validation:

- Before removal, the system verifies that the blood type is not expired. This validation is based on a required date, which is 35 days from the date of donation.

Database Removal:

- Once validated, the blood type is removed from the database.

Fig 65: Remove blood type screen

Show Success Dialog:

- After successful removal, a dialog is displayed to confirm that the blood type has been deleted.

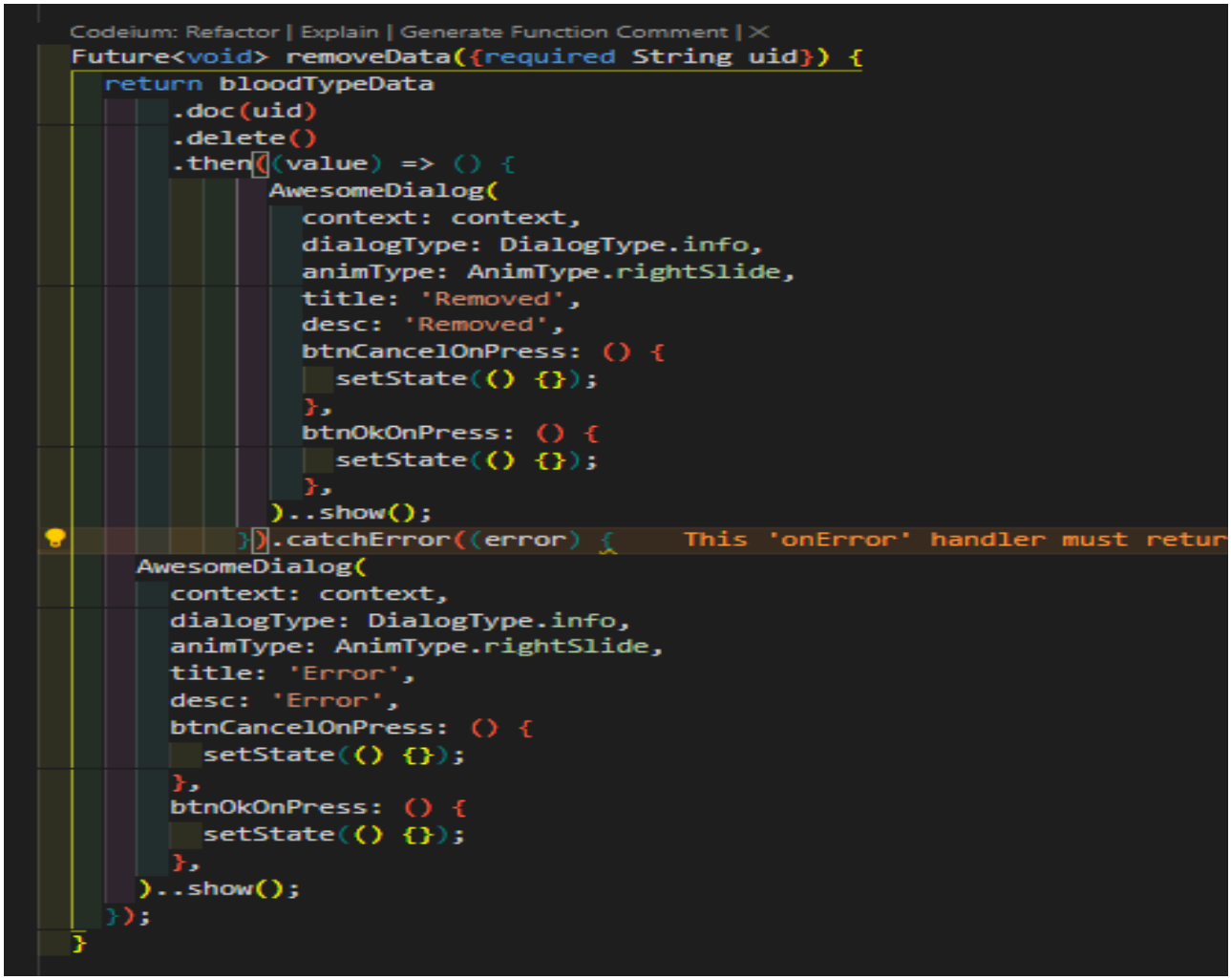


Fig 66: Remove blood type function

Function Definition:

- Future<void> removeData({required String uid}) {: Defines an asynchronous function named removeData that takes a required parameter uid, which represents the unique identifier of the blood type data to be removed.

Database Deletion:

- bloodTypeData.doc(uid).delete(): Deletes the document with the specified uid from the database collection bloodTypeData.

Handling Success and Error:

- .then((value) { ... }).catchError((error) { ... }): Sets up callbacks to handle the completion or failure of the deletion operation.
- If the deletion is successful, an AwesomeDialog is created with a success message, indicating that the blood type has been removed.

- If an error occurs during deletion, another AwesomeDialog is created with an error message.

Dialog Configuration:

- The AwesomeDialog is configured with parameters such as:
  - context: The context where the dialog should be displayed.
  - dialogType: The type of dialog to be displayed (info in this case).
  - animType: The animation type for displaying the dialog (rightSlide in this case).
  - title: The title of the dialog.
  - desc: The description or content of the dialog.
  - btnCancelOnPress: The callback function to execute when the cancel button is pressed.
  - btnOkOnPress: The callback function to execute when the OK button is pressed.

State Update:

- setState(() {});: After showing the dialog, the state of the widget is updated, triggering a rebuild to reflect any changes in the UI.

- **Check Blood Type Existence:**

- Before adding the data extracted from the QR code, the system checks if the blood type already exists.

- **Show Dialog if Blood Type Doesn't Exist:**

- If the blood type doesn't exist in the system, display a dialog to inform the user.

Fig 67: Scan the same blood type (when remove)

Notification

- **User or Hospital Initiation:**

- Users or hospitals initiate the notifications service when there is a deficiency in any blood type.

- **Data Input:**

- They provide the following information:
  - **Username:** Identity of the sender.
  - **Blood Type:** The blood type for which there is a deficiency.
  - **Place:** Location where the deficiency is observed.
  - **Phone Number:** Contact information for further communication.

Fig 68: create notification

Notification Trigger:

- Upon inputting this information, the system triggers a notification process.

Notification Distribution:

- Notifications are sent out to all users within the application.

Content of Notification:

- The notification includes details about the deficiency, such as the blood type, location, and contact information provided.

Purpose:

- The purpose of these notifications is to alert users about the shortage of specific blood types, encouraging them to donate if they are eligible and willing.

```dart
import '../view/hospital_dash_board/views/create_notfication_view.dart';

Codeium: Refactor | Explain
class NotificationHelper {
  sendNotification(topic,
      {required String userName,
      required String bloodType,
      required String place,
      required String phoneNumber}) async {
    var headersList = {
      'Accept': '*/*',
      'Content-Type': 'application/json',
      'Authorization':
          'key=AAAA8a_Wm9c:APA91bGmxIeIoPxiEExpV4i2KaMeKe6lMygjv9MZsCWNq5FGTY' +
          '8FiDIEtWCzUzWCfxtjcwUptRirifcayFSIR_Vzl5ioxf9T1SnYj1jChid4tjCatGBcv5I5D6JWa0FbenOmqKgVisBs'
    };
    var url = Uri.parse('https://fcm.googleapis.com/fcm/send');

    var body = await {
      "to": "/topics/$topic",
      "notification": {
        "title": 'New Notification',
        'body': '${userName} \n${bloodType} \n${place}',
        "mutable_content": true,
        "sound": "Tri-tone"
      },
      "data": {
        "userName": '${userName}',
        "place": '$place',
        "phoneNumber": '$phoneNumber',
      },
    };
```

Fig 69: create notification function

Method Definition:

- sendNotification: This method is responsible for sending a notification to a specified topic.

Parameters:

- topic: The topic to which the notification will be sent.
- userName: The name of the user or hospital sending the notification.
- bloodType: The blood type for which there is a deficiency.
- place: The location where the deficiency is observed.
- phoneNumber: The contact phone number provided for further communication.

HTTP Headers:

- The method sets up necessary HTTP headers including:
  - Accept: Specifies the accepted response format.
  - Content-Type: Specifies the content type of the request.
  - Authorization: Provides the authorization key required for sending notifications via Firebase Cloud Messaging (FCM).

URL:

- The URL for sending FCM notifications is set to https://fcm.googleapis.com/fcm/send.

Notification Payload:

- The notification payload includes:
  - "to": The target topic to which the notification will be sent.
  - "notification": Contains the title and body of the notification message.
    - "title": The title of the notification, set to 'New Notification'.
    - "body": The body of the notification, including the user name, blood type, and place of the deficiency.
    - "mutable_content": Indicates whether the notification content can be modified by the client app.
    - "sound": Specifies the sound to be played when the notification is received.
  - "data": Additional data associated with the notification, including the user name, place, and phone number. This data can be accessed by the client app even if the app is in the background or terminated.

Asynchronous Operation:

- The method operates asynchronously, allowing for non-blocking execution while sending the notification.

Notification Sent for All Users Who Join the Application

2. **Notification Elements:**
    - **Icon App:** Display the app's icon.
    - **Name of User:** Include the name of the user who has joined.
    - **Notification Content:**
        - **Blood Type:** Include the blood type of the new user.
        - **User Location:** Include the location of the new user.

```
getForGroundMessage(BuildContext context) {
  FirebaseMessaging.onMessage.listen((RemoteMessage message) {
    if (message.notification != null) {
      print('Got a message whilst in the foreground!');
      print('Message data: ${message.notification!.title}');
      print(message.notification!.body);
      ScaffoldMessenger.of(context).showSnackBar(SnackBar(
        content: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            Text('${message.data['userName']}'),
            Text('${message.notification!.body}'),
            Text('${message.data['place']}'),
          ],
        ),
        duration: Duration(seconds: 3),
        action: SnackBarAction(
          label: 'Open',
          onPressed: () {
            Navigator.pushReplacement(
                context,
                MaterialPageRoute(
                    builder: (context) => const NotififcationView()));
          },
        ),
      ));
    } else {
      print('error');
    }
  });
}
```

Fig 70: Get ForGroundMessage function

Function Explanation: getForGroundMessage(BuildContext context)

1. **Function Definition:**
    - getForGroundMessage(BuildContext context) is a function that takes BuildContext as an argument to access UI elements and context for navigation.

2. **Listening to Foreground Messages:**
    - FirebaseMessaging.onMessage.listen((RemoteMessage message) { sets up a listener for incoming messages while the app is in the foreground.

3. **Checking for Notification Content:**
    - if (message.notification != null) { checks if the received message contains a notification.

4. **Logging the Message:**
    - print('Got a message whilst in the foreground!'); logs a message indicating that a notification was received.
    - print('Message data: ${message.notification!.title}'); logs the title of the notification.
    - print(message.notification!.body); logs the body of the notification.

5. **Displaying a SnackBar:**
    - ScaffoldMessenger.of(context).showSnackBar(SnackBar(displays a SnackBar to show the notification details within the app.

6. **Setting SnackBar Content:**
    - content: Column(sets the content of the SnackBar to be a Column widget.
    - crossAxisAlignment: CrossAxisAlignment.start, aligns the column's children to the start.
    - children: [contains a list of Text widgets:
        - Text('${message.data['userName']}'), displays the user name from the message's data.
        - Text('${message.notification!.body}'), displays the body of the notification.
        - Text('${message.data['place']}'), displays the location from the message's data.

7. **Setting SnackBar Duration:**
    - duration: Duration(seconds: 3), sets the SnackBar to be visible for 3 seconds.

8. **Adding SnackBar Action:**
    - action: SnackBarAction(adds an action button to the SnackBar.
    - label: 'Open', sets the label of the action button to "Open".
    - onPressed: () { defines the function to be executed when the action button is pressed:
        - Navigator.pushReplacement(replaces the current screen with a new screen.

- context, uses the current context for navigation.
- MaterialPageRoute(creates a route to the new screen.
- builder: (context) => const NotififcationView())); navigates to the NotificationView screen.

9. **Handling Missing Notification:**
    - } else { provides an alternative action if the message does not contain a notification.
    - print('error'); logs an error message.

10. **Closing Function:**
    - }); closes the listener function.
    - } closes the getForGroundMessage function.

Implementation Steps

1. **Create Notification:**
    - User creates a notification.
    - Store the notification details in a persistent storage (e.g., database).
2. **Display List of Notifications:**
    - Fetch the list of notifications from the storage.
    - Display the notifications in a list view within the app.
3. **View Notification:**
    - Provide an option to view the details of a notification.
    - When a notification is selected, display its details on a new screen or in a dialog.
4. **Delete Notification:**
    - Provide an option to delete a notification.
    - When the delete option is selected, remove the notification from the storage and update the list view.

Fig 71: Notification screen

```dart
final FirebaseMessaging _firebaseMessaging = FirebaseMessaging.instance;
CollectionReference notification =
    FirebaseFirestore.instance.collection('notification');
Future<void> addNotification() async {
  try {
    await notification.add({
      'userName': userName,
      'bloodType': bloodType,
      'phoneNumber': phoneNumber,
      'place': place,
      'id': id,
      'date': formatDate,
    });
    print('Document added to collection "users"');
    print(FirebaseAuth.instance.currentUser == null
        ? FirebaseAuth.instance.currentUser!.uid
        : id);
  } catch (e) {
    print('Error adding document: $e');
  }
}
```

Fig 72: Add Notification function

Initialize Firebase Messaging Instance:

- final FirebaseMessaging _firebaseMessaging = FirebaseMessaging.instance;
  - This initializes an instance of Firebase Messaging to handle notifications within the application.

Reference Firebase Firestore Collection:

- CollectionReference notification = FirebaseFirestore.instance.collection('notification');
  - This sets up a reference to the notification collection in Firebase Firestore where notification data will be stored.

Define Asynchronous Function to Add Notification:

- Future<void> addNotification() async {
  - This line defines an asynchronous function named addNotification to add a notification document to the Firestore collection.

Try Block to Handle Addition of Notification:

- try {
  - This begins a try block to execute the addition of a document to the Firestore collection and catch any errors that may occur.

Add Notification Data to Firestore:

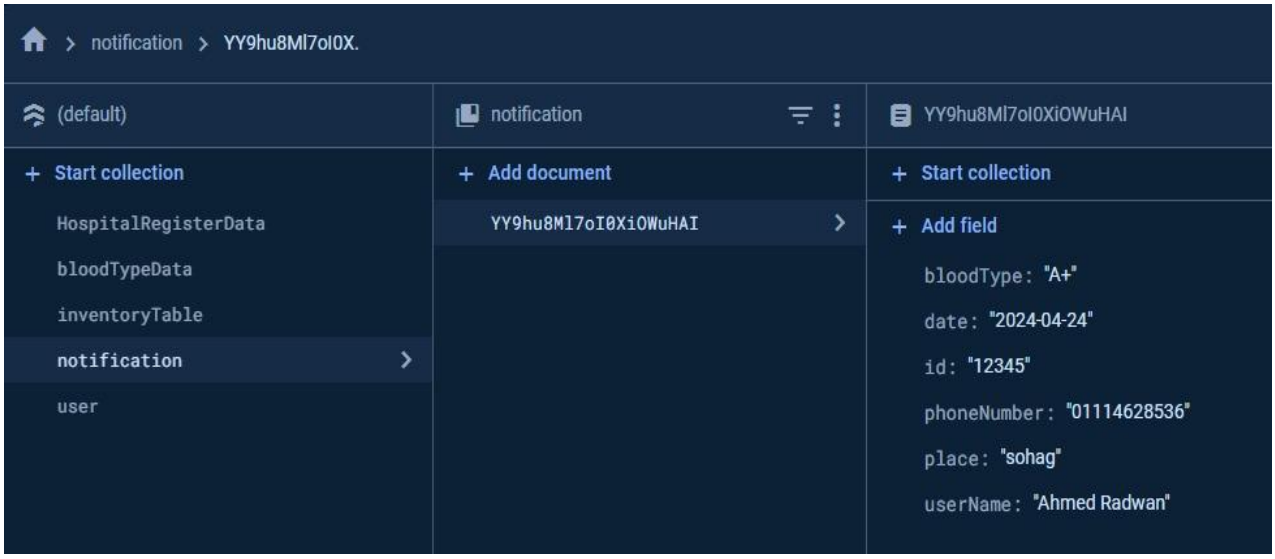


Fig 73: Firebase database (notification)

- await notification.add({
  - This line uses the add method to add a new document to the notification collection with the specified fields.
  - userName: The name of the user.
  - bloodType: The blood type of the user.
  - phoneNumber: The phone number of the user.
  - place: The place associated with the notification.
  - id: A unique identifier for the notification.
  - date: The formatted date of the notification.

Chatbot

When a person is unable to donate blood Go to the chatbot page Here he can correspond with the bot using artificial intelligence

It analyzes the message through which the patient complains about it and then predicts the disease that prevents him from donating He can provide some important advice on dealing with this disease

Fig 74: Chatbot screen

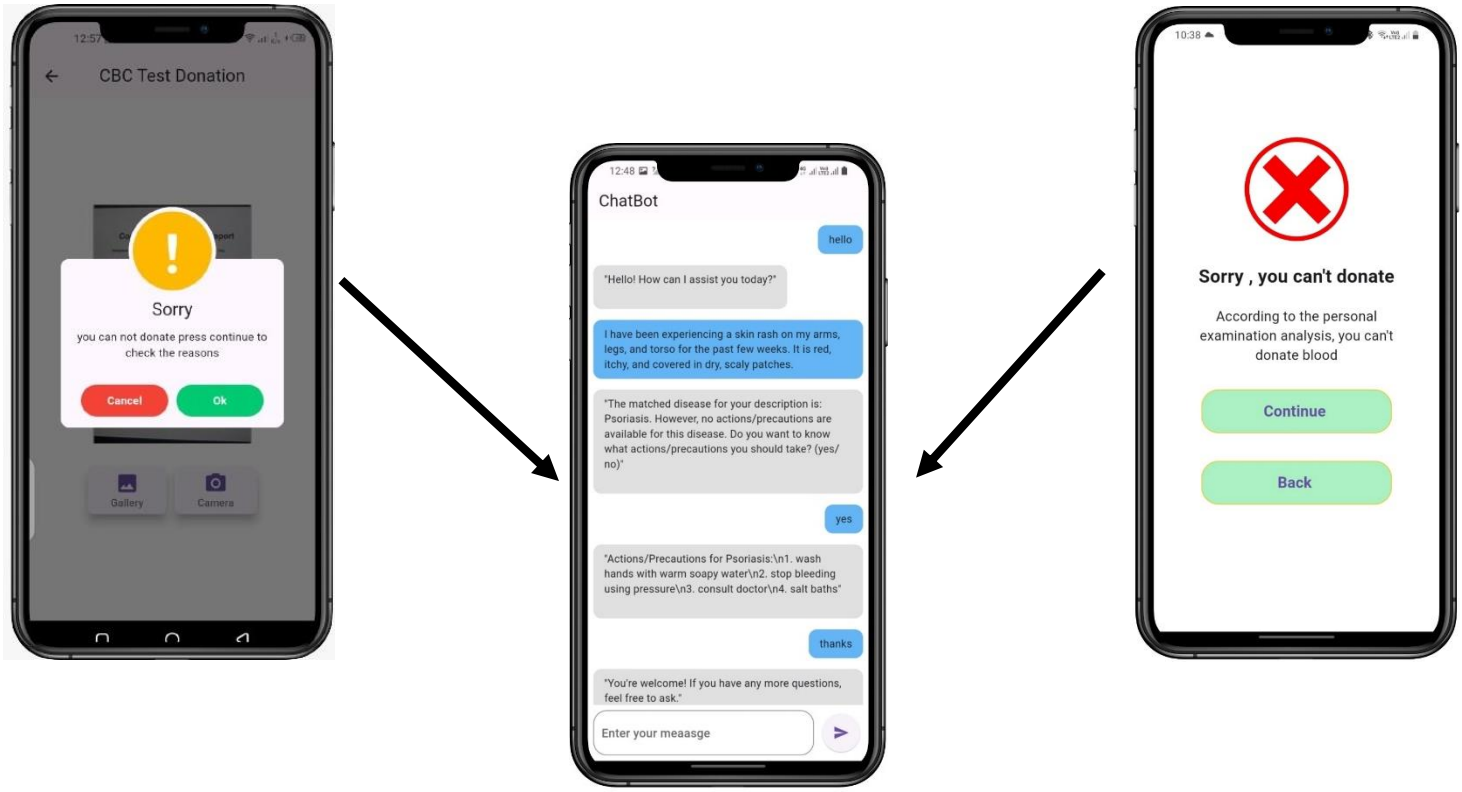


Fig 75: navigate to Chatbot

```
Codeium: Refactor | Explain | Generate Function Comment | ×
Future<void> sendUserInput(String userInput) async {
  final url = Uri.parse('https://hello-h.onrender.com/chat');
  final translator = GoogleTranslator();
  final headers = {"Content-Type": "application/json"};
  String translatedText = userInput;
  bool isArabic = false;
  if (_isArabic(userInput)) {
    isArabic = true;
    translatedText =
        (await translator.translate(userInput, from: 'ar', to: 'en')).text;
  }
  final body = json.encode({"user_input": translatedText});
  try {
    setState(() {
      _messages.add({
        'text': userInput,
        'isUser': true,
      });
      _isTyping = true;
    });
    final response = await http.post(url, headers: headers, body: body);
    if (response.statusCode == 200) {
      var responseText = response.body;
      if (isArabic) {
        responseText =
            (await translator.translate(responseText, from: 'en', to: 'ar'))
                .text;
      }
      _currentResponseWords = responseText.split(' ');
      setState(() {
```

Fig 76: Send user input function

Detailed Explanation:

1. **Function Definition**:
   - The function sendUserInput is asynchronous and returns void.
   - It takes a userInput parameter, which represents the text input provided by the user.
2. **URL and Headers**:
   - The function defines the URL of the chat API endpoint and sets the request headers.
3. **Translation Handling**:
   - If the user input is in Arabic, it sets a boolean flag isArabic to true.
   - It then translates the user input from Arabic to English using the Google Translator library.

4. The translated text is stored in translatedText.

```
Future<void> sendUserInput(String userInput) async {
    if (response.statusCode == 200) {
      var responseText = response.body;
      if (isArabic) {
        responseText =
            (await translator.translate(responseText, from: 'en', to: 'ar'))
                .text;
      }
      _currentResponseWords = responseText.split(' ');
      setState(() {
        _messages.add({
          'text': '',
          'isUser': false,
        });
        _currentWordIndex = 0;
        _isTyping = false;
      });
      await animateMessage();
    } else {
      print('Failed to send user input: ${response.statusCode}');
      setState(() {
        _isTyping = false;
      });
    }
  } catch (e) {
    print('Error: $e');
    setState(() {
      _isTyping = false;
    });
  }
}
```

Fig 77: translate user input function

5. **Preparing Request Body**:
    - The function constructs the request body in JSON format, including the translated user input.

6. **Updating UI**:
    - Before sending the request, the function updates the UI to display the user's input.
    - It adds the user's input to the _messages list and sets the _isTyping flag to true to indicate that the app is typing a response.

7. **Sending HTTP POST Request**:
    - The function sends an HTTP POST request to the specified URL with the request headers and body.
    - It waits for the response from the API.

8. **Handling Response**:
    - If the response status code is 200 (OK), it processes the response text.
    - If the user input was in Arabic, it translates the response text back to Arabic.
    - It updates the _messages list to include the received response.
    - It sets _isTyping to false to indicate that the app has finished typing.

9. **Error Handling**:
    - If an error occurs during the HTTP request or response processing, it catches the error and prints it.
    - It sets _isTyping to false to handle errors gracefully.

Drawer

- Department
- Notification
- Changing between dark and light Mode
- Changing between languages
- Sign out

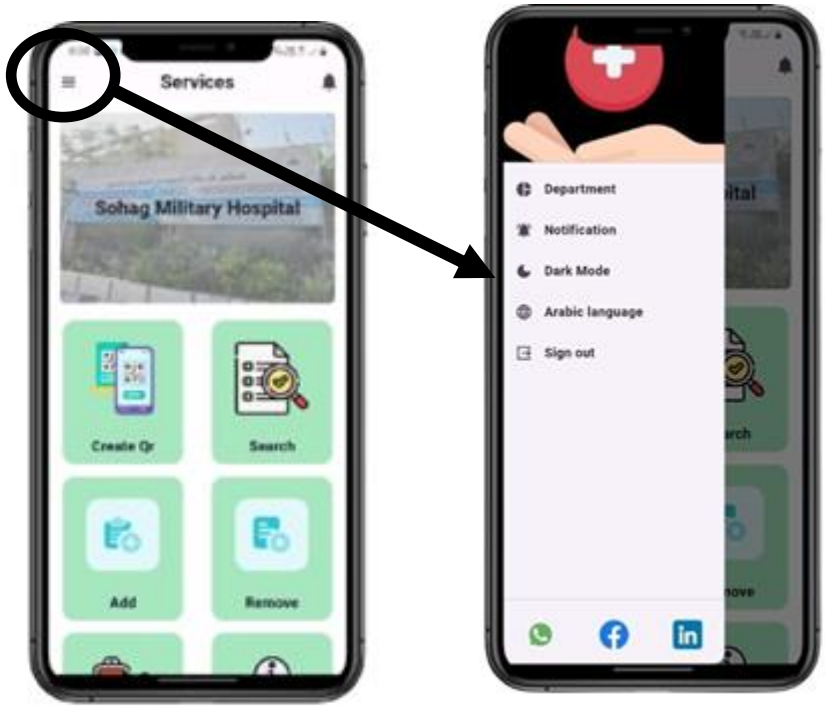


Fig 78: Drawer

Departement

- **Daily Operations Recording**:

- The service records daily operations related to adding or deleting blood types.
- Each operation is time-stamped and logged for future reference.

- **Responsibility Tracking**:

- The system allows the responsible person to review all recorded operations.
- This ensures accountability and transparency in managing blood type data.

- **Search Functionality by Serial Number**:

- Users can search for specific operations by entering the serial number.
- This feature facilitates quick access to relevant information when needed.

Fig 79: Departement

```
Expanded(
    child: SingleChildScrollView(
  child: StreamBuilder<List<QueryDocumentSnapshot>>(
    stream: FirebaseFirestore.instance
        .collection('inventoryTable')
        .where('id', isEqualTo: widget.id)
        .snapshots()
        .map((snapshot) => snapshot.docs),
    builder: (BuildContext context,
        AsyncSnapshot<List<QueryDocumentSnapshot>> snapshot) {
      if (snapshot.hasError) {
        return Center(
            child: Text(S.of(context).Something_went_wrong)); // Center
      }
      if (snapshot.connectionState == ConnectionState.waiting) {
        return Center(child: LinearProgressIndicator());
      }
      List<QueryDocumentSnapshot> dataList = snapshot.data!;
      if (searchQuery.isNotEmpty) {
        dataList = dataList.where((document) {
          Map<String, dynamic> dataItem =
              document.data() as Map<String, dynamic>;
          String serialNumber =
              dataItem['serialNumber'].toString();
          return serialNumber
              .toLowerCase()
              .contains(searchQuery.toLowerCase());
        }).toList();
      }
```

Fig 80: Get department data function

- **StreamBuilder Configuration**:
  - Utilizes a StreamBuilder widget to asynchronously build UI components based on the data stream.
  - The stream originates from a Firestore collection named 'inventoryTable'.
  - Filters documents in the collection based on the 'id' field matching the value of widget.id.
- **Error Handling**:
  - Checks if the snapshot has encountered any errors.
  - If an error occurs, it displays a centered message indicating that something went wrong, utilizing localized text from the app's context.
- **Connection State Handling**:
  - Checks the connection state of the snapshot.
  - If the connection state is waiting, it displays a centered linear progress indicator, indicating that data is being fetched.

- **Snapshot Data Processing**:

  - Extracts the list of QueryDocumentSnapshot objects from the snapshot.
  - If there's a search query present (searchQuery is not empty), it filters the dataList based on the serial number.
  - Converts each document's data to a map of string-dynamic pairs.
  - Extracts the serial number from each document's data and converts it to lowercase for case-insensitive comparison.
  - Filters the dataList to include only those documents whose serial number contains the search query string (case-insensitive).
  - Converts the filtered result back into a list.

- **Return Value**:

  - Returns the filtered dataList or the original snapshot.data if no search query is present.
  - This filtered list will be used to build UI components reflecting the search results or the entire inventory data.

Cell Construction for DataRow Widget:

```
dataList.map((DocumentSnapshot document) {
Map<String, dynamic> dataItem =
    document.data() as Map<String, dynamic>;
return DataRow(
  cells: [
    DataCell(Row(
      children: [
        dataItem['process'] == 'add'
            ? Icon(Icons.arrow_forward,
                color: Colors.green) // Icon
            : Icon(
                Icons.arrow_back,
                color: Colors.red,
              ), // Icon
        SizedBox(
          width: 10,
        ), // SizedBox
        Text(dataItem['bloodType'].toString()),
      ],
    )), // Row // DataCell
    DataCell(Text(dataItem['UpdatedDate'])),
    DataCell(Text(dataItem['serialNumber'])),
  ],
); // DataRow
}).toList(),
```

Fig 81: determine blood type transaction

- Constructs cells for the DataRow widget, representing different attributes of the document.
    - The first cell:
        - Displays an icon indicating the type of operation ('add' or 'delete') based on the value of the 'process' field in the document.
        - If the process is 'add', it shows a green forward arrow icon.
        - If the process is 'delete', it displays a red backward arrow icon.
    - The second cell:
        - Contains the 'bloodType' field value, representing the blood type associated with the operation.
    - The third cell:
        - Contains the 'UpdatedDate' field value, representing the date of the operation.
    - The fourth cell:
        - Contains the 'serialNumber' field value, uniquely identifying the operation.

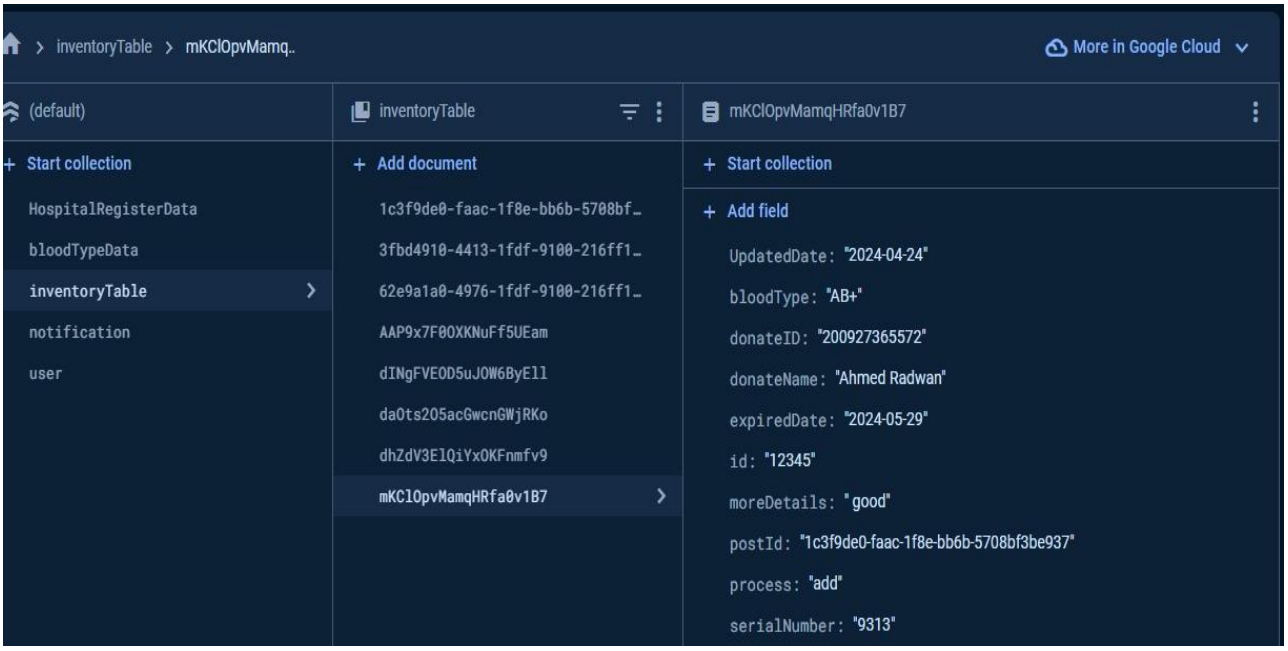


Fig 82: Firebase database (inventory)

Add Data to Inventory Table:

- Use await inventoryTable.add() to add a new entry to the inventory table.
- Include the following details in the entry:
    - donateName: The name of the donor.
    - bloodType: The type of blood being added.
    - donateID: The ID of the donor.
    - expiredDate: The expiration date of the blood donation.
    - serialNumber: The serial number of the blood donation.
    - moreDetails: Any additional details about the blood donation.
    - postId: A unique ID (likely representing the post or event related to the donation).

- UpdatedDate: The current date, formatted as yyyy-MM-dd.
- id: An ID associated with the widget or context from which this code is running.
- process: Set to 'add' to indicate that this entry represents an addition of blood.

My Notification

Displays the current user notifications.
User Interaction:

Allows users to view their notifications.
Provides an option for users to delete notifications.
Notification Management:

A list or panel displaying the notifications.
A delete button or icon next to each notification for removal.
Dynamic Updates:
The page dynamically updates to reflect the current state of notifications.

Fig 83: MyNotification screen

```dart
Future<void> removeNotification(String notificationId, String userId) async {
  try {
    await FirebaseFirestore.instance
        .collection('notification')
        .doc(notificationId)
        .delete();
    print('Notification removed');
  } catch (e) {
    print('Error removing notification: $e');
  }
  setState(() {});
}
```

Fig 84: Remove notification function

Function: removeNotification

1. **Function Signature**:
    - The function removeNotification is defined as Future<void>.
    - It accepts two parameters: String notificationId and String userId.

2. **Try-Catch Block**:
    - The function is wrapped in a try-catch block to handle potential errors.

3. **Delete Notification**:
    - Within the try block:
        - Use await FirebaseFirestore.instance.collection('notification').doc(notificationId).delete() to delete the document with the specified notificationId from the notification collection in Firestore.
        - Print the message 'Notification removed' to the console upon successful deletion.

4. **Catch Errors**:
    - Within the catch block:
        - Catch any exceptions (e) that occur during the deletion process.
        - Print the error message 'Error removing notification: $e' to the console.

5. **Update State**:
    - Call setState(() {}) to trigger a state update in the widget that contains this function. This ensures that the UI reflects the changes after the notification is removed.

Dark Theme

Toggle Switch/Button:

Design and place a toggle switch or button labeled "Dark Mode" within the app's settings

Dark Theme Development:
Develop a dark theme with a color palette suitable for low-light environments, ensuring good contrast and readability.

Create light and dark theme versions of UI elements such as backgrounds, text, buttons, and icons.

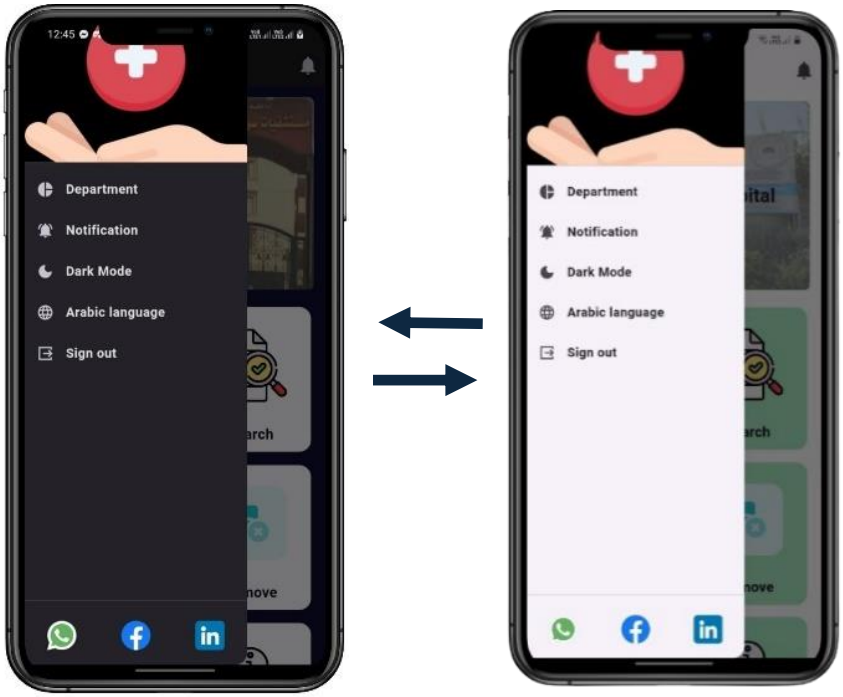


Fig 85: change Theme

```dart
class ThemeProvider with ChangeNotifier {
  bool _isDarkMode = false;

  bool get isDarkMode => _isDarkMode;

  ThemeProvider() {
    _loadTheme();
  }

  void _loadTheme() async {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    _isDarkMode = prefs.getBool('isDarkMode') ?? false;
    notifyListeners();
  }

  void _saveTheme(bool isDarkMode) async {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    await prefs.setBool('isDarkMode', isDarkMode);
  }

  void toggleTheme() {
    _isDarkMode = !_isDarkMode;
    _saveTheme(_isDarkMode);
    notifyListeners();
  }
}
```

Fig 86: change theme function

Class: ThemeProvider

1. **Inheritance**:
   - The class ThemeProvider uses the ChangeNotifier mixin, allowing it to notify listeners when changes occur.

2. **Private Field**:
   - _isDarkMode: A private boolean field initialized to false that tracks the current theme mode.

3. **Public Getter**:
   - isDarkMode: A public getter that returns the current value of _isDarkMode.

4. **Constructor**:
   - ThemeProvider(): The constructor initializes the ThemeProvider and calls the private method _loadTheme() to load the saved theme preference.

5. **Private Method: _loadTheme()**:
    - Asynchronously obtains an instance of SharedPreferences.
    - Retrieves the saved theme preference ('isDarkMode' key) from SharedPreferences.
    - Sets _isDarkMode to the retrieved value or false if the key does not exist.
    - Calls notifyListeners() to update any listeners with the new value.

6. **Private Method: _saveTheme(bool isDarkMode)**:
    - Asynchronously obtains an instance of SharedPreferences.
    - Saves the current theme preference (isDarkMode value) to SharedPreferences under the 'isDarkMode' key.

7. **Public Method: toggleTheme()**:
    - Toggles the value of _isDarkMode (i.e., if _isDarkMode is false, it sets it to true, and vice versa).
    - Calls _saveTheme(_isDarkMode) to save the updated theme preference.
    - Calls notifyListeners() to update any listeners with the new theme mode.

Change Language

Toggle Switch/Button:

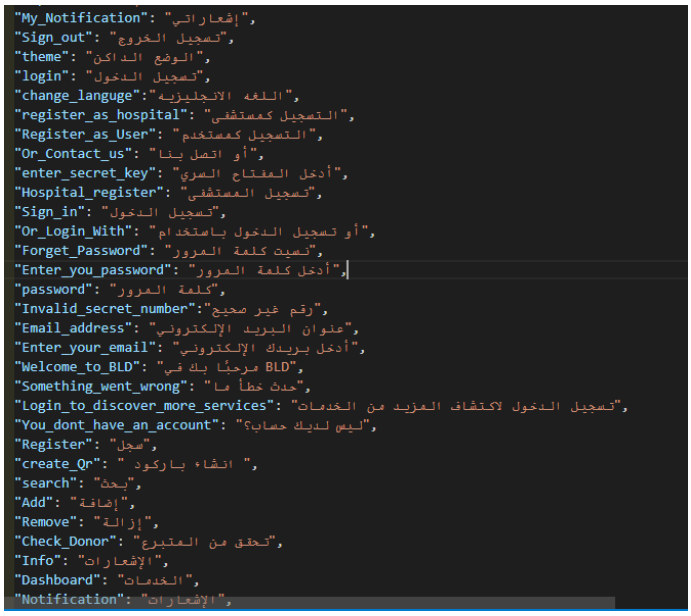
Design a toggle switch or button labeled to switch between Arabic and English.

Place the toggle within the app's settings menu or prominently on the main interface for easy access.

Ensure the button clearly indicates the current language and the language to switch to.

Fig 87: Change language

Localization Setup



```
"My_Notification": "إشعاراتي",
"Sign_out": "تسجيل الخروج",
"theme": "الوضع الداكن",
"login": "تسجيل الدخول",
"change_languge":"اللغه الانجليزيه",
"register_as_hospital": "التسجيل كمستشفى",
"Register_as_User": "التسجيل كمستخدم",
"Or_Contact_us": "أو اتصل بنا",
"enter_secret_key": "أدخل المفتاح السري",
"Hospital_register": "تسجيل المستشفى",
"Sign_in": "تسجيل الدخول",
"Or_Login_With": "أو تسجيل الدخول باستخدام",
"Forget_Password": "نسيت كلمة المرور",
"Enter_you_password": "أدخل كلمة المرور",
"password": "كلمة المرور",
"Invalid_secret_number":"رقم غير صحيح",
"Email_address": "عنوان البريد الإلكتروني",
"Enter_your_email": "أدخل بريدك الإلكتروني",
"Welcome_to_BLD": "مرحبًا بك في BLD",
"Something_went_wrong": "حدث خطأ ما",
"Login_to_discover_more_services": "تسجيل الدخول لاكتشاف المزيد من الخدمات",
"You_dont_have_an_account": "ليس لديك حساب؟",
"Register": "سجل",
"create_Qr": " انشاء باركود ",
"search": "بحث",
"Add": "إضافة",
"Remove": "إزالة",
"Check_Donor": "تحقق من المتبرع",
"Info": "الإشعارات",
"Dashboard": "الخدمات",
"Notification": "الإشعارات",
```

```
"My_Notification": "My Notifications",
"Sign_out": "Sign out",
"Do_you_want_to_sign_out":"Do you want to sign out ?",
"theme": "Dark Mode",
"login": "Login",
"change_languge":"Arabic language",
"register_as_hospital": "Register as Hospital",
"Register_as_User": "Register as User",
"Or_Contact_us": "Or Contact us",
"enter_secret_key": "Enter secret key",
"Hospital_register": "Hospital Register",
"Sign_in": "Sign in",
"Loading":"Loading",
"Or_Login_With": "Or Login With",
"Forget_Password": "Forget Password",
"Enter_you_password": "Enter your password",
"password": "Password",
"Email_address": "Email Address",
"Enter_your_email": "Enter your email",
"Welcome_to_BLD": "Welcome to BLD",
"Login_to_discover_more_services": "Login to discover more services",
"You_dont_have_an_account" : "You dont have an account?",
"Register": "Register",
"create_Qr": "Create Qr",
"search": "Search",
"Add": "Add",
"Remove": "Remove",
"Check_Donor": "Check Donor",
"Info": "Notifications",
"Data":"Data",
"Dashboard": "Services",
```

Fig 88: English & Arabic Language

Identified Supported Languages:

We began by identifying the languages our app would support. We decided to support English and Arabic.
Created Localization Files:

Next, we created separate localization files for each supported language, such as en.json for English and ar.json for Arabic.
These files contained translations for all text strings used in our app.
Integrated Localization Packages:

To handle localization in Flutter, we included the intl package in our project. We added the package to our pubspec.yaml file and ran flutter pub get to install it.
Configuration

Sign out

- **Access SharedPreferences**:

- SharedPreferences pref = await SharedPreferences.getInstance();
- This line retrieves an instance of SharedPreferences, which is a key-value store for persistent data.

  The await keyword ensures that the function waits for the SharedPreferences instance to be available before proceeding.

```
onTap: () async {
  SharedPreferences pref =
      await SharedPreferences.getInstance();
  await pref.setBool('isLongging', false);
  Navigator.pushAndRemoveUntil(
    context,
    MaterialPageRoute(
      builder: (context) => JoinBloodDonationApp(),
    ), // MaterialPageRoute
    (route) => false,
  );
},
```

Fig 89: Signout

- **Set a Boolean Value**:

- await pref.setBool('isLongging', false);
- This line stores a boolean value (false) in the SharedPreferences under the key 'isLongging'. Again, await is used to ensure this operation completes before moving to the next step.

- **Navigate to a New Screen**:

- Navigator.pushAndRemoveUntil(...);
- This line initiates navigation to a new screen and removes all the previous routes from the navigation stack.

4.18 OCR and Flask Connection

This Flask application serves for analyzing Complete Blood Count (CBC) images to extract relevant blood parameters such as Platelet, RBC (Red Blood Cell), WBC (White Blood Cell), and Hemoglobin levels. The application utilizes optical character recognition (OCR) to process uploaded images.[5]

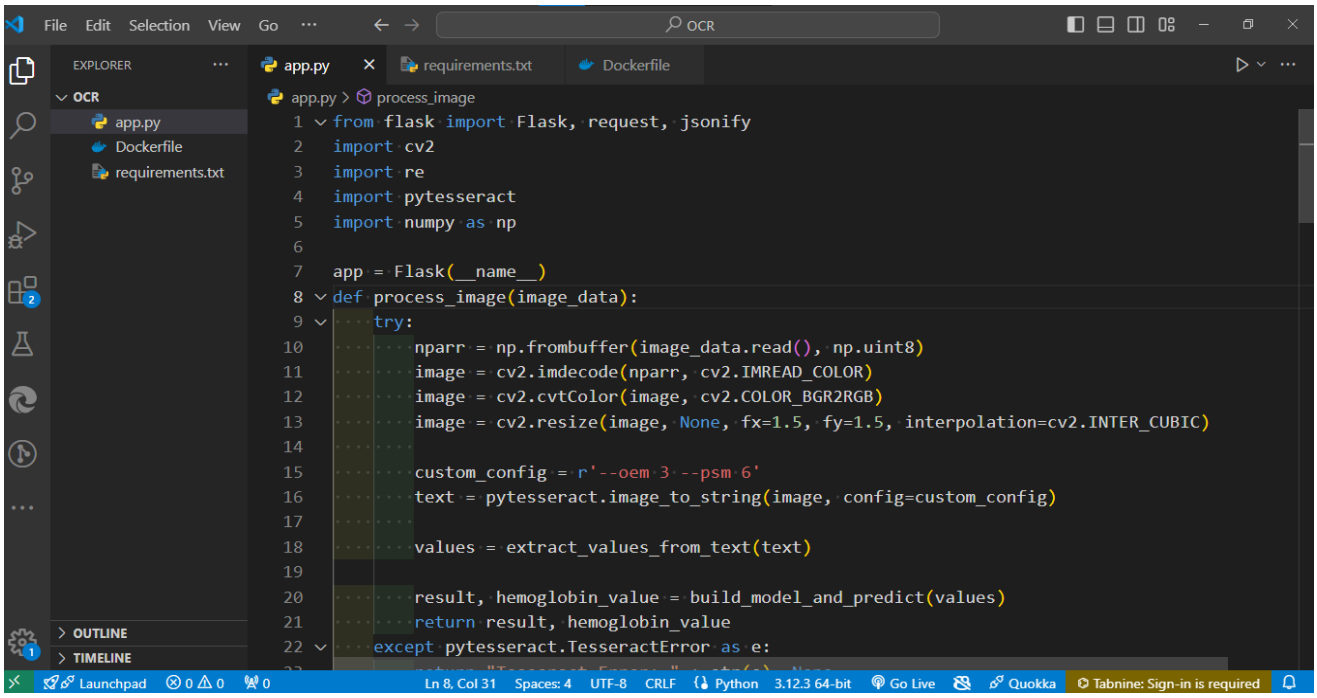


Fig 90: OCR connection with Flask (1)

```
    except Exception as e:
        return "An error occurred: " + str(e), None

def extract_values_from_text(text):
    values = {}
    lines = text.split('\n')
    for line in lines:
        if 'Platelet' in line:
            platelet_value = re.search(r"\d+\.\d+", line)
            values['Platelet'] = platelet_value.group() if platelet_value else None
        elif 'RBC' in line:
            rbc_value = re.search(r"\d+\.\d+", line)
            values['RBC'] = rbc_value.group() if rbc_value else None
        elif 'WBC' in line:
            wbc_value = re.search(r"\d+\.\d+", line)
            values['WBC'] = wbc_value.group() if wbc_value else None
        elif 'Hemoglobin' in line:
            hemoglobin_value = re.search(r"\d+\.\d+", line)
            values['Hemoglobin'] = hemoglobin_value.group() if hemoglobin_value else None
    return values

```

Fig 91: OCR connection with Flask (2)

```

def build_model_and_predict(values):
    platelet_min = 150
    platelet_max = 400
    rbc_min = 4.40
    rbc_max = 6.00
    wbc_min = 4.00
    wbc_max = 11.00
    hemoglobin_min = 13.5
    hemoglobin_max = 18.0

    platelet = float(values.get('Platelet', 0)) if values.get('Platelet') is not None else 0
    rbc = float(values.get('RBC', 0)) if values.get('RBC') is not None else 0
    wbc = float(values.get('WBC', 0)) if values.get('WBC') is not None else 0
    hemoglobin = float(values.get('Hemoglobin', 0)) if values.get('Hemoglobin') is not None else 0

    result = "NORMAL" if (platelet_min <= platelet <= platelet_max and
                          rbc_min <= rbc <= rbc_max and
                          wbc_min <= wbc <= wbc_max and
                          hemoglobin_min <= hemoglobin <= hemoglobin_max) else "ABNORMAL"

```

Fig 92: OCR connection with Flask (3)

```
    if result == "NORMAL":
        return "NORMAL", hemoglobin

    return result, None
@app.route('/', methods=['POST'])
def upload_image():
    if 'image' not in request.files:
        return jsonify({'error': 'No image part'}), 400
    file = request.files['image']
    if file.filename == '':
        return jsonify({'error': 'No selected image'}), 400
    if file:
        result, hemoglobin_value = process_image(file)
        if hemoglobin_value is not None:
            return jsonify({'result': result, 'Hemoglobin': hemoglobin_value})
        else:
            return jsonify({'result': result})
    return jsonify({'error': 'An error occurred while processing the image'}), 500

if __name__ == '__main__':
    app.run(debug=True, host='0.0.0.0')
```

Fig 93: OCR connection with Flask (4)

The following dependencies are used in the project:

- **Flask:** A framework for Python used to create applications.
- **OpenCV:** An open-source computer vision and machine learning software library used for image processing tasks.
- **Pytesseract:** A Python wrapper for Google's Tesseract-OCR Engine, which is used for extracting text from images.
- **NumPy:** A powerful library for numerical computing in Python, used for array manipulation and mathematical operations

To start the Flask server running this script `app.py`.

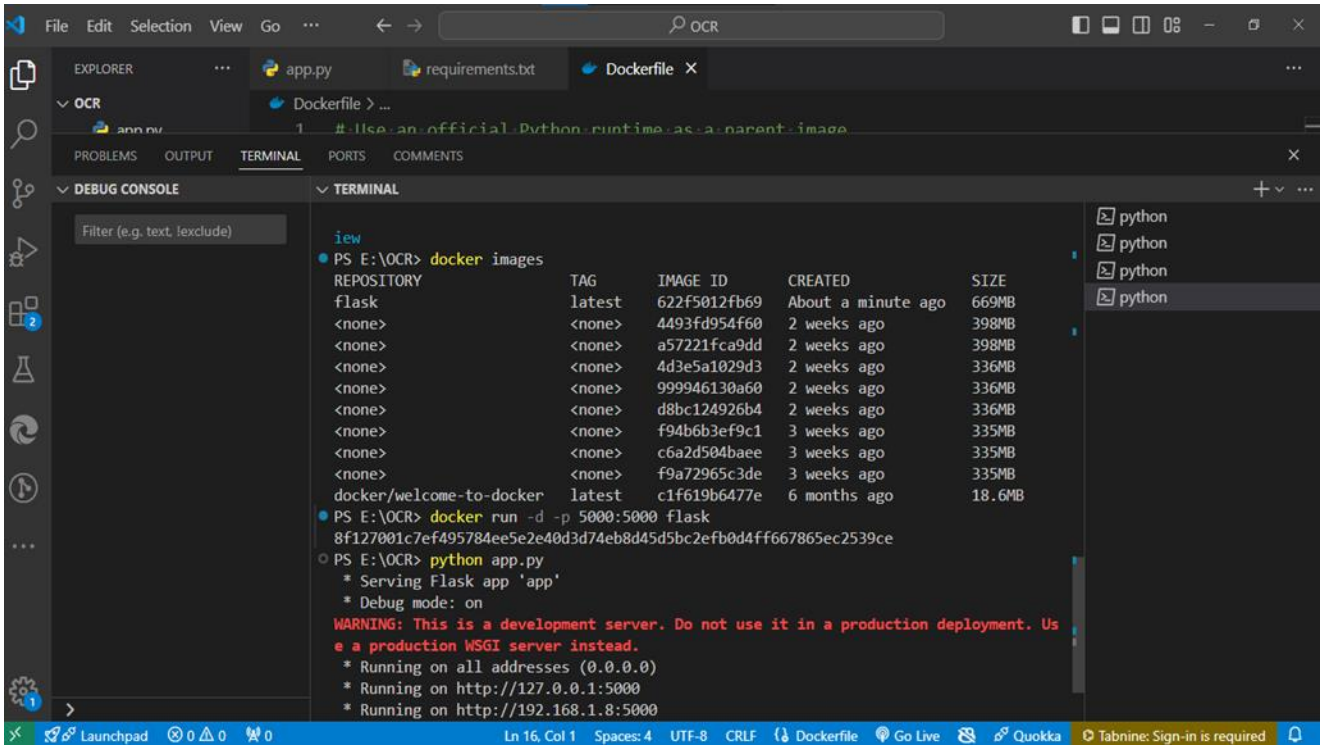


Fig 94: Flask server running

- This indicates that Flask application is running and listening for incoming requests on port 5000.
- After running get the IP address "192.168.1.8:5000"

- Send a POST request to the root endpoint (`/`) with an image file attached as form data.

- Receive analysis results in JSON format, including the classification of blood parameters and, if available, the extracted Hemoglobin level.

- Endpoint
- POST /:
- Accepts an image file (JPEG, PNG, etc.) as form data.
- Returns a JSON response containing the analysis results.

Testing IP address Locally

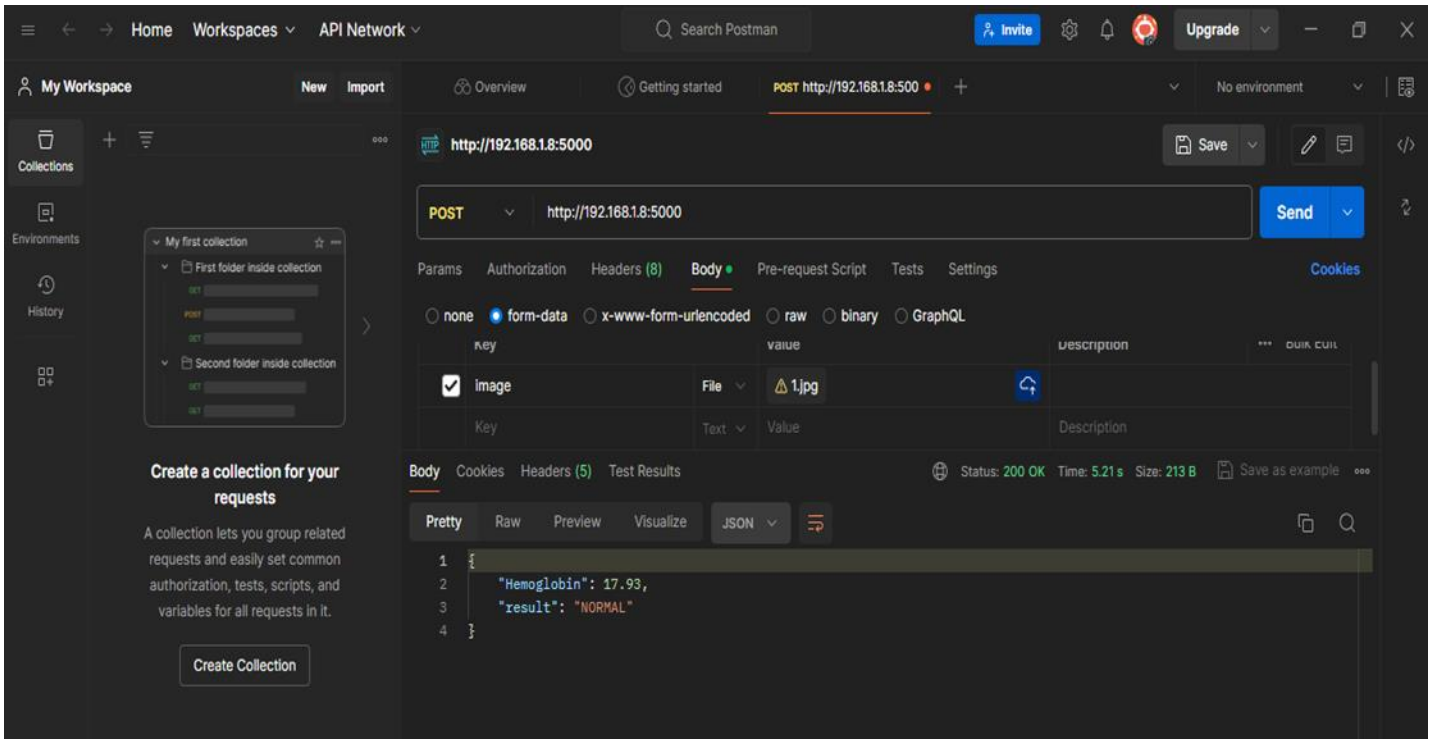


Fig 95: Testing IP address Locally

Reason for Using Docker

Error Encountered

During the initial deployment attempts, the following error was encountered:

pytesseract.pytesseract.TesseractNotFoundError: tesseract is not installed.

Build and run the docker image

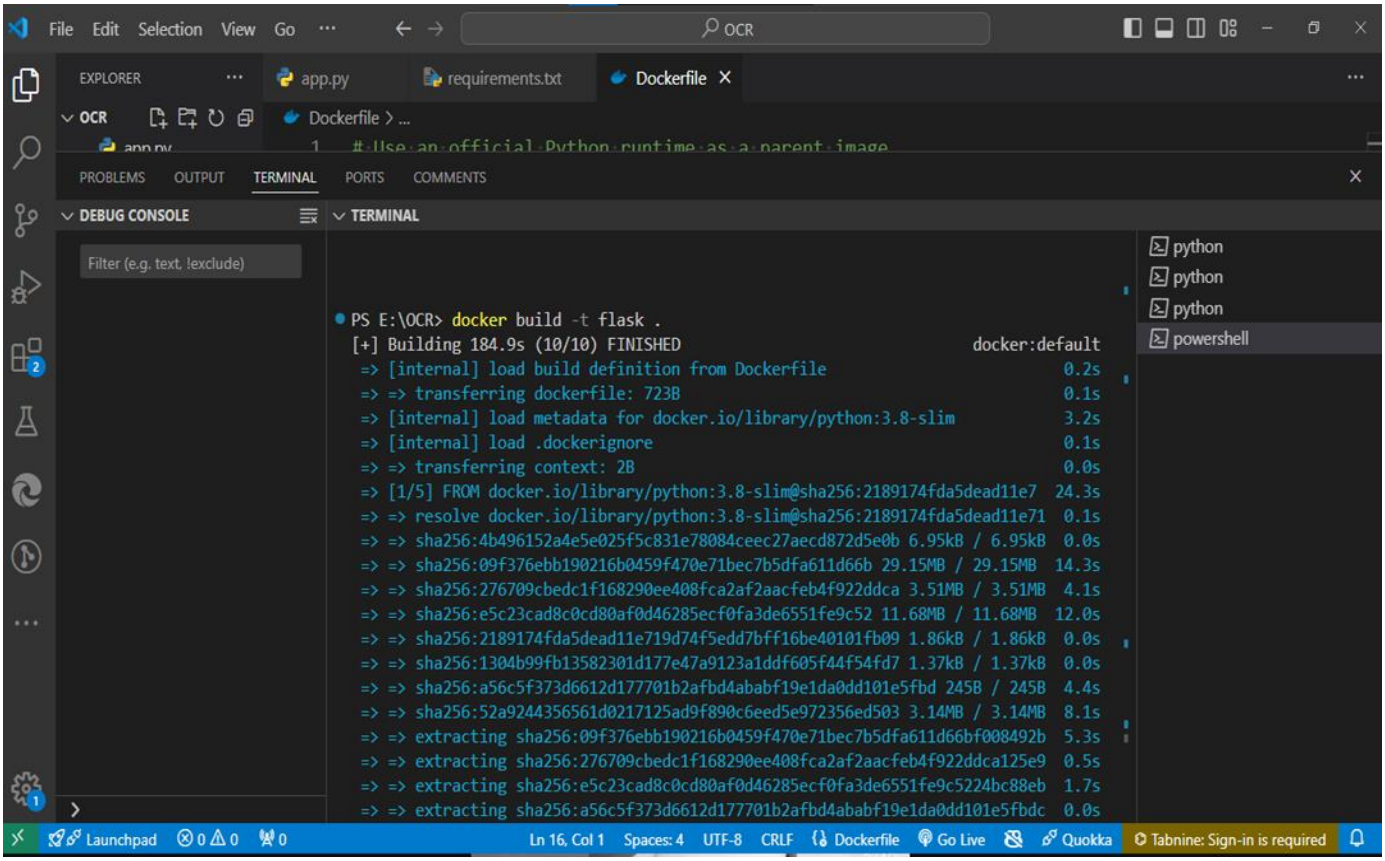


Fig 96: Build and run the docker image

- Running the command `docker build -t flask` will build a Docker image tagged as "flask" using the Dockerfile in the current directory (`.`). Let's break down what each part of this command does:[6]

- `docker build`: This command tells Docker to build an image.

- `-t flask`: This option sets the tag of the image to "flask". Tags are like version labels for Docker images.

- `.` : This specifies the build context, which is the location of the Dockerfile and any other files needed during the build process. Here, `.` indicates the current directory.
- So, when you run this command, Docker will look for a Dockerfile in the current directory, use it to build an image, and tag that image as "flask".
- Once the build is complete, you can use the newly created Docker image to run containers,

- deploy your Flask application, and more.

Dockerization

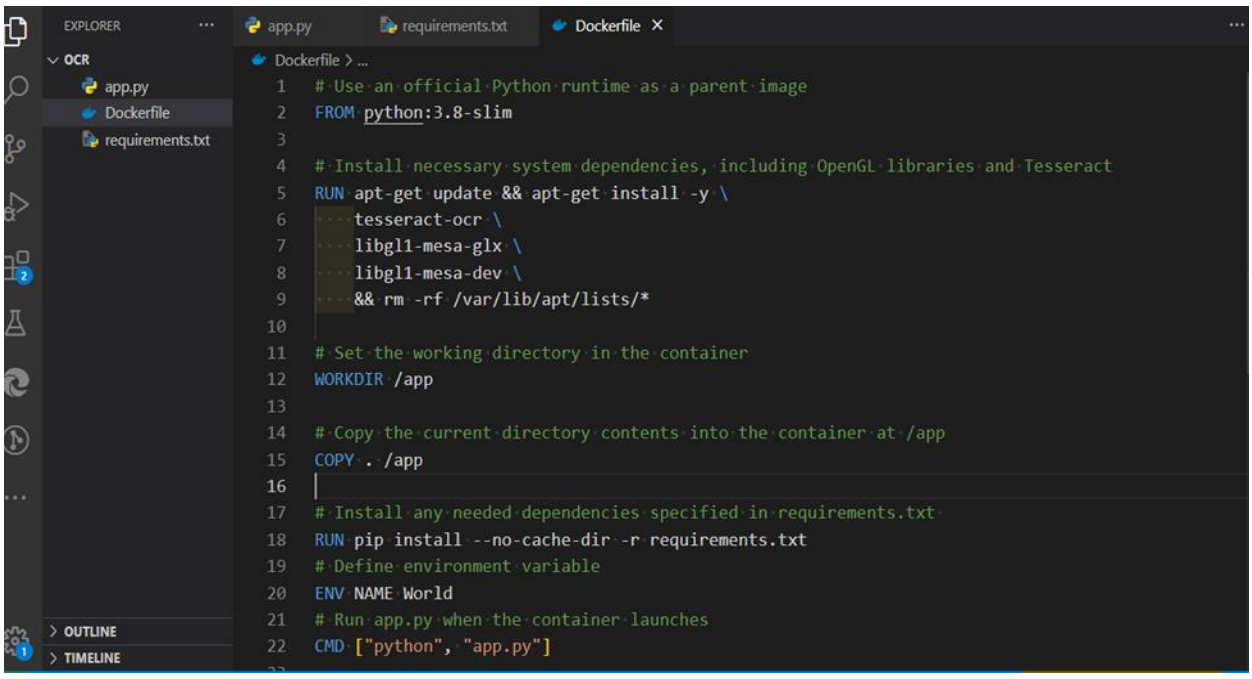


Fig 97: Docker file

Resolution

To resolve this issue, the decision was made to Dockerize the application. Docker provides a way to encapsulate the application and its dependencies into a container, ensuring consistent behavior across different environments.

Dockerization

The Dockerfile was configured to install Tesseract and other necessary dependencies within the Docker container. This ensured that the application had access to the required binaries, including Tesseract, during runtime.

This Dockerfile does the following:

1. Uses the official Python 3.8 slim image as the base image.
2. Installs necessary system dependencies like Tesseract and OpenGL libraries.
3. Sets the working directory inside the container to `/app`.
4. Copies the current directory's contents into the `/app` directory of the container.
5. Installs the Python dependencies listed in `requirements.txt`.
6. Sets an environment variable `NAME` to "World".
7. Specifies the command to run (`app.py`) when the container launches using `CMD`.

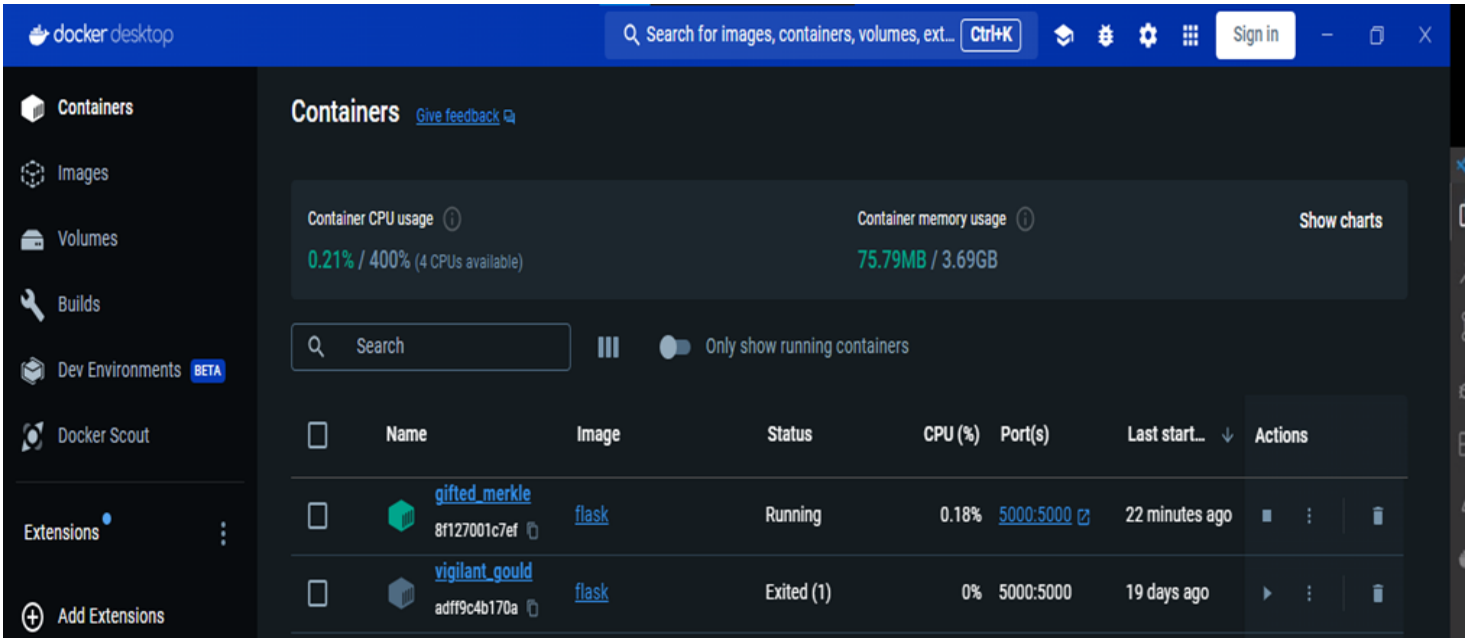


Fig 98: Containers

Requirements.txt the requirements.txt file based on the dependencies

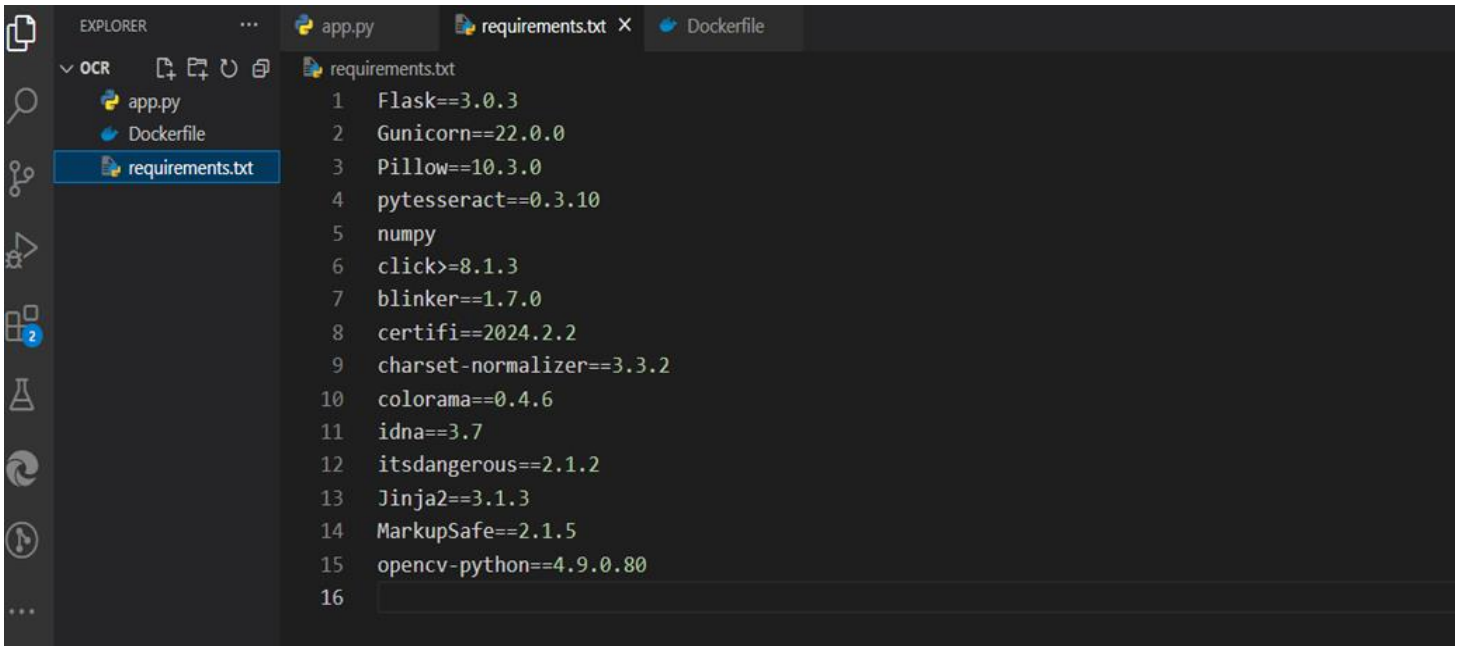


Fig 99: Requirements.txt

You can use this `requirements.txt` file to install the dependencies for your Flask

application using `pip`:

`pip install -r requirements.txt` This command will install all the dependencies listed in the `requirements.txt` file.

Dockerization Steps and Deployment Process on the Render Server

1. **Create Dockerfile**: Define the Docker image configuration in a Dockerfile. Include instructions to install Tesseract and other dependencies required by the application.
2. **Build Docker Image**: Use the Dockerfile to build the Docker image. This process will create a containerized version of the OCR application with all dependencies included.
3. **Test Locally**: Test the Dockerized application locally to ensure that it runs without errors.
4. **Push to Repository**: Push the Docker image to a container registry repository such as GitHub Container Registry.
5. **Deploy on Platform**: Deploy the Dockerized application on a platform that supports Docker container deployments. Configure the deployment

settings and provide any necessary environment variables such as Render.[8]

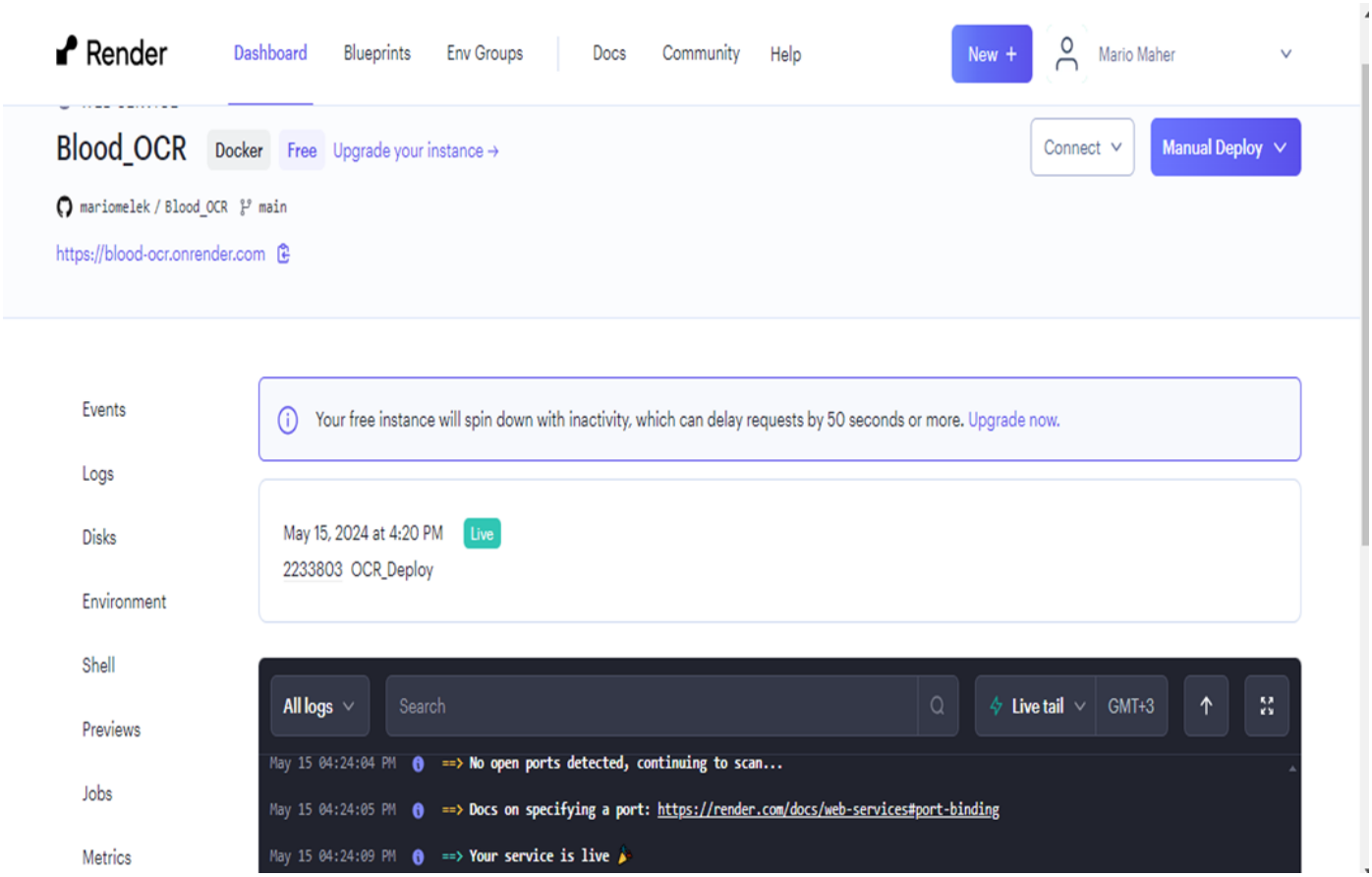


Fig 100: Deploy on render

API Testing After Deployment

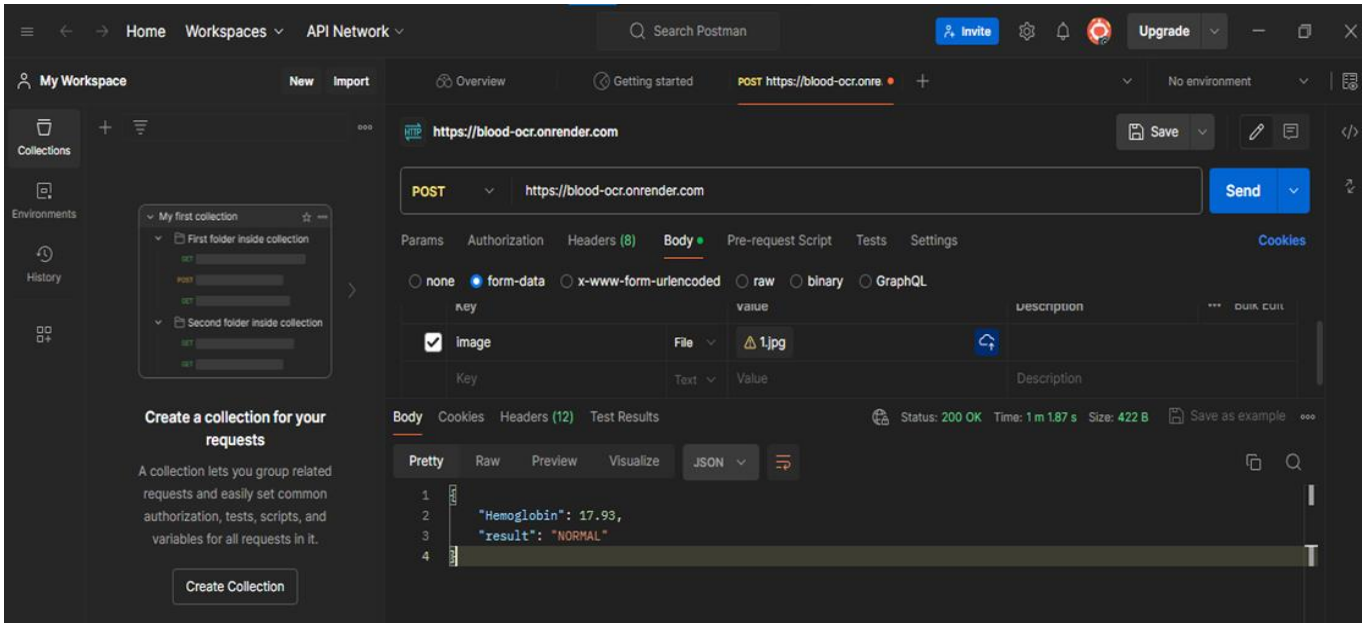


Fig 101: API Testing After Deployment

HTTP response with a status code of "200 OK"

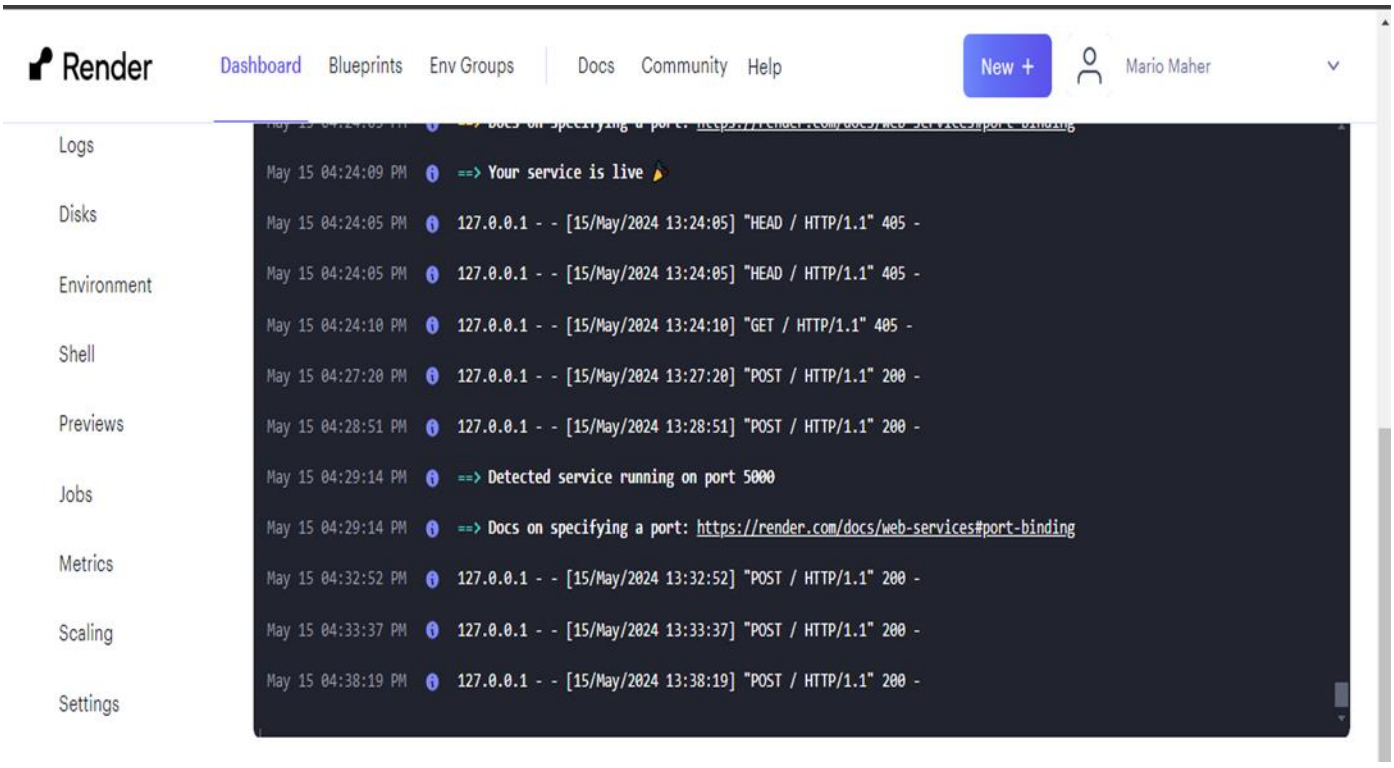


Fig 102: HTTP response with "200 OK"

4.19 Chatbot with connection deep learning and flutter app by Flask

Overview

This project aims to develop a chatbot using Flask that can predict diseases based on user symptoms and provide appropriate precautions.
The chatbot uses a TF-IDF vectorizer and a label encoder to process and classify input text.

Components

The code imports necessary libraries, including Flask for web framework, NLTK for natural language processing, Pandas for data manipulation, and scikit-learn

```
from flask import Flask, request, jsonify
import numpy as np
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import LabelEncoder
from sklearn.metrics.pairwise import cosine_similarity
import pandas as pd
import string
import nltk
from nltk.tokenize import word_tokenize
from nltk.corpus import stopwords
from nltk.stem import WordNetLemmatizer
```

Fig 103: Flask chatbot

NLTK Data Download

The NLTK data is downloaded to ensure that stopwords, tokenizers, and lemmatizers are available for text preprocessing.

```
nltk.download('stopwords')
nltk.download('punkt')
nltk.download('wordnet')
```

Fig 104: NLTK Data Download

Loading and Preprocessing Datasets

Two datasets are loaded using Pandas:

- Symptom2Disease.csv for mapping symptoms to diseases.
- symptom_precaution.csv for providing precautions for each disease.

The TF-IDF vectorizer is used to transform the text data into feature vectors, and the label encoder encodes the disease labels.

Purpose of the Code

The primary goal of this code is to build an interactive chatbot that can:

- Identify diseases based on user-described symptoms.
- Provide corresponding precautions for the identified disease.

Key Functions

- predict_disease(user_input): Predicts the disease based on user input using cosine similarity.
- get_actions(matched_disease): Retrieves precautions for the matched disease.
- chatbot_response(user_input, user_id): Generates a response based on user input and current state.

- Running the command `docker build -t flask` will build a Docker image tagged
  - as "flask" using the Dockerfile in the current directory (`.`). Let's break down what
  - each part of this command does:

- `docker build`: This command tells Docker to build an image.

- `-t flask`: This option sets the tag of the image to "flask". Tags are like version labels for Docker images.

- `.` : This specifies the build context, which is the location of the Dockerfile and any other files needed during the build process. Here, `.` indicates the current directory.
- So, when you run this command, Docker will look for a Dockerfile in the current directory, use it to build an image, and tag that image as "flask".

- Once the build is complete, you can use the newly created Docker image to run containers,

- deploy your Flask application, and more.

Dockerization

```
# Use the official Python image.
FROM python:3.10-slim

# Set the working directory to /app
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY . /app

# Install system dependencies and build-essential for nltk
RUN apt-get update && apt-get install -y \
    build-essential \
    && rm -rf /var/lib/apt/lists/*

# Install any needed packages specified in requirements.txt
RUN pip install --no-cache-dir -r requirements.txt

# Install NLTK and download required data
RUN pip install nltk
RUN python -m nltk.downloader stopwords punkt wordnet

# Make port 80 available to the world outside this container
EXPOSE 80

# Define environment variable
ENV NAME World

# Run app.py when the container launches
CMD ["python", "app.py"]
```

Fig 105: Dockerization

Resolution

To resolve this issue, the decision was made to Dockerize the application. Docker provides a way to encapsulate the application and its dependencies into a container, ensuring consistent behavior across different environments.

Dockerization

The Dockerfile was configured to install Tesseract and other necessary

dependencies within the Docker container. This ensured that the application had access to the required binaries, including Tesseract, during runtime.

This Dockerfile does the following:

1. Uses the official Python 3.8 slim image as the base image.
2. Installs necessary system dependencies like Tesseract and OpenGL libraries.
3. Sets the working directory inside the container to `/app`.
4. Copies the current directory's contents into the `/app` directory of the container.
5. Installs the Python dependencies listed in `requirements.txt`.
6. Sets an environment variable `NAME` to "World".
7. Specifies the command to run (`app.py`) when the container launches using `CMD`.

Requirements.txt the requirements.txt file based on the dependencies

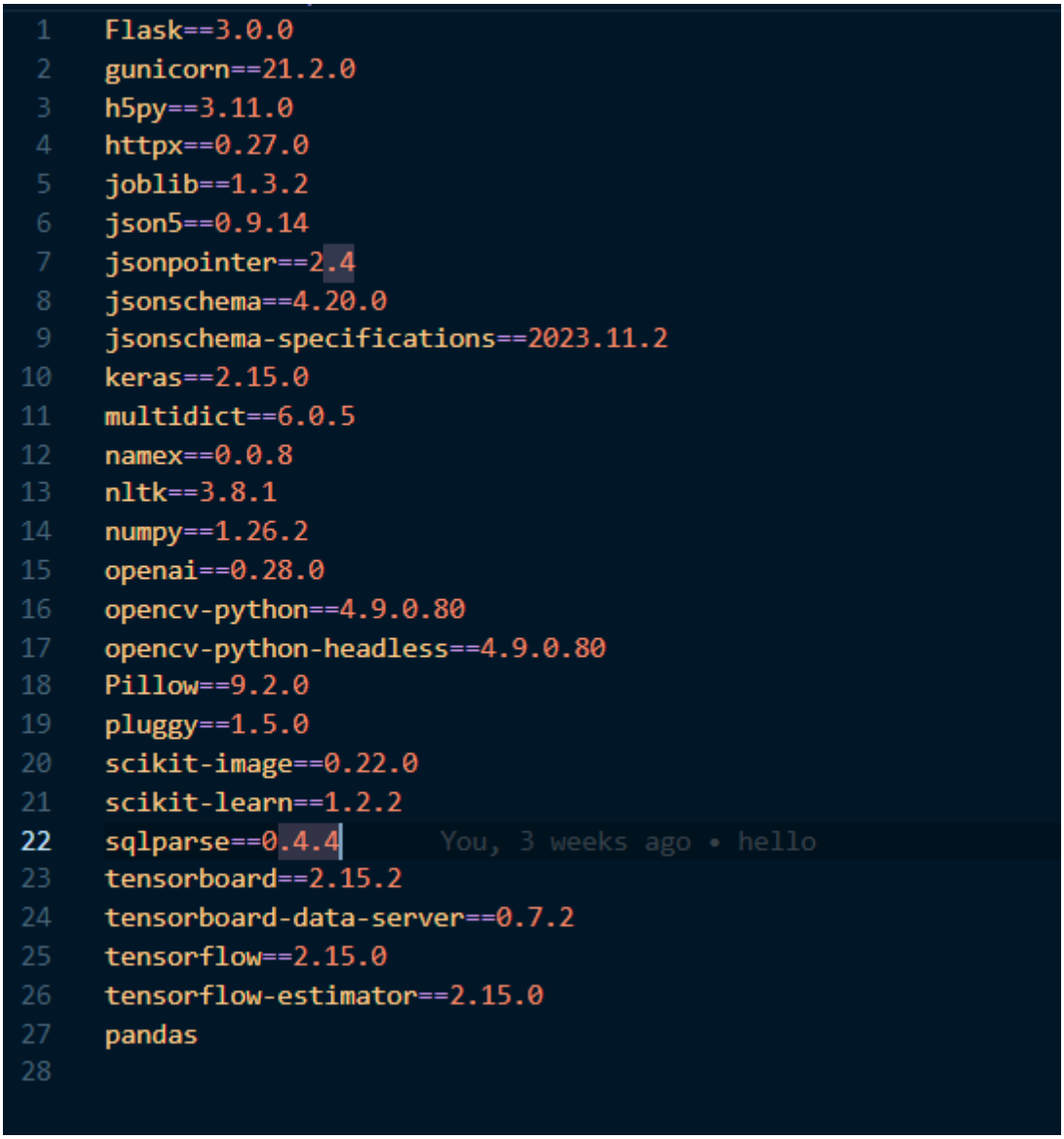


Fig 106: Requirements.txt

You can use this `requirements.txt` file to install the dependencies for your Flask application using `pip`:

`pip install -r requirements.txt` This command will install all the dependencies listed in the `requirements.txt` file.

6. **Create Dockerfile**: Define the Docker image configuration in a Dockerfile. Include instructions to install Tesseract and other dependencies required by the application.
7. **Build Docker Image**: Use the Dockerfile to build the Docker image. This process will create a containerized version of the OCR application with all dependencies included.
8. **Test Locally**: Test the Dockerized application locally to ensure that it runs without errors.
9. **Push to Repository**: Push the Docker image to a container registry repository such as GitHub Container Registry.
10. **Deploy on Platform**: Deploy the Dockerized application on a platform that supports Docker container deployments. Configure the deployment settings and provide any necessary environment variables such as Render.

4.20 Connection by Django (Check)

Overview

The PredictView class is a Django REST framework view that handles the prediction of a certain condition (presumably a disease or medical condition) based on several input features. It leverages a pre-trained[7]

XGBoost model stored in a joblib file to make predictions.

This documentation provides a detailed explanation of the class, its methods, and the workflow.

Imports and Dependencies

The class relies on Django REST framework's CreateAPIView and the joblib library for loading the pre-trained model.

```
from rest_framework.response import Response
from rest_framework import status
from rest_framework.generics import CreateAPIView
import joblib
import numpy as np
```

Fig 107: import (Django)

Validate Input Data:

- Uses the serializer to validate the incoming data. Raises an exception if validation fails.

```
serializer = self.get_serializer(data=request.data)
serializer.is_valid(raise_exception=True)
input_data = serializer.validated_data
```

Fig 108: validation (Django)

Purpose

The purpose of the PredictView class is to provide an API endpoint that accepts medical-related featuresand returns a prediction based on a pre-trained XGBoost model. This can be used in medical applications to predict the likelihood of a condition based on patient data.

Workflow

1. **Model Loading**:
   - The pre-trained XGBoost model is loaded from a specified path using joblib.load.

2. **Input Validation**:
   - The incoming data is validated against the PredictSerializer to ensure all required features are present and correctly formatted.

3. **Feature Extraction**:
   - The validated data is extracted and organized into a format suitable for the model's input.

4. **Prediction**:
   - The model makes a prediction using the extracted features.

5. **Response Construction**:
   - The prediction result is formatted into a JSON response and returned to the client.

Key Functions

- **create(self, request, *args, **kwargs)**:
  - Main function that handles the POST request, validates input data, prepares features, makes predictions, and returns the result.

Error Handling

- **Validation Errors**:
  - If the input data is invalid, the serializer raises an exception which is handled by

    the raise_exception=True parameter. This ensures the client receives appropriate error messages.

Chapter 5:
Conclusion & Future work

5.1 Conclusions

Finally, we managed to implement a comprehensive mobile application that serves both users and hospitals by providing a suite of essential features for blood donation and management. The application allows users to search for the nearest hospitals that have the required blood type. It presents a list of the three closest hospitals along with the available quantity of the specific blood type at each location. Once a user selects a hospital, a map view is provided to guide them directly to the hospital's location.

In addition to helping users find the right blood type, the application assists medical staff in determining whether a person is eligible to donate blood. It includes a CBC (Complete Blood Count) test feature, which provides analysis results to evaluate the donor's health. An artificial intelligence-powered bot is integrated into the system to analyze messages from users describing their symptoms. The bot predicts potential diseases that might prevent the user from donating blood and offers valuable advice on managing these conditions.

The application also has a robust notification system that sends and receives alerts to all registered users. These notifications contain crucial information about blood type shortages, ensuring that the community is promptly informed when there is a need for specific blood types.

Moreover, the app provides a QR code service for hospitals, simplifying the creation of barcodes to retrieve data about blood types. Users can utilize barcode technology to scan a barcode and add blood type information to the system or search for a blood type by its serial number. The application also allows users to remove a blood type from the system, either by scanning its barcode or searching for it using its serial number, provided this is done before the blood type becomes invalid after 35 days.

5.2 Future Work

1. collect real data from hospitals and enable their utilization to enhance healthcare outcomes

2. deploy app in app store and make it available to use.

3. Continuous Improvement: Gather feedback from users and iterate on the app to continuously improve its features and performance.

4. Security Testing: Security testing evaluates the application's resilience to potential threats and vulnerabilities. Testers assess the effectiveness of security controls and validate compliance with security standards.

5. Maintenance and Support: After deployment, testing continues during the maintenance phase to address any issues that arise post-release. Testers collaborate with developers to troubleshoot and resolve reported defects.

Chapter 6: Reference

6.1Reference

[1] OCR. (2024). Example: Implementing OCR for Invoice Processing. [Online]. Available: https://www.affinda.com/tech-ai/machine-learning-ocr

[2] Pub dev. (2024). Example: Publishing a Dart Package. [Online]. Available: https://pub.dev/

[3] Flutter. (2024). Example: Developing Mobile Applications with Flutter. [Online]. Available: https://docs.flutter.dev/

[4] Firebase. (2024). Example: Integrating Firebase Authentication in a Flutter App. [Online]. Available: https://firebase.flutter.dev/

[5] Flask. (2024). Example: Creating a Simple Microblog with Flask. [Online]. Available: https://flask.palletsprojects.com/en/3.0.x/tutorial/

[6] Docker. (2024). Example: Containerizing a Python Application. [Online]. Available: https://docs.docker.com/manuals/

[7] Django. (2024). Example: Building a Blog Application with Django. [Online]. Available: https://www.djangoproject.com/

[8] Render deployment. (2024). Example: Deploying a Node.js Application using Docker on Render. [Online]. Available: https://docs.render.com/docker

Articles :

[9] Chatbot for disease prediction and treatment recommendation using machine learning

Rohit Binu Mathew, Sandra Varghese, Sera Elsa Joy, Swanthana Susan Alex

2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), 851-856, 2019

https://scholar.google.com/scholar?q=related:qJAJqw7wDkEJ:scholar.google.com/&hl=en&as_sdt=0,5#d=gs_qabs&t=1718091331682&u=%23p%3D_Ye79BBJ1xYJ

[10] Philip Indra Prayitno, Reinhart Perbowo Pujo Leksono, Fernando Chai, Richard Aldy, Widodo Budiharto

2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI) 1, 62-67, 2021

Health chatbot using natural language processing for disease prediction and treatment

https://scholar.google.com/scholar?q=related:hCczxj5_gFYJ:scholar.google.com/&hl=en&as_sdt=0,5#d=gs_qabs&t=1718090960723&u=%23p%3DqJAJqw7wDkEJ