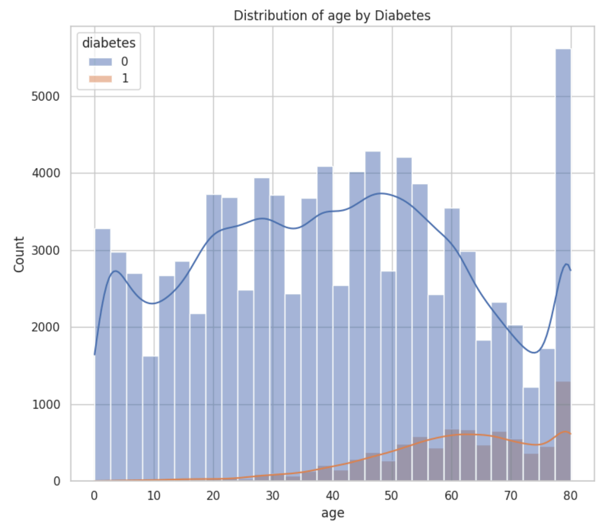
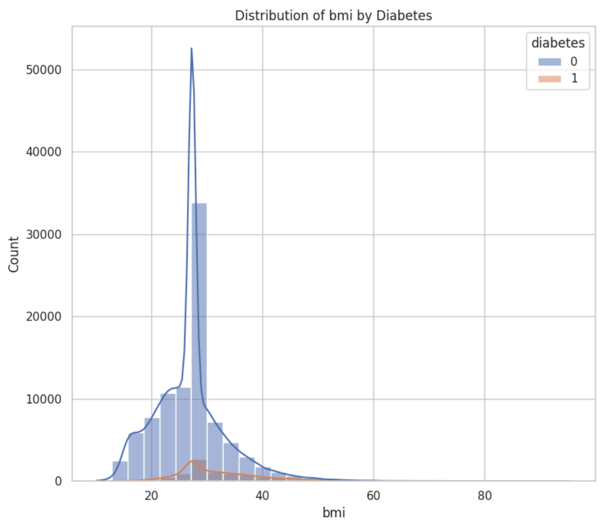
Diabetes Predictive Modelling

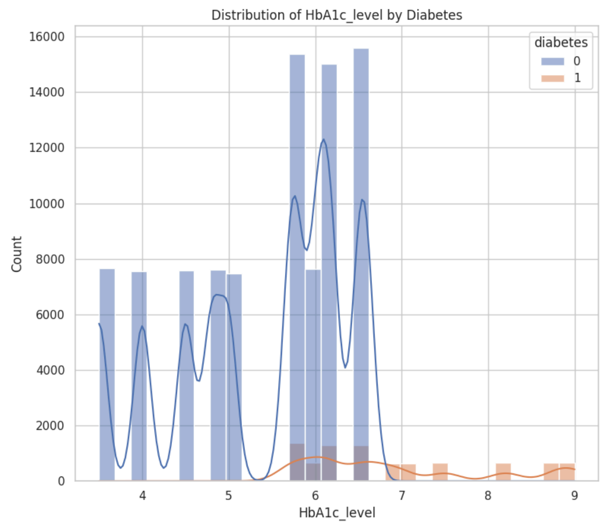
Our report covers the building of a Diabetes predictive model (as well as Hypertension & Heart Disease) based on a Kaggle diabetes prediction dataset.

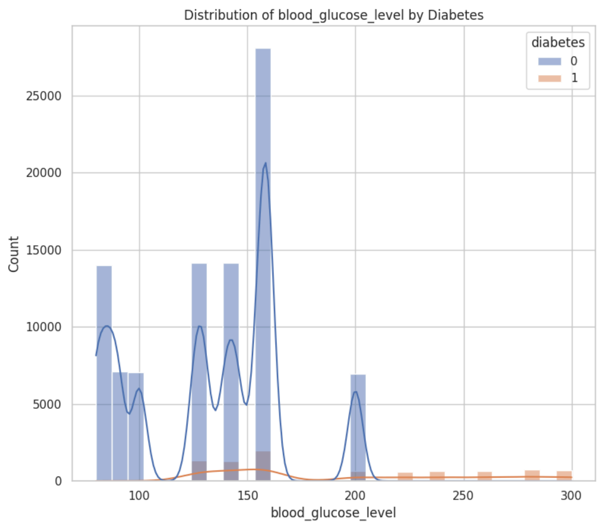
The Kaggle Diabetes dataset contains 8 features, 2 categorical (gender, smoking_history) and 6 numerical (age, hypertension, heart_disease, bmi, HbA1c_level, blood_glucose level). It is dense and complete with no missing values, so no imputation is required. It is however imbalanced, as is natural for there to be less diabetes than non-diabetes people, and only 8.5% of the population has diabetes.

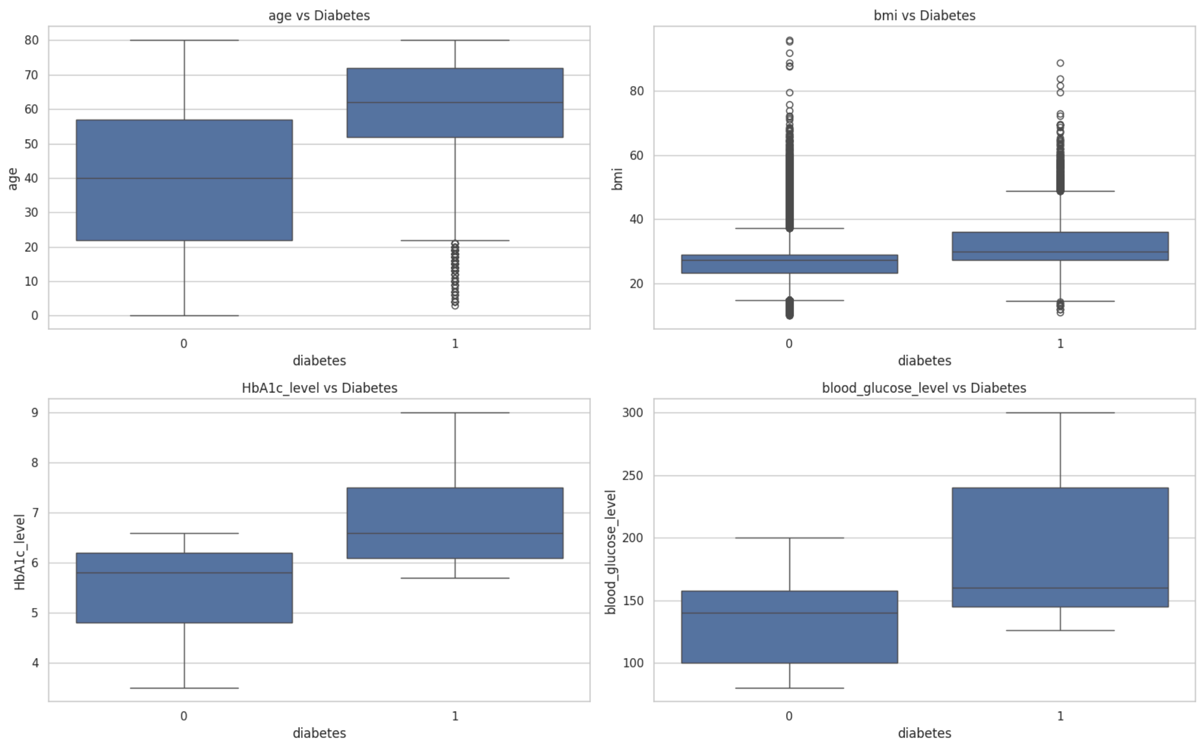
Exploratory Data Analysis



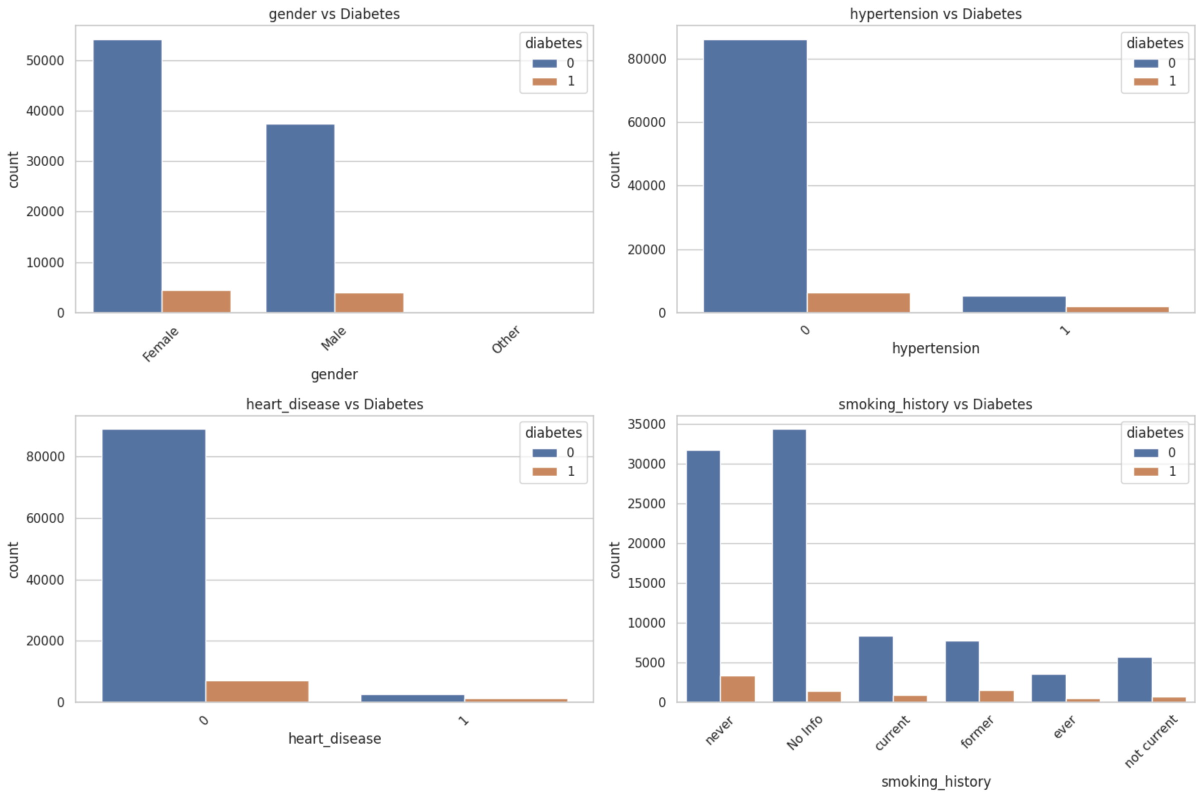




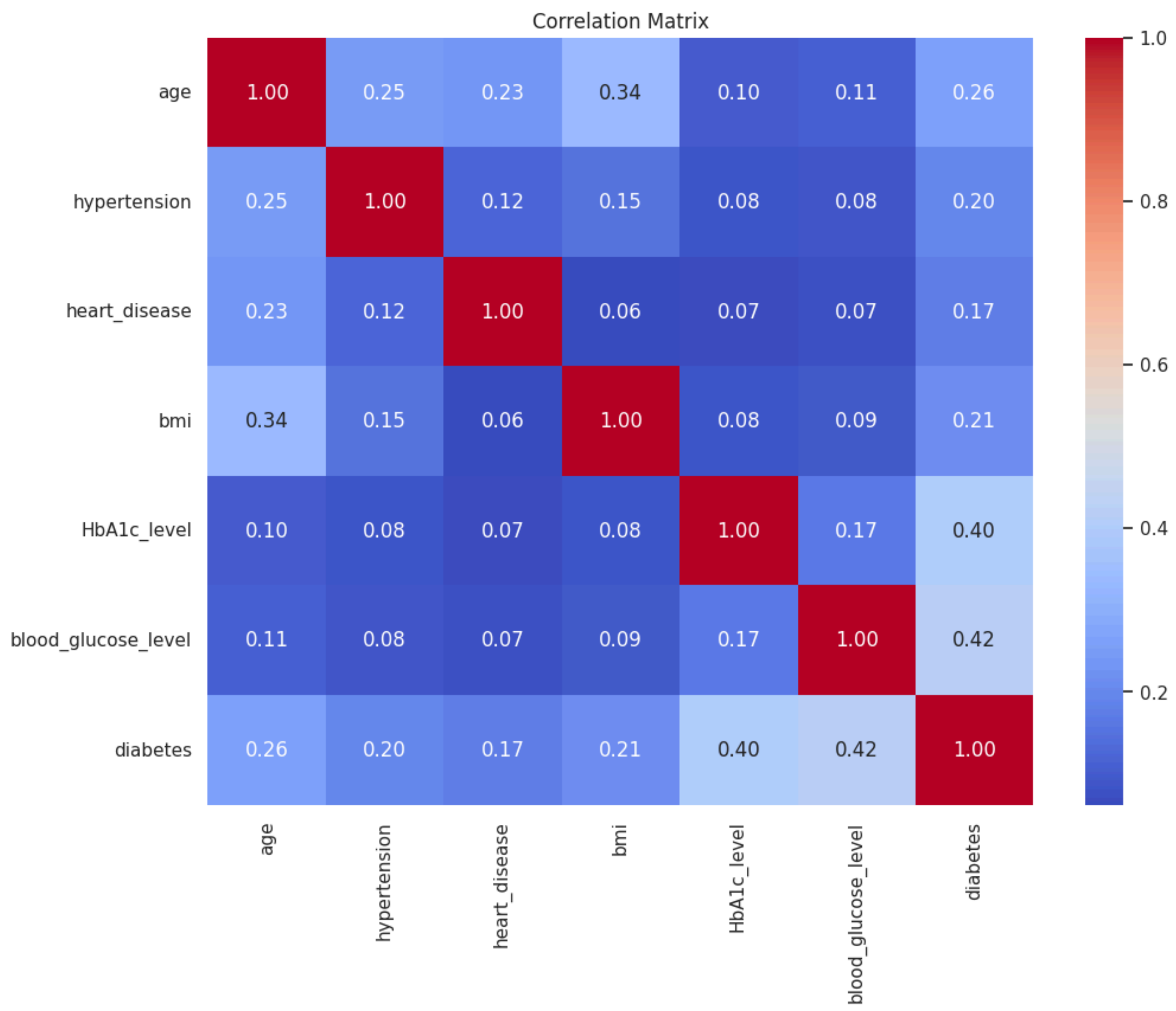




For the numerical non-binary features, we plotted their distributions and found that diabetes patients are generally older and have higher BMI, HbA1c levels and blood glucose levels.

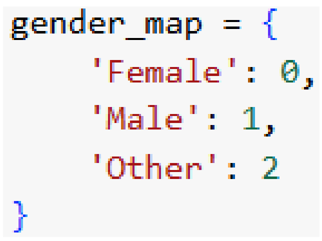


For the categorical and binary features, we also plotted barcharts and found that a much higher proportion of those with hypertension have diabetes as compared to those without. The other features have no significant relationships with diabetes.

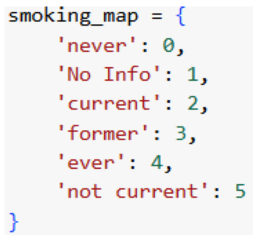


We also plotted a correlation matrix of the numerical features and the target column (diabetes) against one another and found that there are moderate positive correlations of HbA1c levels and blood glucose levels with diabetes and weak positive correlations of age, hypertension, heart disease and BMI with diabetes.

Dataset Preprocessing



```
gender_map = {
    'Female': 0,
    'Male': 1,
    'Other': 2
}
```



```
smoking_map = {
    'never': 0,
    'No Info': 1,
    'current': 2,
    'former': 3,
    'ever': 4,
    'not current': 5
}
```

Before we put the data into our models, we first encoded the categorical features so that they are in the appropriate input format to be fed into the models. The mappings are as shown above. We also separated the features and the target columns, and then split the data into training and test sets of 80% and 20% respectively. The test sets are hidden from the model during the training phase. Recognising the major imbalance in the diabetes classes, we also calculated the scale factor from the positive class (diabetes) to the negative class (non-diabetes) which came out to be 10.7647, and incorporated it into our models to manage the imbalance.

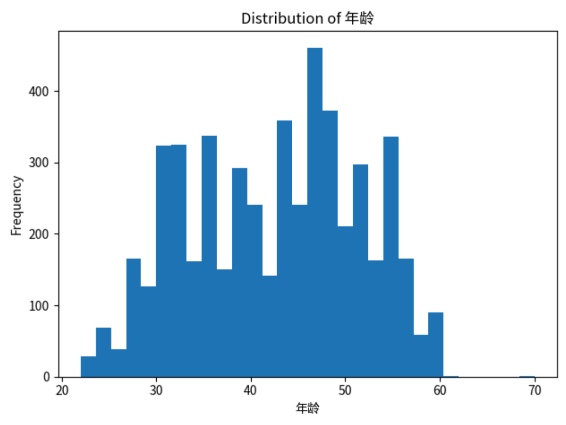
Incorporating Other Datasets

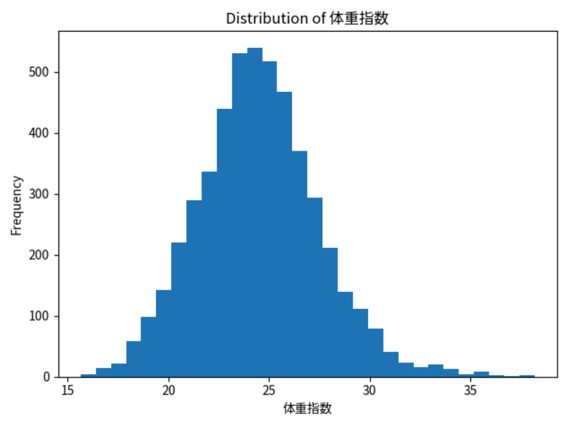
We also considered using other datasets for model training, but this is not a practical solution. Other diabetes datasets we found do not perfectly overlap with the main Kaggle dataset, so if we just added them to our training dataset there would be huge holes where some columns are just missing for thousands of datapoints. Coincidentally the XGBoost trees we use for our main model can actually deal with missing values in training by learning a heuristic to which leaf node data points with missing values should be sent when splitting on a specific feature, however in all likelihood this would not have been stable and just degraded model training. Especially since the other available datasets almost never provide glucose and HbA1c values, which are very important for predicting the diabetes label.

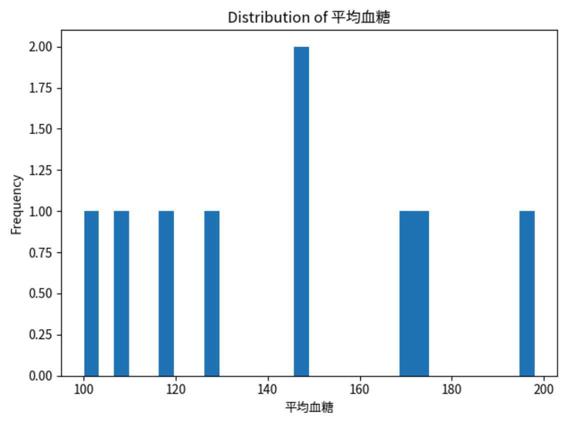
Reformatting and Analysing the Original Hospital Data

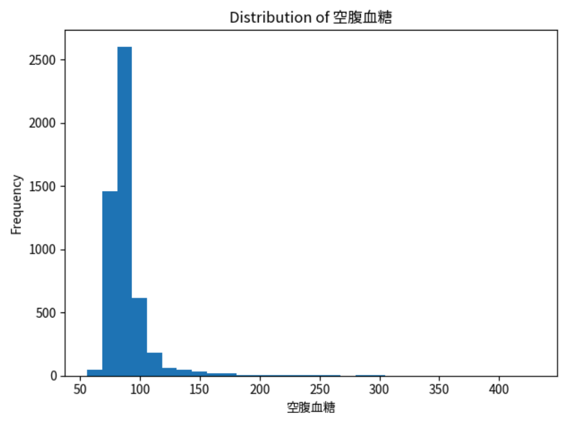
We also wasted tons of time trying to turn the Physical_Examination_Data.xlsx file into one with the same feature format as the Kaggle dataset. You told us the data on which you're grading us would resemble this file, so we were hoping to find information that would help us to train a better model. But the only thing this showed me (Justin) is that the grading is going to be ridiculous. The Physical_Examination_Data.xlsx file does not contain data about smoking, so this is one feature that's missing. It's already impossible to get this hospital data into the same shape as the kaggle dataset and I have no idea how you teachers want to solve that. It also has no feature for heart disease. For my reformatting I considered systematically scanning the handwritten diagnosis in column 体检结论 for heart disease to reconstruct the feature that way but since the data quality is already bad I decided that this is not worth my time. Even if had found a strategy to reconstruct a heart disease label, the original Kaggle dataset does not provide information about what exact disease would be considered heart disease. So this is another feature that's missing, because even if we reconstruct it it's based on an arbitrary definition of what heart disease is and we have no way of knowing whether it's the same criterion for heart disease used in the original Kaggle dataset, so any reconstructed label would certainly have a very different distribution. Again, no idea how you want to deal with that problem.
Lastly, and this is the most important reason why Physical_Examination_Data.xlsx is unusable for the model, only 9 people have test values for average glucose level and HbA1c values. More than 5000 patients did glucose tests while fasting (空腹血糖) but this is not the same thing as average glucose (平均血糖) over a longer period of time and as we will see the distribution is extremely different. By the way, the unit of measurement for the blood glucose is different between the two datasets. Physical_Examination_Data uses mmol/L, whereas the Kaggle dataset probably uses mg/dL, since its glucose values are about 18 times higher. I hope you remember to take that into account when you give us the grading dataset.
That is the last problem and why I expect our and the models from all our class mates to bomb on your grading dataset, the features are just distributed differently. The patients are generally much younger, the BMI of the Kaggle dataset has a huge spike and higher average compared to the Physical_Examination_Data, the patients are overwhelmingly male instead of more balanced like in Kaggle and lastly there are simply far fewer people with diabetes. As far as we were able to read from the doctor's diagnosis only around 30 people are definitely diagnosed with diabetes. Another 70 people are "at high risk" but even when

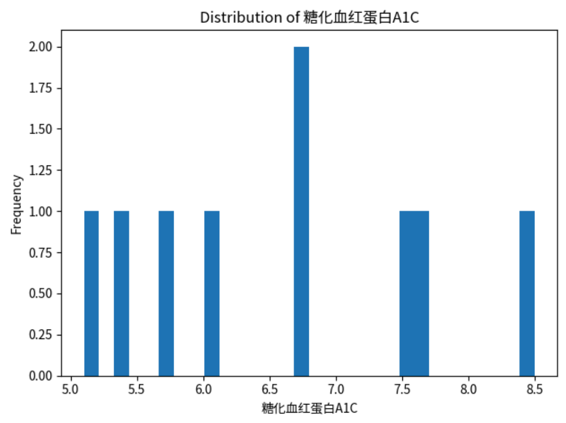
counting these at-risk patients as fully diabetic only 2% of people have diabetes, as opposed to 8,5% in the Kaggle dataset.
Also the general fact that this dataset was collected on a specific ethnicity at a specific hospital with specific machines, as opposed to the Kaggle dataset, which's origin is to my knowledge completely unknown.

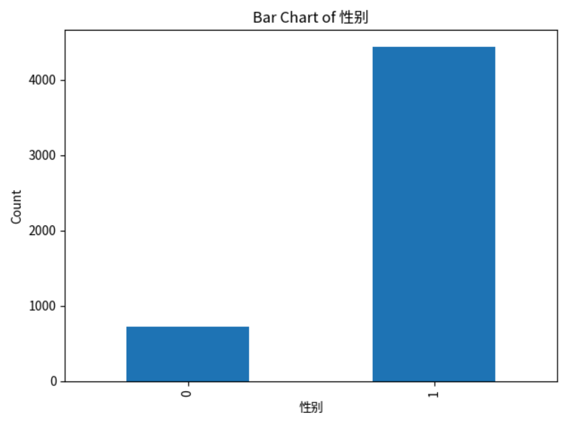




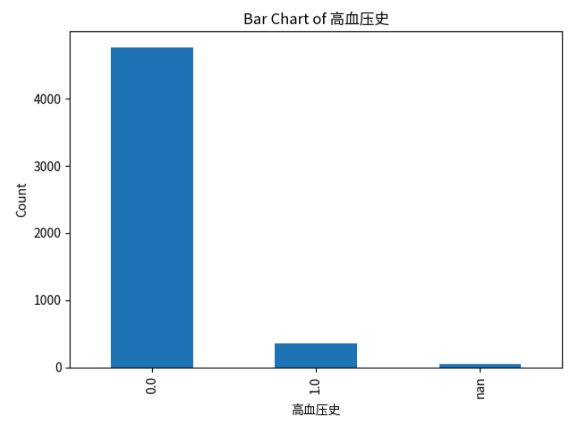




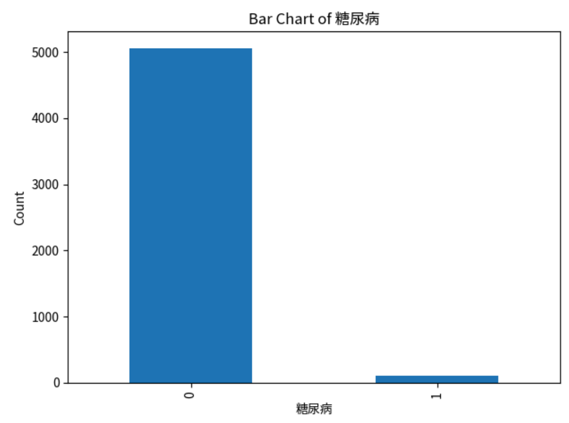




0 = female, 1 = male



0 = halthy, 1 = hypertension



0 = healthy, 1 = diabetes

5160 patients:
4851 have no mention of diabetes in any form in their 体检结论 column.
205 have low risk (糖尿病筛查风险评估轻度风险)
74 have high risk (糖尿病筛查风险评估高风险或糖尿病)
30 have diabetes('糖尿病, 糖尿病性视网膜病变Ⅰ期, or 糖尿病性视网膜病变Ⅱ期)

For the statistic I also labeled 糖尿病筛查风险评估高风险或糖尿病 patients as diabetic.

	mean	median	min	max	standard devation	Missing Values
年龄	42.450969	44.00	22.00	70.00	9.023194	0
体重指数	24.498135	24.41	15.64	38.19	3.007013	0
平均血糖	143.280000	146.16	100.08	198.00	32.837052	5151 (99.83%)
空腹血糖	89.464436	85.14	56.16	429.66	21.185708	7 (0.14%)
糖化血红蛋白A1C	6.600000	6.70	5.10	8.50	1.140175	5151 (99.83%)

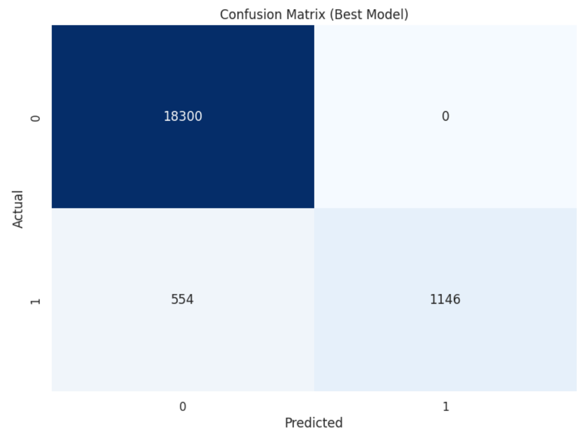
Distribution for 性别:
  Value=0: 725 patients (14.05%)
  Value=1: 4435 patients (85.95%)
  Missing values: 0
Distribution for 高血压史:
  Value=0.0: 4759 patients (92.23%)
  Value=1.0: 355 patients (6.88%)
  Value=nan: 46 patients (0.89%)
  Missing values: 45 (0.89%)
Distribution for 糖尿病:
  Value=0: 5056 patients (97.98%)
  Value=1: 104 patients (2.02%)
  Missing values: 0

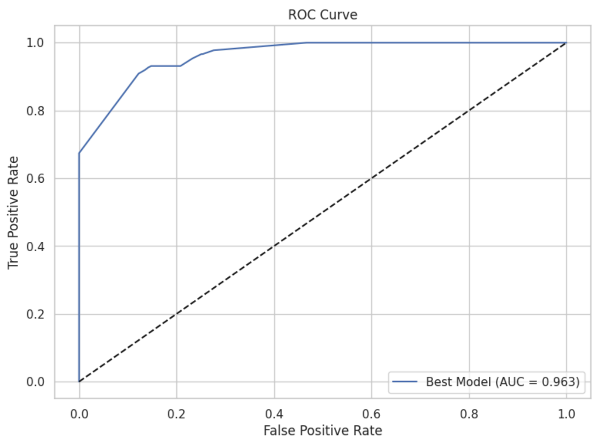
Training of XGBoost Models

Before we started to design the model we did some research on what kind of models generally perform well on medical classification problems with tabular test data like this. We concluded that gradient boosted decision trees are probably the best option. Gradient boosting is a variation of basic decision trees. The way it works is that first, an imperfect tree of a certain depth is trained. This tree missclassifies some samples, which can be quantified with a loss function. Then a second decision tree is trained that tries to predict the loss for the misclassified samples, which can then be used to correct the first tree's prediction. We do this iteratively, training more and more decision trees, each refining the loss function of the sum of the previous trees' predictions.
There are various frameworks that implement gradient boosted decision trees. We decided on XGBoost.

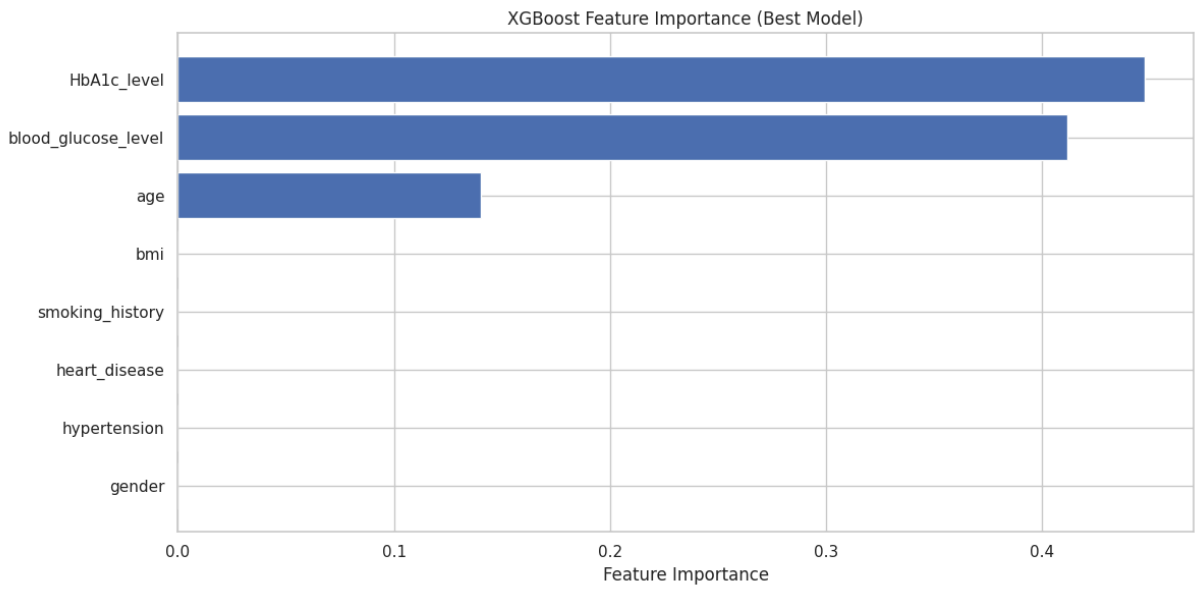
XGBoost takes in various parameters such as number of decision trees (n_estimators), the max depth of the trees (max_depth), the learning rate (learning_rate), the minimum loss reduction required to make a further partition on a leaf node of a tree (gamma), the constraint on the rows for each tree (subsample) and the constraints on the columns for each tree (colsample_bytree). Constraints on rows allows for better generalisation and constraints on columns allows for less dependence on specific features. L2 regularisation is used to prevent overfitting and the scale factor from the positive class to the negative class as mentioned earlier is also passed as an argument into the model to manage the class imbalance.
GridSearchCV was used for hyperparameter tuning, with the objective being to maximise the F1 score, which is the metric being scored for this project. The optimum hyperparameters are as follows:

```
{'colsample_bytree': 1.0, 'gamma': 0, 'learning_rate': 0.01,
'max_depth': 3, 'n_estimators': 50, 'subsample': 0.8}
```

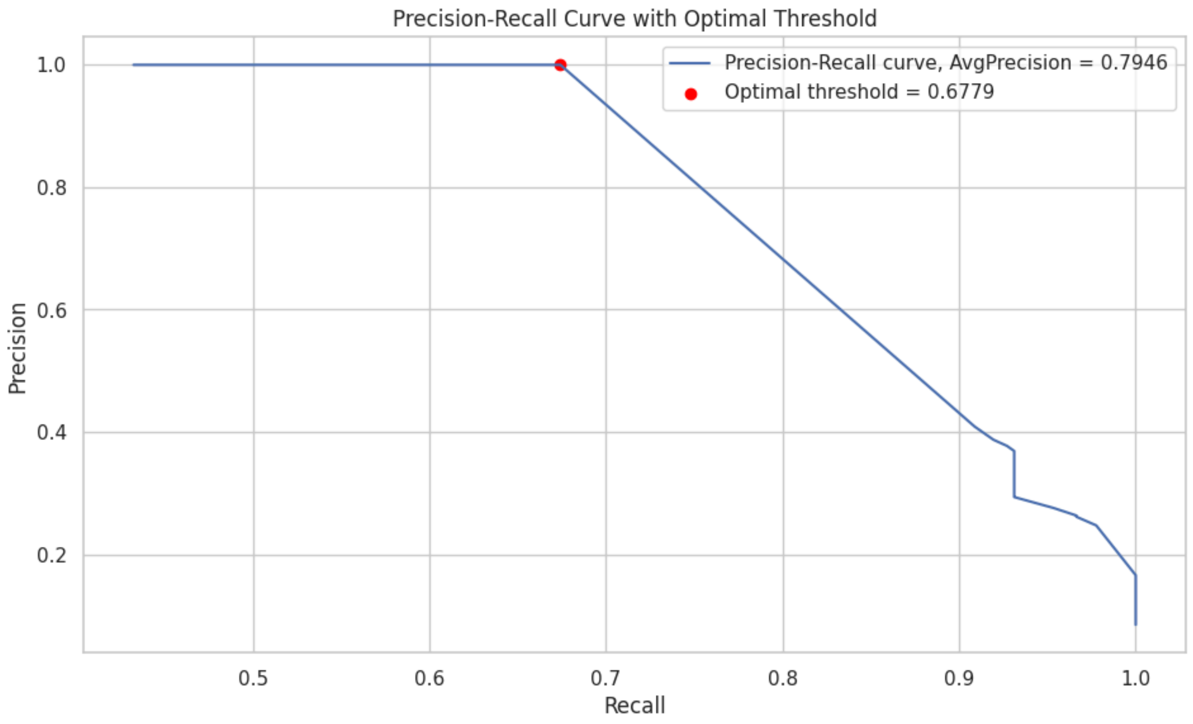




Our results show the best model after hyperparameter tuning with the default classification threshold of 0.5 to have an F1 score of 0.8008, accuracy of 0.9723 and AUC of 0.963. Its confusion matrix and ROC curve are as shown above.



Feature importance of the best model is as shown above, with only HbA1c levels, blood glucose levels and age having importance, which is rather peculiar and unexpected. It is the best model for F1 score nonetheless.



The precision-recall curve of the best model when the classification threshold is varied is as shown above. We found the optimum threshold that maximises the F1 score to be 0.6779. Model evaluation at the optimum threshold gives an F1 score of 0.8053, precision of 1.0000, recall of 0.6741 and accuracy of 0.9723.

Training NN and KNN Models

Since we were sceptical of the XGBoost models performance we also trained some other model variants, a KNN model and traditional neural network. We did hyperparameter tuning for both models using Grid Search and the Optuna library. Ultimately we arrived at these architectures/parameters:

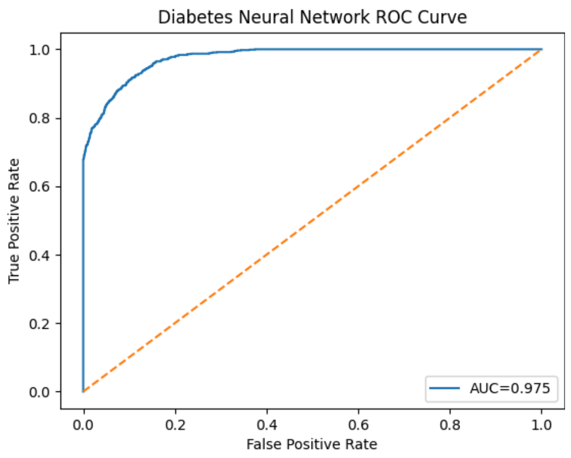
KNN: 9 nearest neighbors, uniform weights (so no weighting of neighbors based on their distance), and Manhattan distance for proximity measurement.
Traditional NN: 2 hidden layers with 16 neurons each, dropout rate of 0.3685, learning rate 0.0018, batch size 64.

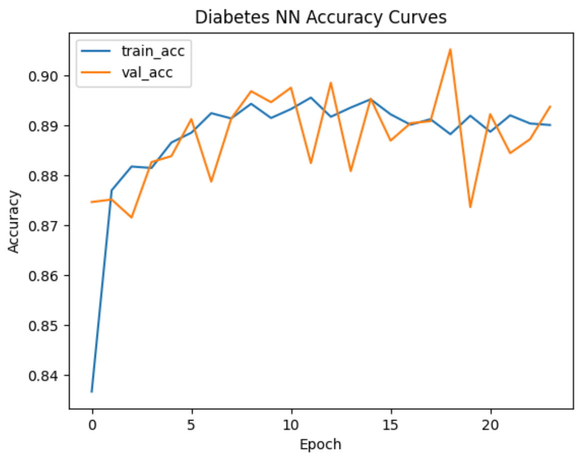
Neural Network Performance Statistics:

	Predicted Healthy	Predicted Diabetes
True Healthy	8288	862
True Diabetes	85	765

Accuracy: 0.91

	Precision	Recall	F1-Score
Healthy	0.99	0.91	0.95
Diabetes	0.47	0.90	0.62
Macro Average	0.73	0.90	0.78
Weighted Average	0.95	0.91	0.92



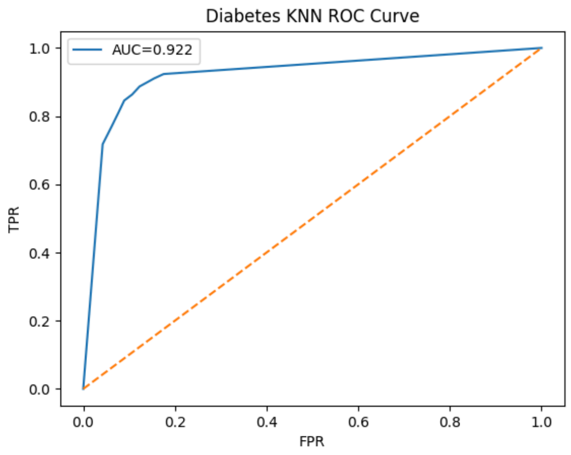


	Predicted Healthy	Predicted Diabetes
True Healthy	8173	977
True Diabetes	116	734

Accuracy: 0.89

	Precision	Recall	F1-Score

Healthy	0.99	0.89	0.94
Diabetes	0.43	0.86	0.57
Macro Average	0.71	0.88	0.76
Weighted Average	0.94	0.89	0.91



We can see that these models achieved performance that is reasonably close to that of the XGBoost Model. The class imbalance is also a problem here. During training we did increase the class weight for diabetes in the loss function proportional to the class imbalance. We can see that both models turned out to be “cautious”, in that they predicted that lots of healthy people were in fact diabetic, hence the bad precision score of all models on diabetes. In the real world this is what we would actually want, because it’s much worse not to recognize a patient who is actually diabetic than to accidentally classify a healthy person as diabetic, since you can just do further tests to confirm whether they actually do have diabetes or not.

Building an Ensemble Model

In hopes to combine the power of all three models we wanted to create an ensemble model, where the models take a vote based on their predicted probabilities. We noticed that incorporating the KNN model with the predictions of the other two was very tricky, so in the end we only combined the XGBoost and traditional neural network model. We did this by, after finalizing our models, having them generate predicted probabilities for the entire kaggle dataset, giving us two vectors of size 1x<number of all data points>. We then stacked the two vectors into a 2x<number of all data points> matrix. So each row contains the XGBoost tree’s predicted probability and the NN’s predicted probability. We then trained a single neuron perceptron to determine how the model's predictions should be combined. This gave us these statistics.

	Predicted Healthy	Predicted Diabetes
True Healthy	8181	969
True Diabetes	75	775

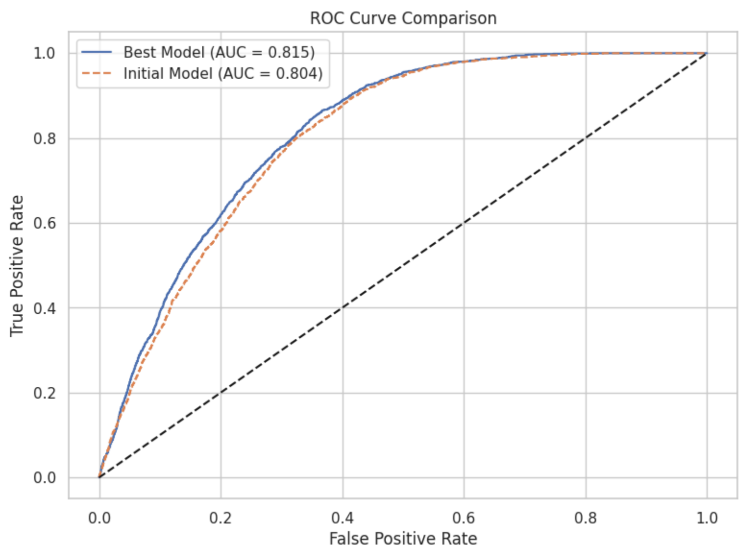
Accuracy: 0.90

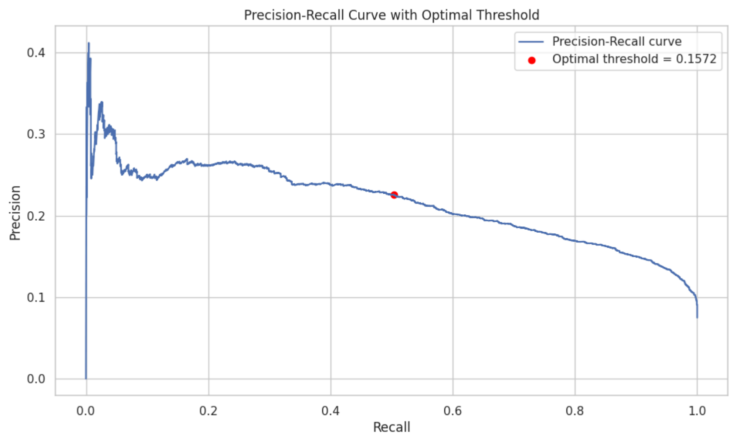
	Precision	Recall	F1-Score
Healthy	0.99	0.89	0.94
Diabetes	0.44	0.91	0.60
Macro Average	0.72	0.90	0.77
Weighted Average	0.94	0.90	0.91

ROC AUC : 0.9750

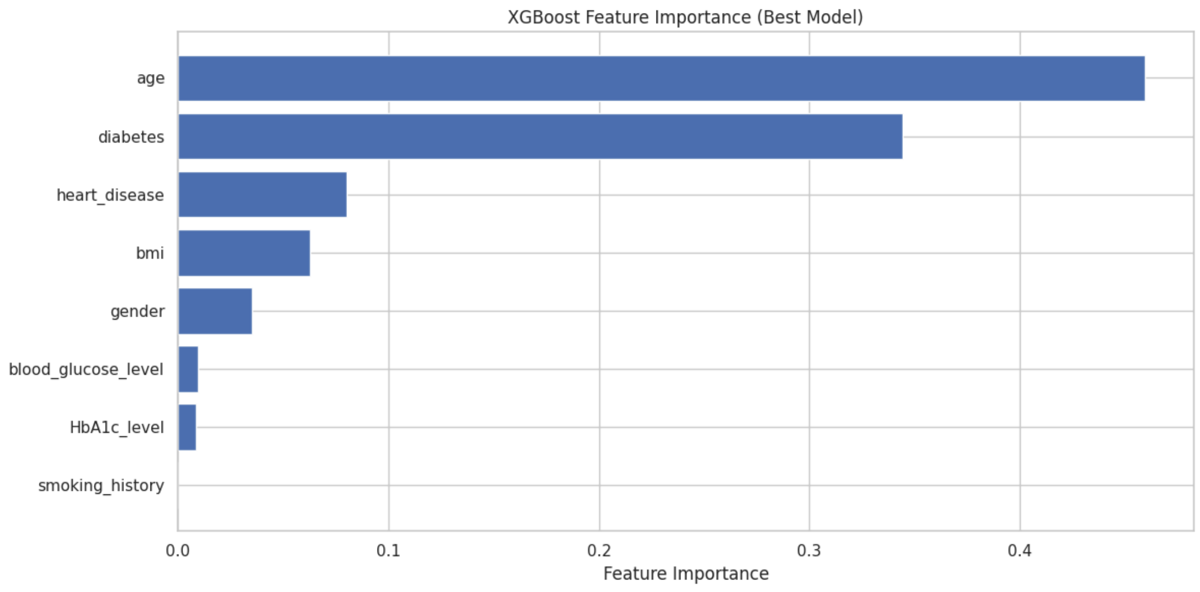
So the ensemble was actually worse than the XGBoost tree. The tree is probably just much better at recognizing hidden structure in the data and trying to mix in the NN model doesn't work well because it doesn't provide any further insights. So we ended up not using the ensemble model.

Extension to Hypertension



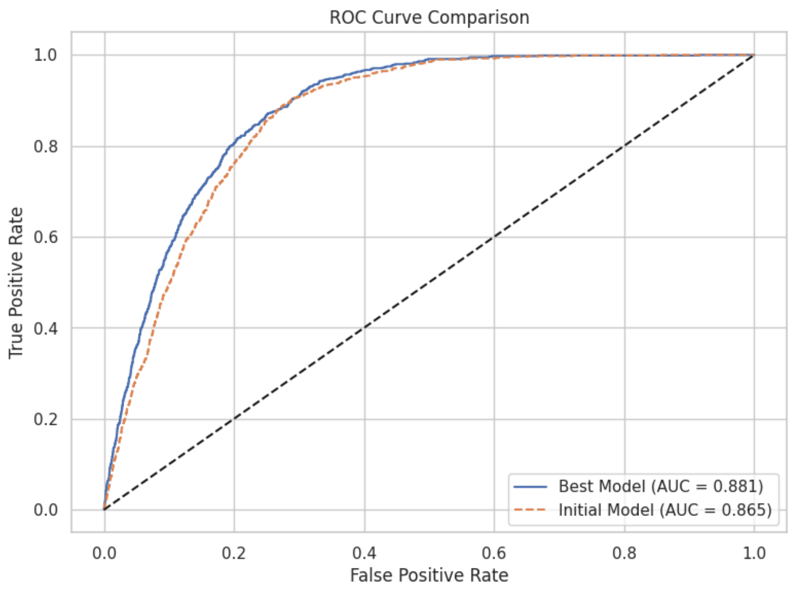


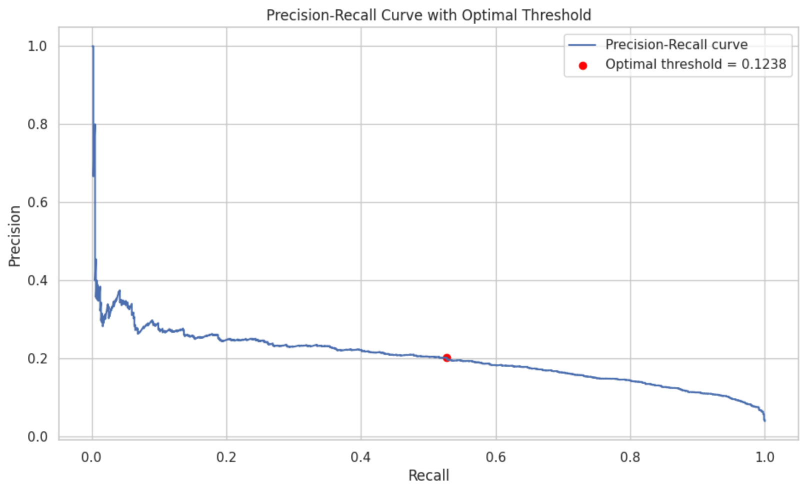
For hypertension, we used XGBoost with hyperparameter tuning, class balancing, optimisation for AUC. The above graphs are with regards to the best model. The optimal classification threshold was found to be 0.1572 and the model evaluation at the optimal threshold gives an accuracy of 0.8336.



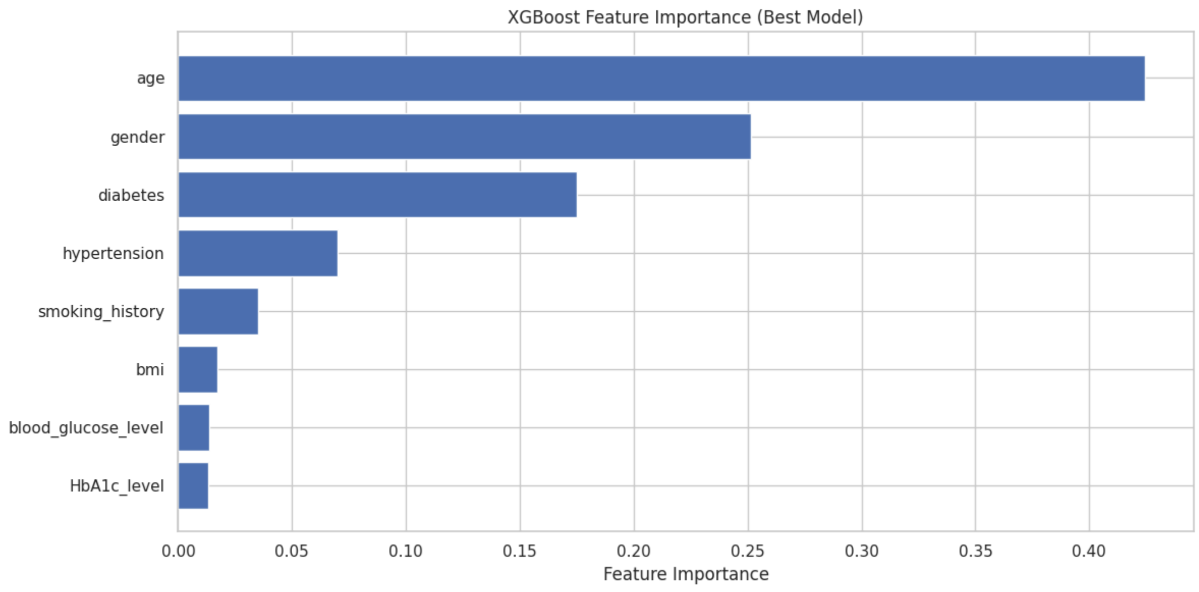
Feature importance is as shown above with age, diabetes and heart disease being the most important features for hypertension prediction.

Extension to Heart Disease





For heart disease, we also used XGBoost with hyperparameter tuning, class balancing, optimisation for AUC. The above graphs are with regards to the best model. The optimal classification threshold was found to be 0.1238 and the model evaluation at the optimal threshold gives an accuracy of 0.89985.



Feature importance is as shown above with age, gender and diabetes being the most important features for heart disease prediction.

Disease Prediction Web Application Documentation

The Disease Prediction Web Application is a full-stack solution designed to predict various health conditions including diabetes, heart disease, and hypertension based on patient data. The application leverages machine learning models to provide accurate predictions and risk assessments.

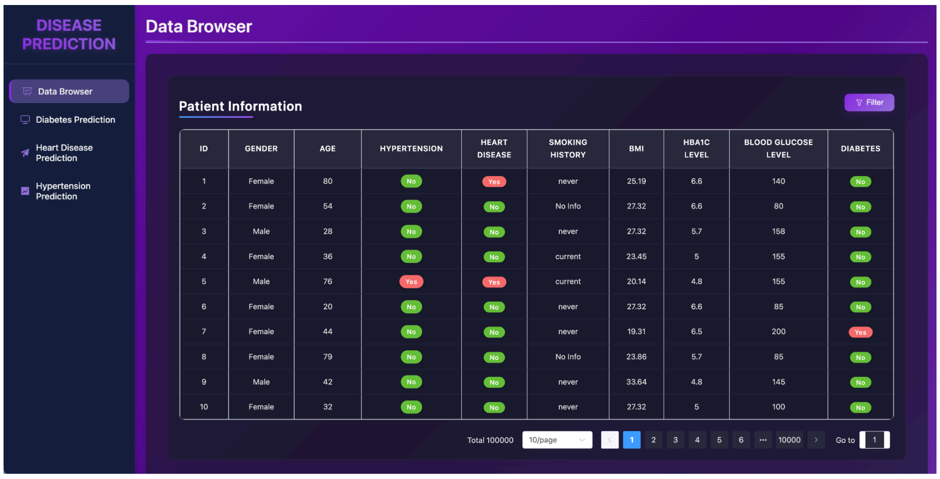
Our Web App is now available to visit on http://119.3.226.59:6061/.

[PART 1: DEVELOPMENT]

Frontend Development

The frontend of the Disease Prediction Web Application is built using Vue 3 with TypeScript, providing a modern and type-safe development experience. The application utilizes Vite as the build tool for fast development and optimized production builds. The user interface is enhanced with Element Plus, a comprehensive UI library that provides a rich set of components including buttons, forms, dialogs, and icons. The frontend architecture follows a component-based structure with the main App.vue serving as the container for all other components.

The application features four main views: a Data Browser for viewing patient records and three disease-specific prediction interfaces for Diabetes, Heart Disease, and Hypertension.

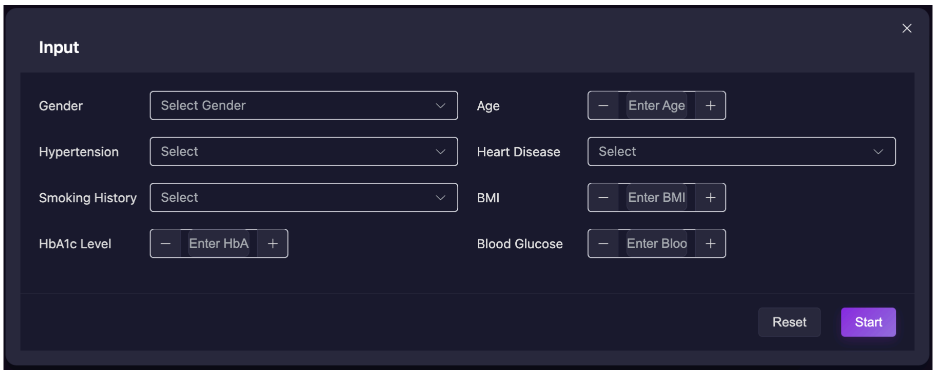


Patient Information

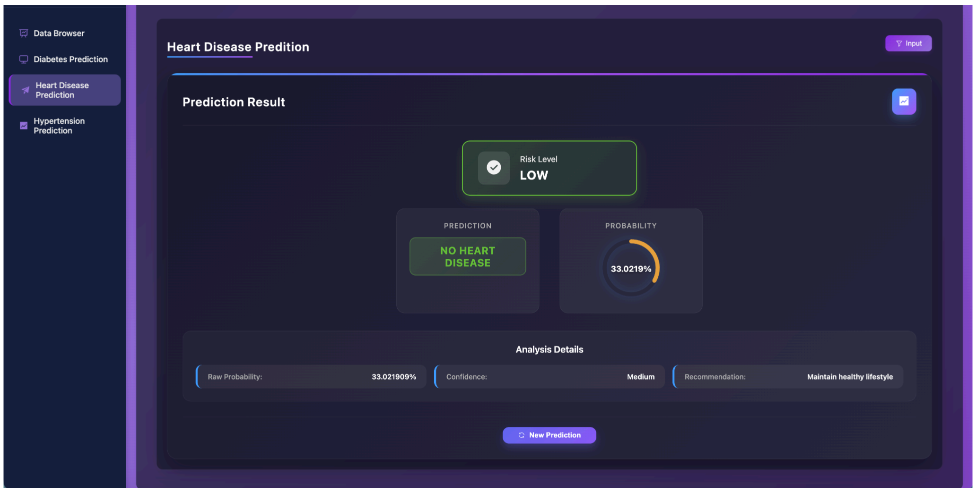
ID	GENDER	AGE	HYPERTENSION	HEART DISEASE	SMOKING HISTORY	BMI	HBA1C LEVEL	BLOOD GLUCOSE LEVEL	DIABETES
1	Female	80	No	Yes	never	25.19	6.6	140	No
2	Female	54	No	No	No Info	27.32	6.6	80	No
3	Male	28	No	No	never	27.32	5.7	158	No
4	Female	36	No	No	current	23.45	5	155	No
5	Male	76	Yes	Yes	current	20.14	4.8	155	No
6	Female	20	No	No	never	27.32	6.6	85	No
7	Female	44	No	No	never	19.31	6.5	200	Yes
8	Female	79	No	No	No Info	23.86	5.7	85	No
9	Male	42	No	No	never	33.64	4.8	145	No
10	Female	32	No	No	never	27.32	5	100	No

Total 100000 10/page 1 2 3 4 5 6 ... 10000 Go to 1

Each disease prediction component includes an intuitive form interface for inputting patient data, with validation and appropriate input controls for different data types.



The UI design implements a modern aesthetic with gradient backgrounds, card-based layouts, and smooth animations to enhance user experience. For data visualization, the application includes custom-designed circular progress indicators to represent prediction probabilities, color-coded risk levels (high/low), and detailed analysis sections.



Heart Disease Predition

Prediction Result

Risk Level
LOW

PREDICTION
NO HEART DISEASE

PROBABILITY
33.0219%

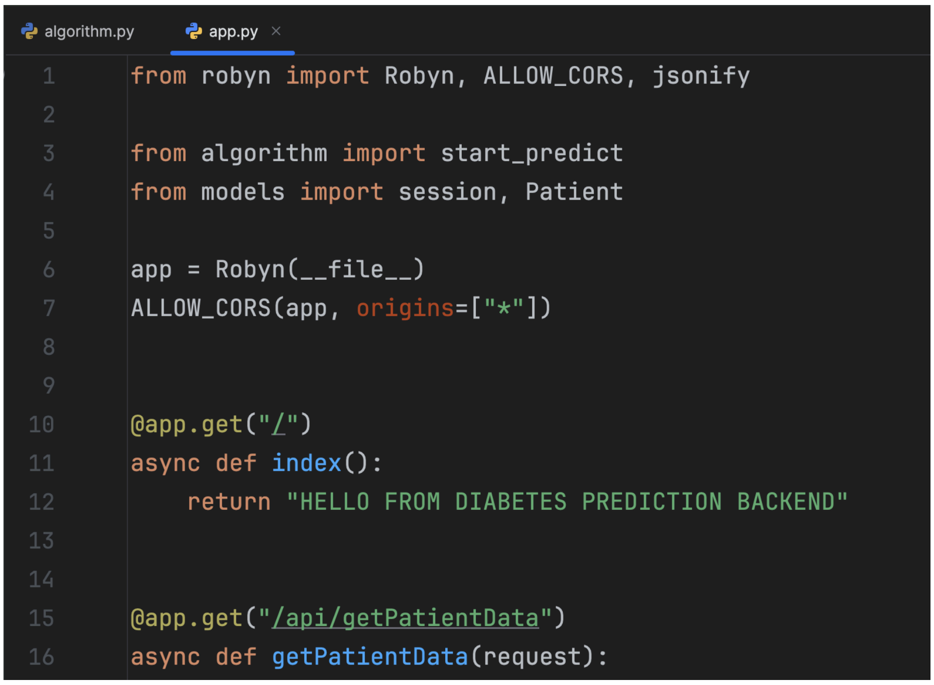
Analysis Details

Raw Probability: 33.021909%
Confidence: Medium
Recommendation: Maintain healthy lifestyle

The frontend communicates with the backend through a dedicated request utility built on top of “ofetch”, which handles HTTP requests with proper error handling. The application is fully responsive, adapting to different screen sizes through CSS media queries and flexible layouts.

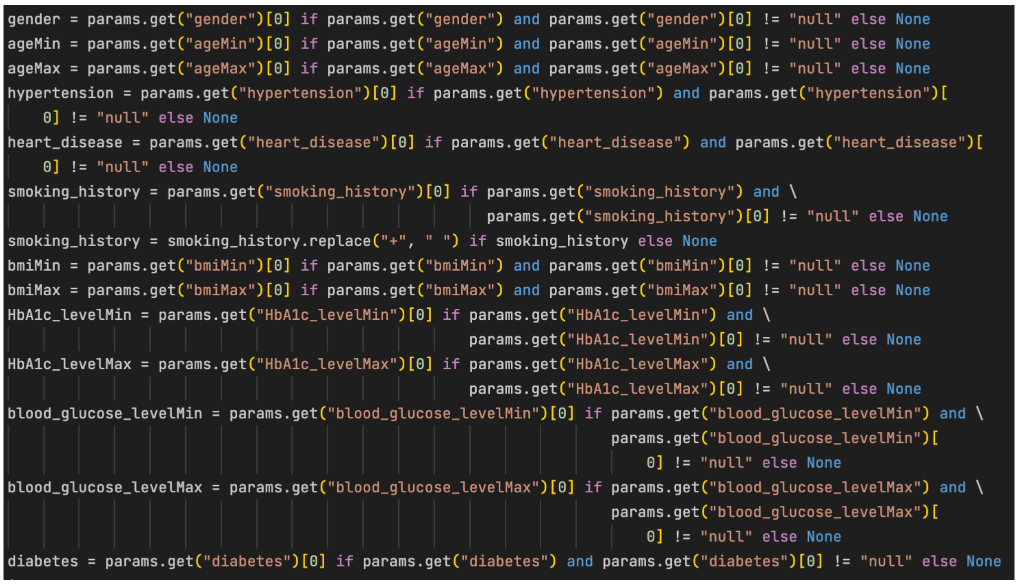
Backend Development

The backend of the Disease Prediction Web Application is developed using Robyn, a Python-based web framework that provides high-performance request handling. The server-side architecture is organized around several key components that work together to process requests, interact with the database, and generate predictions. The main application file, app.py, defines the API endpoints that the frontend can access, including routes for retrieving patient data and generating disease predictions.



```python
from robyn import Robyn, ALLOW_CORS, jsonify

from algorithm import start_predict
from models import session, Patient

app = Robyn(__file__)
ALLOW_CORS(app, origins=["*"])


@app.get("/")
async def index():
    return "HELLO FROM DIABETES PREDICTION BACKEND"


@app.get("/api/getPatientData")
async def getPatientData(request):
```

The backend implements comprehensive data filtering capabilities, allowing the frontend to request specific subsets of patient data based on various criteria such as age, gender, BMI, and other health metrics.



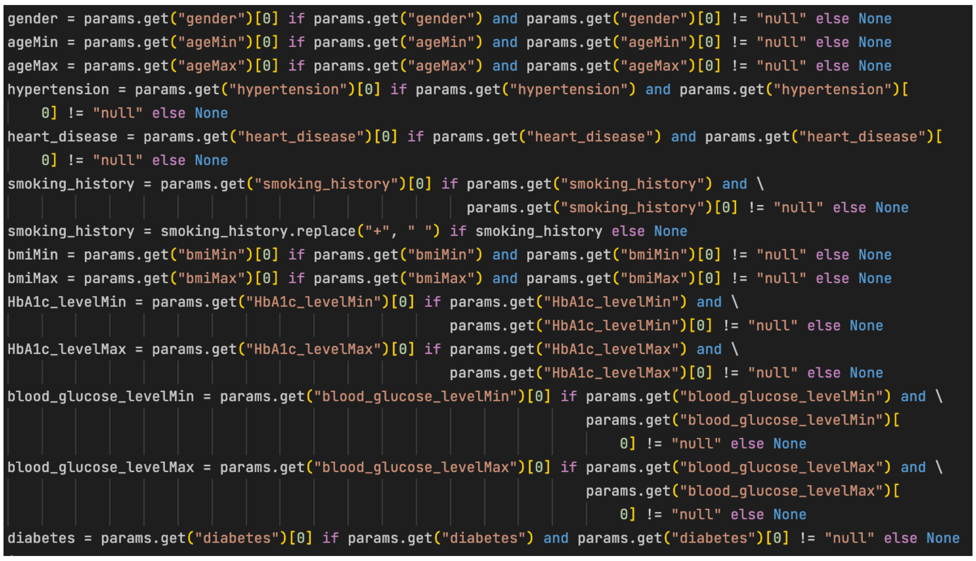
```python
gender = params.get("gender")[0] if params.get("gender") and params.get("gender")[0] != "null" else None
ageMin = params.get("ageMin")[0] if params.get("ageMin") and params.get("ageMin")[0] != "null" else None
ageMax = params.get("ageMax")[0] if params.get("ageMax") and params.get("ageMax")[0] != "null" else None
hypertension = params.get("hypertension")[0] if params.get("hypertension") and params.get("hypertension")[
    0] != "null" else None
heart_disease = params.get("heart_disease")[0] if params.get("heart_disease") and params.get("heart_disease")[
    0] != "null" else None
smoking_history = params.get("smoking_history")[0] if params.get("smoking_history") and \
                                                     params.get("smoking_history")[0] != "null" else None
smoking_history = smoking_history.replace("+", " ") if smoking_history else None
bmiMin = params.get("bmiMin")[0] if params.get("bmiMin") and params.get("bmiMin")[0] != "null" else None
bmiMax = params.get("bmiMax")[0] if params.get("bmiMax") and params.get("bmiMax")[0] != "null" else None
HbA1c_levelMin = params.get("HbA1c_levelMin")[0] if params.get("HbA1c_levelMin") and \
                                                   params.get("HbA1c_levelMin")[0] != "null" else None
HbA1c_levelMax = params.get("HbA1c_levelMax")[0] if params.get("HbA1c_levelMax") and \
                                                   params.get("HbA1c_levelMax")[0] != "null" else None
blood_glucose_levelMin = params.get("blood_glucose_levelMin")[0] if params.get("blood_glucose_levelMin") and \
                                                                   params.get("blood_glucose_levelMin")[
                                                                       0] != "null" else None
blood_glucose_levelMax = params.get("blood_glucose_levelMax")[0] if params.get("blood_glucose_levelMax") and \
                                                                   params.get("blood_glucose_levelMax")[
                                                                       0] != "null" else None
diabetes = params.get("diabetes")[0] if params.get("diabetes") and params.get("diabetes")[0] != "null" else None
```

The core prediction functionality is encapsulated in the algorithm.py module, which handles loading the pre-trained machine learning models, preprocessing input data, and generating prediction results. The application uses XGBoost models that have been trained on health datasets and serialized using pickle for efficient storage and loading. The prediction process includes data normalization, feature transformation, and probability calibration to ensure accurate results. The backend implements a consistent response format using JSON, with standardized status codes and message structures to facilitate error handling on the frontend. For database interactions, the application uses SQLAlchemy as an Object-Relational Mapping (ORM) tool, providing an abstraction layer over the underlying

SQLite database. The backend also includes configuration management through a dedicated config.py file, allowing for easy environment-specific settings.

Database Design

The database design for the Disease Prediction Web Application utilizes SQLite as the database engine, chosen for its simplicity, reliability, and zero-configuration nature. The database schema is defined using SQLAlchemy's declarative base approach, which allows for object-oriented representation of database tables. The primary data model is the Patient class, which maps to the 'patient' table in the database. This table stores all relevant health metrics needed for disease prediction, including gender, age, hypertension status, heart disease status, smoking history, BMI, HbA1c level, blood glucose level, and diabetes status. Each field is carefully typed to ensure data integrity, with appropriate use of Integer, Boolean, and Float column types.

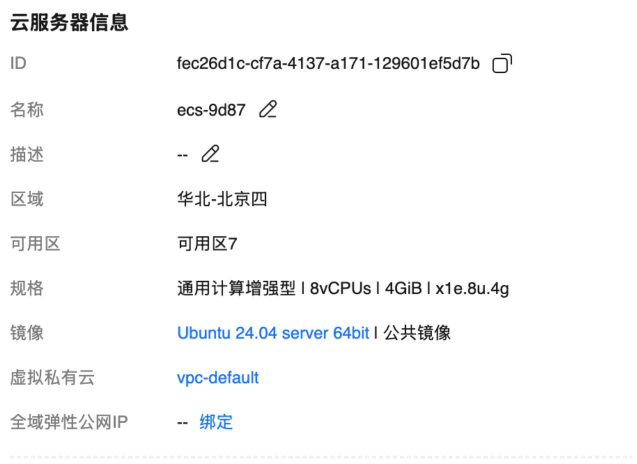


```
gender = params.get("gender")[0] if params.get("gender") and params.get("gender")[0] != "null" else None
ageMin = params.get("ageMin")[0] if params.get("ageMin") and params.get("ageMin")[0] != "null" else None
ageMax = params.get("ageMax")[0] if params.get("ageMax") and params.get("ageMax")[0] != "null" else None
hypertension = params.get("hypertension")[0] if params.get("hypertension") and params.get("hypertension")[
    0] != "null" else None
heart_disease = params.get("heart_disease")[0] if params.get("heart_disease") and params.get("heart_disease")[
    0] != "null" else None
smoking_history = params.get("smoking_history")[0] if params.get("smoking_history") and \
                                                      params.get("smoking_history")[0] != "null" else None
smoking_history = smoking_history.replace("+", " ") if smoking_history else None
bmiMin = params.get("bmiMin")[0] if params.get("bmiMin") and params.get("bmiMin")[0] != "null" else None
bmiMax = params.get("bmiMax")[0] if params.get("bmiMax") and params.get("bmiMax")[0] != "null" else None
HbA1c_levelMin = params.get("HbA1c_levelMin")[0] if params.get("HbA1c_levelMin") and \
                                                    params.get("HbA1c_levelMin")[0] != "null" else None
HbA1c_levelMax = params.get("HbA1c_levelMax")[0] if params.get("HbA1c_levelMax") and \
                                                    params.get("HbA1c_levelMax")[0] != "null" else None
blood_glucose_levelMin = params.get("blood_glucose_levelMin")[0] if params.get("blood_glucose_levelMin") and \
                                                                    params.get("blood_glucose_levelMin")[
                                                                        0] != "null" else None
blood_glucose_levelMax = params.get("blood_glucose_levelMax")[0] if params.get("blood_glucose_levelMax") and \
                                                                    params.get("blood_glucose_levelMax")[
                                                                        0] != "null" else None
diabetes = params.get("diabetes")[0] if params.get("diabetes") and params.get("diabetes")[0] != "null" else None
```

The database implementation includes a to_json method in the Patient model to facilitate serialization of patient records for API responses. For database migrations and schema evolution, the application uses Alembic, which provides a version-controlled approach to database changes. The migration configuration is stored in the alembic directory, with the env.py file setting up the migration environment and connecting to the SQLAlchemy models. The database connection is managed through a session factory pattern, with a global session object available for database operations throughout the application. The naming convention for database constraints is explicitly defined to ensure consistency across different database environments. The database is initialized with the Base.metadata.create_all method, which creates all defined tables if they don't already exist, though this functionality is primarily handled by Alembic migrations in production environments.

[PART 2: DEPLOYMENT]

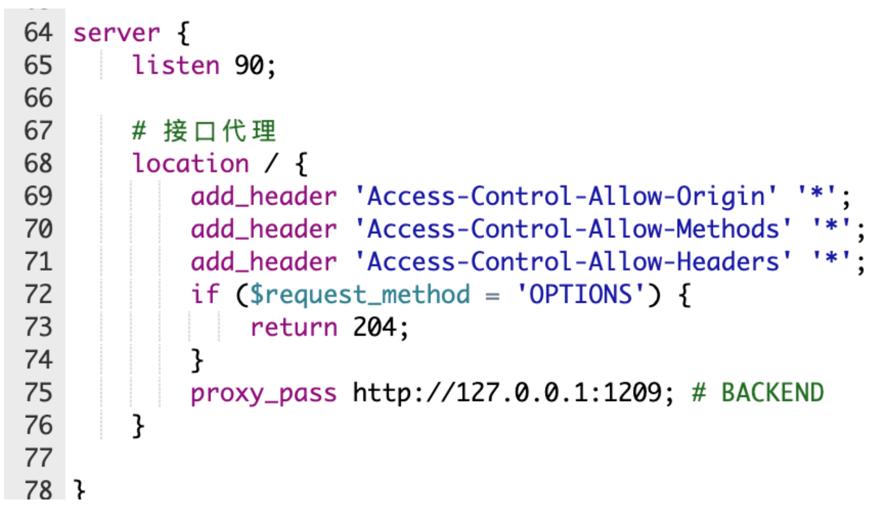
The Disease Prediction Web Application is deployed on Huawei Cloud's Elastic Cloud Server (ECS) infrastructure. We selected a General Computing Enhanced instance with 8 virtual CPUs and 4 GiB of memory, providing a balanced combination of computing power and memory resources suitable for our application's requirements. The deployment process is facilitated by BT Panel (also known as aaPanel), a popular web hosting control panel that simplifies server management and application deployment.



云服务器信息

ID	fec26d1c-cf7a-4137-a171-129601ef5d7b
名称	ecs-9d87
描述	--
区域	华北-北京四
可用区	可用区7
规格	通用计算增强型 I 8vCPUs I 4GiB I x1e.8u.4g
镜像	Ubuntu 24.04 server 64bit I 公共镜像
虚拟私有云	vpc-default
全域弹性公网IP	-- 绑定

For the frontend deployment, we utilized Nginx as a web server and reverse proxy. The Vue.js+Vite application is built into static files using the production build process, which optimizes the JavaScript, CSS, and HTML for performance. These static files are then served by Nginx from a designated directory on the server. The Nginx configuration includes specific settings for handling Single Page Application (SPA) routing, ensuring that all routes are properly directed to the index.html file. Additionally, we configured Nginx to act as a reverse proxy for API requests, forwarding them to the backend service. This reverse proxy setup is crucial for resolving Cross-Origin Resource Sharing (CORS) issues that would otherwise occur when the frontend attempts to communicate with the backend API. By routing API requests through Nginx to the backend service, we eliminate the need for complex CORS configurations in the backend code.



```
server {
    listen 90;

    # 接口代理
    location / {
        add_header 'Access-Control-Allow-Origin' '*';
        add_header 'Access-Control-Allow-Methods' '*';
        add_header 'Access-Control-Allow-Headers' '*';
        if ($request_method = 'OPTIONS') {
            return 204;
        }
        proxy_pass http://127.0.0.1:1209; # BACKEND
    }

}
```

The backend deployment involves setting up a dedicated Python virtual environment to ensure dependency isolation and consistent execution. We created a virtual environment using Python's built-in venv module, which provides a clean, isolated Python installation for our application. Within this virtual environment, we installed all the required dependencies specified in the requirements.txt file, including Robyn for the web framework, SQLAlchemy for database operations, and the necessary machine learning libraries for prediction functionality. The backend service is configured to run as a persistent process using a process manager, which ensures that the application remains running even after server restarts or potential crashes. The process manager also handles logging, monitoring, and automatic restarts if needed. The backend service listens on a specific port that is only accessible locally, with external access managed through the Nginx reverse proxy for enhanced security.

The entire deployment architecture is designed with security and performance in mind. All communication between the client and server is encrypted using HTTPS, with SSL certificates managed through the BT Panel interface. Regular backups of the database and application files are scheduled to prevent data loss. The deployment process follows a systematic approach with version control integration, allowing for smooth updates and rollbacks when necessary. This cloud-based deployment provides scalability options, allowing us to adjust resources based on usage patterns and growth requirements.