AI DRIVEN SYMPTOM-BASED DISEASE CLASSIFICATION

Submitted by

21PWELE5844 ASHAN AHMAD

21PWELE5863 RIJJA NOOR

Supervisor

DR. GUL MUHAMMAD

DEPARTMENT OF ELECTRICAL ENGINEERING

UNIVERSITY OF ENGINEERING AND TECHNOLOGY

PESHAWAR, PAKISTAN

June 2025

AI DRIVEN SYMPTOM-BASED DISEASE CLASSIFICATION

by

21PWELE5844 Ashan Ahmad

21PWELE5863 Rijja Noor

FYP report

submitted to the University of Engineering and Technology, Peshawar

as a part of the requirement for completing the degree of

Bachelor of Science

in

Electrical Communication Engineering

2025

AUTHOR'S DECLARATION

We want to clearly state that this thesis was written entirely by us, and everything included in it is based on our own work, unless we've specifically said otherwise. We've also made any final changes that were suggested by our examiners during the review process.

Throughout this project, we have tried our best to follow the ethical and academic rules given by the Higher Education Commission, especially the ones related to research honesty and quality standards.

We're also aware that this thesis might be shared online for public use. It is an assumption that we take responsibility and is very confident that the information and research work being put forth here is pure and true. Any inquiry or further clarification is open here for giving details.

Finally, we think research should be available openly, so we are happy for anybody in the academic world to use our work further, with hopes that it will serve future students or researchers pursuing similar topics.

Abstract

This project analyzes some of today's current issues faced by healthcare delivery systems-the dramatically rising rates of non-communicable diseases (NCDs) and the wide disparity in accessing medical care most glaringly seen in the rural and underdeveloped areas. One plausible means to reduce some of these challenges is through Artificial Intelligence (AI). AI would help people benefit from medical aid by accelerating early diagnosis and making it more accessible to those who needed it. AI-based diagnostic tools have shown they can help improve the accuracy of diagnoses, reduce the pressure on doctors, and give faster early-stage assessments. One area where this kind of technology fits well is in identifying diseases based on symptoms — something doctors have always done manually. But because many illnesses share similar symptoms, and because mistakes can happen or medical help isn't always nearby, getting a correct diagnosis isn't always easy. These issues point to the need for a smart system that can take in symptoms and suggest possible illnesses reliably. That's what this project focuses on: figuring out how to build a deep learning model using an Artificial Neural Network (ANN) that can correctly predict diseases from symptoms reported by patients, with strong accuracy and dependability.

This study was primarily conducted to develop an AI-based disease classification system to assist an individual to know what diseases he or she could possibly suffer from, based solely on symptoms, before ever seeing a doctor. For this purpose, a dataset containing 53 symptoms corresponding to 10 diseases was collected. The collected data was cleaned and sorted appropriately so that it could be used. Afterward, some random noise was introduced to the data to make it a little more diverse for training the model even better. Artificial Neural Network (ANN) was designed based on the above data to analyze enter symptoms and suggest a probable disease for the person. For testing how good the model really is, we used information and cases taken from trusted websites like WHO, CDC, Mayo Clinic, and some others like MedlinePlus and Cleveland Clinic. In the end, we took the trained model and built it into a mobile app. The app was made using React Native for the user side, and Flask was used to handle things on the backend.

The scope of this project is focused on the classification of 10 highly prevalent diseases in Pakistan. The developed model provides a preliminary assessment based on symptom patterns and is designed to function purely as a decision-support tool to inform and assist, rather than replace, professional medical consultation or diagnosis. The model's capabilities are limited; it does not diagnose disease severity, suggest specific treatments, or handle real-time physiological data. The scope explicitly includes the development of a mobile application interface to ensure the accessibility and usability of the trained model for end-users.

This study contributes to the field of intelligent healthcare by demonstrating the effective application of deep learning for disease diagnosis based solely on symptoms. The approach offers potential benefits for individuals in areas with limited medical infrastructure and supports telehealth systems, by providing a scalable model.

Acknowledgments

First, we want to extend our gratitude to Allah (SWT) for helping us reach here. Honestly, there were points when we were stuck, so frustrated, and thinking if we were ever going to make it through, but somehow every knot got untied just because of His help.

Dr. Gul Muhammad, our supervisor, deserves a very warm thank you from us. Not only has he guided us through research stuff, but at times of need, he has motivated and encouraged us when we felt that we were stuck. Sometimes he wouldn't just enlighten us with technical advice-he gave us that much-needed, well suitable push towards the focused direction. We honestly learned a lot from him, more than just how to write a thesis.

Big thanks as well to UET Peshawar. Having a proper lab, quiet spots to work in, and just the whole environment helped us stay productive most of the time. It might not seem like a big deal, but it made things easier.

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AI	Artificial Intelligence
NCDs	Non-Communicable Diseases
ANN	Artificial Neural Network
WHO	World Health Organization
CDC	Centers for Disease Control and Prevention
DL	Deep Learning
ML	Machine Learning
FNN	Feedforward Neural Network
SVM	Support Vector Machine (Classifier)
KNN	K-Nearest Neighbors (Classifier)
GNNs	Graph Neural Networks
AISympCheck	AI-powered Symptom Checker Apps
XAI	Explainable AI
UX	User Experience
CTA	Call to Action
API	Application Programming Interface
REST API	Representational State Transfer Application Programming Interface
JSON	JavaScript Object Notation
UI	User Interface84
LLMs	Large Language Models
HTTP	Hypertext Transfer Protocol
PNG	Portable Network Graphics (Image format)
CSV	Comma-Separated Values (File format)

CHAPTER 1 INTRODUCTION

1.1 Background

Healthcare these days is facing all kinds of problems, and one big issue is how diseases that aren't contagious, like diabetes or heart conditions (NCDs) [3], are becoming more and more common. On top of that, lots of people, especially in places far from cities or clinics, still don't have easy ways to get checked early when something's wrong. The way most health systems work now, where doctors need to be there in person and everything is done step by step, just doesn't seem to be enough anymore. There aren't enough medical staff, and a lot of people have to travel really far just to see someone. That's why we see so many people crowding into emergency rooms, even when their problem could've probably been handled earlier if there was a simple check-up or first-level diagnosis option.

Because of all the pressure on healthcare systems now, and also because tech has improved really fast lately, people have started looking at Artificial Intelligence (AI) as a new way to solve some of these problems. AI has been doing pretty well in medical work, especially when it comes to getting things done faster and more accurately. It's not just that AI replaces heavy physical work; now it can tackle tasks reserved for trained professionals. It is faster, oftentimes more consistent, and at times cheaper than standard practices. One instance where AI can stick to a given task better than a human is when technically it has to pay equal attention to the same thing over and over.

One way AI is being used in healthcare that really stands out is for figuring out what illness a person might have based on the symptoms they tell. Usually, this kind of thing is done by doctors, but now with machine learning, it can actually be handled by computers in a faster and more repeatable way. These AI systems are getting more common, especially when it comes to catching things early or helping figure out what could be wrong. They don't replace a full diagnosis but can give a quick idea based on data, and that also takes some pressure off doctors. [3]

This project actually aims to develop an AI system that operates solely by understanding the symptoms in order to predict the disease. We opted for Deep Learning, and more specifically, something referred to as Artificial Neural Network (ANN). The idea here is that we would train it on a dataset where the symptoms are listed in yes/no values, and each combination of symptoms corresponds to a particular disease. Hopefully, this will help the model learn how symptoms are connected to different illnesses. The final version of this will be made into a mobile app so it's easy to use, and the idea is that it can give people a rough idea of what might be going on before they get to a hospital or clinic.

1.2 Problem Statement

Even though AI sounds like a great solution, figuring out what disease a person has just from symptoms is still really tough. There are a few reasons for that. For starters, many diseases have the same signs, which makes it confusing. Like, one symptom can show up in a bunch of totally different illnesses, so it's not easy to know exactly what's wrong just by looking at that. [7] You need to see patterns across multiple symptoms to make a good guess, and that can get complicated. Also, when diagnosis is done by people the usual way, mistakes can happen. Doctors may be weary, make unintentional assumptions, or simply lack the necessary information. That is also part of the problem. Finally, there is the issue of limited access to healthcare. People in many remote areas do not have access to doctors on a timely basis, if at all. This only exacerbates the difficulty of receiving a good diagnosis on time.

All these problems just show that there's a real need for something smart that can take in someone's symptoms and give back a pretty accurate idea of what illness it might be. If there was a system that could do that reliably, it would really help — not just doctors doing the first checks in hospitals or clinics, but also regular people trying to figure out if what they're feeling might be serious or not.

What this project is mainly trying to solve is: how can we use Artificial Intelligence — especially an Artificial Neural Network (ANN) — to build something that can guess the disease just from the symptoms a person says they have, and do it in a way that's actually dependable? That means figuring out how symptoms link together, using machine learning in the right way, and making sure the results are good enough to be helpful without replacing real medical advice.

1.3 Objectives

Basically, the primary purpose of this project was to make an AI model that would tell some disease based on the symptoms that someone would tell. Rather, this would be like an initial check-up so that people could know before visiting any doctor, especially when needed in most places.

Initially, we used a dataset of 53 symptoms with 10 diseases; the data was very much dirty initially, so we had to clean many things such as missing values and weird formats. After that, some random changes were introduced in the combinations of symptoms to help the model learn better and not be stuck on repeating patterns.

After this, we built a neural network, a type of AI that tries to learn the correlation of symptoms with diseases. The inputs were in yes/no style (1 or 0) and tested the various versions of the model. Finally, the best version was selected. After training, we didn't just rely on the training data. We also tested it with examples from places like WHO, CDC, Mayo

Clinic, and a few others to see how it holds up in different cases.

Finally, once the model seemed solid enough, we added it into a mobile app. It's made using React Native on the user side and Flask on the server side, and it was made in a way that people can open it on their phones and check symptoms without needing to know how the tech works.

1.4 Scope of the Study

This project mainly looks at classifying 10 diseases that are pretty common in Pakistan. The model we made is supposed to give an early idea based on what symptoms someone has — like a first check, not a full diagnosis or anything like that.

Also, it's important to be clear about what we didn't cover in this study. The model does not diagnose disease severity. It provides a probability or likelihood score for each of the 10 diseases based on reported symptoms, but it does not quantify how severe the potential condition might be. Similarly, the system does not suggest specific treatments or medical interventions. Its function is limited to aiding in the identification of a potential condition, leaving treatment decisions to qualified healthcare professionals. Furthermore, the model does not handle real-time physiological data (e.g., data from wearables); its input is confined to symptom descriptions provided by the user.

The system is intended to function purely as a decision-support tool, designed to inform and assist, not replace, professional medical consultation or diagnosis. The scope also explicitly includes the development of a mobile application interface, ensuring the accessibility and usability of the trained model for end-users. The project's evaluation includes focusing on prevalent diseases, particularly in the context of Pakistan, to provide a localized perspective.

1.5 Significance of the Study

This study holds significant value and contributes to the burgeoning field of intelligent healthcare. By demonstrating the effective application of deep learning for disease diagnosis based solely on symptoms, the project provides a tangible example of how advanced AI techniques can address real-world healthcare challenges.

The approach developed offers several key benefits. Firstly, it can directly benefit individuals in areas lacking adequate medical infrastructure or access to professional healthcare. By providing a reliable preliminary assessment tool via a mobile application, it bridges geographical gaps and empowers individuals to seek appropriate medical attention sooner. This aligns with the broader goal of increasing access to health information and care, potentially impacting population health outcomes.

Secondly, the system developed supports telehealth initiatives. As demand for remote diagnosis increases, especially in contexts like the COVID-19 pandemic, tools that can accurately classify potential conditions based on user-reported symptoms are invaluable for virtual consultations and remote patient monitoring. [3] Implementing AI into actual medical practices can go a long way in dealing with the gigantic amount of medical data as well as helping to reduce those activities that put a strain on doctors' time. Thus, this would make a life easier for doctors so that they may actually spend time caring for their patients.

A good thing about this project is that our model can evolve in the future. More diseases could be added, or more detail could be applied to symptoms. What we are doing here is almost a starting point for building better AI-based tools in healthcare that will hopefully give rise to more detailed and advanced models later on.

This whole idea is really about using data and AI to make health stuff more accessible, especially online. It matches with how the world is moving toward digital health, where people can get quick help or insights without going to a hospital right away. Since AI can lower costs and help systems run better, this project shows one way it could actually be used. And because we put it into a phone app, it can reach more people who want a quick way to check symptoms on their own.

CHAPTER 2 LITERATURE REVIEW

2.1 Introduction to Symptom-Based Disease Classification

So, for a long time, the usual way of figuring out what sickness someone might have just come down to the doctor's own experience and how they kind of connect the dots with what the patient says. It works, yeah, but it can be slow, and not every doctor might think the same thing depending on the situation or how busy they are or stuff like that. But now, what with all this new technology, mac and ai-machine-learning-things, things are really starting to twist. Quite like, you don't have to rely only on the doctor's brain. AI can help make sense of things quickly, especially when getting to a doctor becomes difficult, such as in distant or rural parts of the world. [6] And AI can also help reduce mistakes sometimes and just take care of the early steps, so doctors don't have to start from scratch every time. One example is where it looks at symptoms and tries to figure out what the illness could be — something doctors do, but now machines can kind of do too, once they're trained on that kind of data.

These days, lots of people look up health info online and try to figure out what's wrong with them before they even see a doctor. Because of this, tools for self-diagnosis have become way more important, especially the ones that use algorithms powered by AI. Basically, an algorithm-based symptom checker is a service where you enter your symptoms, and it tries to guess what disease you might have and what you should do next. Some popular examples are the Mayo Clinic symptom checker, Babylon Health, the Ada app, and K Health [1]. While these apps can help more people get easy access to medical info, they also bring up tricky questions about how reliable they are and whether people can really trust them.

2.2 Artificial Intelligence and Machine Learning in Healthcare

Machines that are taught to do things that people ordinarily do are called "Artificial Intelligence." No longer is this limited to merely the application of physical acts; now other classes of work, considered so far as "cognitive" functions, are being performed by machines as well: for example, pattern recognition or decision-making. In healthcare, this shift is showing up more and more. From the boring admin stuff to real medical jobs, AI's being used in ways that weren't really possible before. One major thing it's helping with is spotting diseases early and improving how accurate diagnoses are. Some studies even say that using AI in healthcare could make things run smoother, safer, and honestly just make life a bit easier for both doctors and patients.

Artificial intelligence has brought tremendous advantages to the healthcare sector, especially in terms of time and cost-saving from treatment and resource use. Fastening the tours of diagnosis and decision-making is crucial for a decisive influence — sometimes saving the lives of patients. AI is being used for all sorts of things like diagnosing illnesses, helping with

treatment plans, sharing information, monitoring patients, collecting data, offering advice, and even doing surgeries from a distance. It's also proving super useful when it comes to analyzing huge amounts of medical data and pulling out insights that can help researchers move forward with their work. On top of that, AI's role in telemedicine and keeping track of patients remotely is getting more important, especially since it can handle a steady flow of health data and flag serious problems early.

Still, even though AI has a lot of promise in medicine, there are some hurdles holding it back from being fully used everywhere. Some of the main issues include figuring out how to pull together different types of data, keeping patient information private and secure, dealing with legal stuff, and making sure patient safety isn't compromised along the way. Furthermore, systems must address crucial issues of bias, transparency, and regulation before they can be widely adopted in clinical practice. There is currently a lack of empirical research on the financial implications (costs and profits) for healthcare organizations using AI. Researchers also note the need for further interdisciplinary study exploring the links between AI, data quality management, and ethical issues. There is a strong sentiment that AI can offer significant advantages, but future research must carefully analyze obstacles related to data integrity, patient safety, and privacy within the stringent regulatory environment of healthcare. Effectively reducing technical, organizational, ethical, data, policy, political, and legal challenges is a key area for future research.

2.3 Deep Learning for Symptom-Based Classification

Deep Learning (DL), a subfield of AI and ML, has profoundly impacted numerous domains due to its exceptional ability to learn complex patterns and representations from large and high-dimensional datasets. Unlike traditional ML methods that often require manual feature engineering, DL models, composed of multiple layers, can automatically extract hierarchical features directly from the raw data.

Our project specifically explores an AI-driven approach to symptom-based disease classification utilizing Deep Learning, focusing on an Artificial Neural Network (ANN) to learn the intricate relationships between symptoms and diseases. The core idea is to leverage the power of ANNs to replicate and potentially enhance the diagnostic decision-making process traditionally performed by humans. As outlined in Chapter 1, a typical ANN structure relevant to this task includes an input layer (receiving symptom data), hidden layers (learning intermediate patterns), activation functions (introducing non-linearity), and an output layer (producing disease probabilities). This review validates the design choice of using Artificial Neural Networks for symptom-based disease classification, aligning with current trends in the field. Comparative studies in related work have shown promising results using ANNs and other neural models for disease prediction.

2.4 Dataset Considerations, Curation, and Evaluation

AI models can do a good job in healthcare, but it all depends on the data they're trained with. If the data isn't solid or diverse, the model might end up being unreliable and inaccurate. In a lot of early research focused on classifying symptoms, many projects (including ones similar to this) often start by using symptom-disease datasets that are publicly available. These datasets usually show whether a symptom is present or not, or sometimes include a scale to measure how severe it is.

But there's a big problem that shows up in the literature — most of these datasets are either synthetic or come from non-clinical sources. That means they often miss the messy, unpredictable nature of real medical situations. To work around this issue, this project takes a more hands-on route by collecting actual symptom descriptions from reliable and well-known medical sources. Some of these sources are the World Health Organization (WHO), CDC, Mayo Clinic, MedlinePlus, and the Cleveland Clinic. Using info from these reliable places helps ensure the AI is learning from real medical knowledge instead of just basic or fake examples.

I messed around with the original data a bit. Since the dataset was so small, I added some random noise to it. It's a simple idea but it keeps things from getting boring and stops the model from getting stuck in one pattern. So, I went through the data I collected and cleaned it up. Then I ran the noise injection to sort of expand the variety and then formatted it for the neural net model. When I tested the thing I used a separate set of data — from proper medical sources like WHO and Mayo — to keep things fair. I didn't just look at the numbers; I looked at the results across different runs to see what was working and what wasn't.

2.5 Previous Work and Comparative Analysis

There's been quite a bit of work trying to use automated tools to predict diseases just from symptoms, mostly by using machine learning algorithms. One of the earlier projects by Gomathy and co-workers made a "Disease Predictor" system that used classic methods like Decision Trees, Random Forest, and Naive Bayes. They put it on a web app and tested it on a small group of diseases, like diabetes, malaria, jaundice, dengue, and tuberculosis, and got an accuracy of around 98.3% [4]. Later, Gomathy teamed up with Naidu and compared Random Forest and SVM classifiers on a bigger dataset with 25 diseases. They found Random Forest did slightly better, hitting nearly 99%, while SVM wasn't far behind at about 96.5%. Another study by Grampurohit and Sagarnal also tried Random Forest and got about 95% accuracy [5].

More recently, people have been exploring neural networks to improve how these symptom-based predictors work. Nesterov et al. introduced a neural model with logic regularization for symptom and diagnosis prediction, aiming to overcome limitations of traditional methods (like

Bayesian or Decision Trees, which can suffer from low relevance) and complex reinforcement learning models. Their model demonstrated improved performance, particularly in scenarios with large and sparse data, and was noted for its ease of implementation and training stability. However, a comparative analysis showed that their Feedforward Neural Network (FNN) model, while used for a larger number of classes (200 and 400), achieved lower accuracies (96.5% and 49% respectively) compared to other models [5]. Notably, their FNN architecture used a significantly larger number of neurons in hidden layers compared to the ANN approach utilized in this project (e.g., 6000+ and 3000 vs. 16 and 16 neurons in hidden layers).

This project's work, based on the comparative summary in the sources, demonstrates promising results and seems to improve on previous studies. When evaluating models (RF, KNN, SVM, FNN) on a dataset with 41 classes, the accuracies achieved were notably high: RF 99.05%, KNN 99.19%, SVM 98.51%, and FNN 99.19% [5]. Comparing these with previous works, particularly the FNN results which utilized a much less complex network architecture in our approach, indicates advancements in both accuracy and computational cost. The FNN model, despite having a longer execution time compared to KNN in a comparative analysis, was highlighted for its potential to handle more complex datasets in a real clinical context. The chosen ANN architecture aligns with these findings, focusing on efficiency while maintaining high accuracy. Other advanced techniques like Graph Neural Networks (GNNs) have also shown promise in symptom-based disease detection, with one study reporting 95% accuracy, suggesting avenues for future exploration but validating the high performance achievable with neural network approaches [5].

2.6 Broader Context: Mobile Health Implementation and Ethical Considerations

The scope of this project extends beyond model development to include the implementation of the trained model in a mobile application using React Native and Flask. This trend fits well with the growing use of AI models through easy-to-access platforms like mobile apps, showing how technology and practical healthcare solutions are moving forward together. But when it comes to putting AI into mobile health apps — especially those AI-powered symptom checkers (often called AISympCheck apps) — there are some important ethical and legal issues that can't be ignored. These apps can help more people get health info, but they might also reinforce the biases that already exist in healthcare. The problem usually comes from the data used to train the AI — if the data has biases, the AI will probably copy them, which is sometimes called a "garbage in, garbage out" issue [1]. Plus, the people who use these apps may not be representative of the population as a whole so the app's results may not be accurate or fair for everyone.

Another big issue is who's to blame when something goes wrong. With so many of these symptom checker apps and so little regulation around how reliable they are it's not clear who's responsible if the app gives wrong info or a wrong diagnosis. Is the app safe? Should doctors

be recommending them? With more and more of these apps popping up it’s clear health professionals need more guidance on how to deal with them.

People’s trust in AI tools, like symptom checkers, is often based on how transparent these systems are about how they work. Research shows that when users don’t really get how the system is coming up with results or when they get asked questions that seem off topic or repetitive they lose confidence in the app. That’s where Explainable AI, or XAI, comes in — it tries to make the AI’s “black box” a bit more transparent so users can see what’s going on behind the scenes. Making systems more transparent can really help users feel better about the whole experience [2].

Fixing these issues isn’t something that just one group can do. It calls for teamwork — from building in technical safety measures, training AI on less biased data, to setting up rules and regulations [1]. Groups like professional organizations and advocacy groups play a big role in creating guidelines and limits for using these symptom checker apps. Even though rules by themselves can't solve the deep-rooted widespread unfairness in healthcare well-thought-out laws and clear directions play a key role to lower dangers ensure proper AI design, and keep it in line with the real needs of patients and the public. Your work, which aims to build a dependable model checked against solid sources and strives to be user-friendly, fits into this tricky but vital area.

2.7 Summary and Research Gap

This literature review has established that AI and Machine Learning, particularly Deep Learning with ANNs, represent a promising direction for automating symptom-based disease classification. Existing research demonstrates the feasibility of achieving high accuracy in this task using various ML techniques. However, the review also highlights critical areas for advancement. The significant reliance on synthetic or non-clinical datasets in many studies presents a limitation for real-world applicability. While comparative studies exist, the landscape is continuously evolving with new architectures and techniques. Furthermore, the integration of such AI models into accessible platforms like mobile applications raises important ethical, legal, and usability challenges, including bias, accountability, and user trust, which stem partly from the lack of regulation and transparent.

By focusing on curating and utilizing data from verified external medical sources (WHO, CDC, Mayo Clinic, etc.) and incorporating data augmentation, this project tackles the limitation of relying solely on synthetic data, aiming for a model with better real-world applicability and generalization. The specific choice of an ANN architecture, validated against comparative studies, seeks to balance high accuracy with computational efficiency. Finally, the implementation of the model within a mobile application highlights the practical aspect of delivering AI-powered preliminary diagnosis to users, engaging with the opportunities and

challenges of the mHealth space. While acknowledging the broader ethical landscape, the project's focus on a validated, symptom-based approach aims to contribute a reliable tool for preliminary assessment. This literature review thus provides a strong theoretical and practical foundation for the methodology and objectives pursued in this study.

CHAPTER 3 METHODOLOGY

3.1 Data Collection

The foundation of any machine learning model lies in the quality, reliability, and diversity of the data used to train and evaluate it. For this project, the aim was to develop a robust AI-driven system capable of classifying diseases based on symptoms — a task that requires carefully curated and representative data reflecting real-world diagnostic patterns. So, they went with a two-part plan for their data: they used a well-known organized set of data for training and created fake medical data that looked real for testing. They made both these sources even better by using some clever tricks to make the AI learn more and not get too fixed on specific patterns.

3.1.1 Primary Dataset (SymbiPredict Dataset)

To begin training the neural network model, a dataset called “SymptomPredict” was selected. It’s hosted on Mendeley Data and, for the sake of this report, is referred to as the SymbiPredict Dataset. This dataset maps out various symptoms to their related diseases using a fairly straightforward binary format—basically marking symptoms as either present or not. In total, there are 53 symptoms and 10 diseases included.

Each entry in the dataset represents a unique pattern of symptoms associated with one condition, which works well for training classification models. What makes this dataset particularly useful is that it's built on information inspired by respected health organizations like the CDC, which gives it a certain degree of trustworthiness. It isn’t perfect, but the structure is obvious and easy to handle for learning tasks such as machine learning.

Furthermore, the corpus appears to be based on actual clinical observations as well as guided by medical experts’ opinions. Thus, this is not just any set of random symptom-disease pairings but how these kinds of presentations may emerge in real-life medical scenarios. This gives it a good first step towards any kind of AI model intending to predict diseases from symptoms.

3.1.2 Testing Dataset (Curated Web-Sourced Datasets)

To make sure the trained model could handle more than just synthetic or simplified data, a separate test set was put together using real-world information. This wasn’t done automatically—symptom and disease links were gathered manually from five trusted and well-known medical sources. The goal was to bring in data that better reflects the complexity and unpredictability seen in actual healthcare settings. By relying on sources recognized for clinical accuracy, the idea was to build a more realistic testing ground—something that could truly

measure how well the model might perform when used outside of a controlled, ideal environment:

- World Health Organization (WHO)
- Centers for Disease Control and Prevention (CDC)
- Mayo Clinic
- MedlinePlus
- Cleveland Clinic

From these sources, symptom patterns and disease associations were extracted to form an independent evaluation dataset. Each data sample was carefully converted into the same binary format used during training, making sure everything matched up with the model's input requirements. This new dataset was shaped to reflect real-world medical patterns and the way symptoms are typically recorded in clinical settings, making it a solid reference point for testing how well the ANN model can handle unfamiliar cases. By bringing in this outside data, the model's performance can be checked in a more realistic way, helping reduce bias and better show how it might work with actual diagnostic challenges.

3.1.3 Data Augmentation

To better reflect the uncertainty that often comes with real-life medical diagnoses—and to help the model learn to generalize—both the training and test datasets went through a carefully designed data augmentation process. This process had two main parts: grouping symptoms into categories and then applying a probabilistic encoding, followed by adding a bit of controlled randomness, or noise, to simulate variation in symptom reporting.

3.1.3.1 Symptom Categorization and Probabilistic Encoding

Symptoms were grouped into three levels, or tiers, depending on how often they typically appear in real medical cases based on clinical records and documentation:

- **Critical Symptoms**: Found in 90–100% of cases (e.g., chest pain for heart disease)
- **Partial Symptoms**: Present in 60–80% of cases (e.g., fever, fatigue)
- **Rare Symptoms**: Seen in only 30–40% of cases (e.g., muscle twitching, rash)

Based on this classification, new synthetic entries were created by assigning symptom values (either 0 or 1) in a way that reflects the natural variation seen in how patients actually show symptoms. This encoding method introduced controlled variation while preserving the underlying symptom-disease associations.

3.1.3.2 Synthetic Data Generation with Noise Injection

To replicate diagnostic inconsistency and patient self-reporting variation, **random noise** was injected into the data:

- For the Primary Dataset:
    - Critical Symptoms: 10% noise (small variability)
    - Partial Symptoms: 25% noise (moderate variability)
    - Rare Symptoms: 40% noise (high variability)

Using this scheme, 1,000 synthetic samples were generated for each of the 10 diseases. To further increase dataset robustness, the process was replicated 20 times, resulting in a final training dataset of **200,000 entries** (20,000 per disease).

- For the Testing Dataset:
    - Only **Critical** and **Partial** symptoms were considered (aligned with web-sourced data patterns)
    - Noise injection: 10% for Critical, 20% for Partial symptoms
    - 1,000 samples per disease were synthesized, totaling **10,000 test samples**

This augmentation approach not only diversified the symptom expressions but also closely mimicked the variability encountered in real clinical scenarios and online self-reported symptom checkers. Because of this, the ANN model was developed and evaluated using data that captured both the organized nature of clinical records and the unpredictable variations found in real medical cases.

This well-rounded approach from carefully selecting reliable datasets to introducing realistic variations through augmentation shows a clear focus on creating a model that performs well not just in theory but also in real-world healthcare settings where it can truly make a difference.

3.2 Data Pre-processing Techniques

In order for the artificial neural network (ANN) model to operate well, remain stable, and generalize on un-seen data, a careful and systematic pre-processing protocol was applied to both the primary training data set and the secondary testing data set. Pre-processing is an important aspect of any machine-learning workflow because the quality of the input data is very influential on the success of the model. Therefore, each step of the pre-processing process was deliberately selected in a manner consistent with the expectations and demands of deep learning models to prepare the raw data.

3.2.1 Feature-Label Separation

The first step in the pre-processing workflow involved a clear separation of input features from target labels. In both datasets, **53 binary symptom indicators** were treated as the **independent variables (features)**, and the associated disease class was assigned as the **dependent variable (label)**. This clear demarcation allowed for a supervised learning framework, where the ANN model could learn meaningful patterns from symptom combinations and associate them with specific disease classes.

Each row in the dataset thus became a 53-dimensional binary vector (composed of 0s and 1s) representing the presence or absence of symptoms for a hypothetical patient, accompanied by a single categorical label denoting the diagnosed disease.

3.2.2 Label Encoding

Since machine learning models cannot interpret categorical variables directly, disease names (e.g., “Diabetes”, “Flu”, “Migraine”) were encoded into **numerical labels** using **Label Encoding**. This transformation was performed using the LabelEncoder utility from the scikit-learn library. The encoder assigned a unique integer value to each disease class, effectively converting the categorical target variable into a numeric format suitable for loss functions like sparse categorical cross entropy used during training.

This approach ensured that the learning algorithm could treat disease labels as discrete class indices while maintaining compatibility with softmax-based output layers in the ANN.

3.2.3 Data Splitting

To develop and evaluate the model fairly, the dataset was **partitioned into three subsets**: training, validation, and testing. A stratified sampling strategy was used throughout the splitting process to preserve the class distribution and ensure balanced representation of all diseases across the subsets.

- **Initial Split (Training + Final Testing)**:
  The complete dataset (both primary and testing) was initially split into two equal halves:
  - 50% for training and validation
  - **50% for final model testing**
    This decision reflects the project's emphasis not just on training accuracy, but also on model generalization to unseen data.

- **Further Split (Training + Validation)**:
  The training portion was then further divided equally into:
    - **50% training samples** (used for gradient-based weight optimization)
    - **50% validation samples** (used to monitor the model's performance on unseen data during training epochs and to guide regularization)

Such multi-phase splitting helped mitigate overfitting and provided a realistic assessment of model robustness at various stages of development.

3.2.4 Standardization of Input Features

Although the original input features were binary (0 or 1), standardization was still employed using **StandardScaler** from scikit-learn. This scaler transforms the data to have a **mean of zero** and a **standard deviation of one**.

The rationale for applying standardization even to binary data lies in the operational behavior of artificial neural networks:

- Neural networks converge faster and more reliably when input features are normalized, as this leads to better-behaved gradients during backpropagation.
- Standardized inputs are also beneficial when regularization (e.g., dropout or weight decay) is used, as they reduce the sensitivity of the model to input scale variations and ensure a more uniform treatment across neurons.

Standardization was applied **after splitting the data** to avoid data leakage from the test set into the training process.

In conclusion, the pre-processing pipeline was deliberately designed to uphold **best practices in deep learning** and to reflect a real-world clinical application where accuracy and reliability are paramount. From proper feature engineering and label encoding to data stratification and normalization, every step aimed to enhance the model's capacity to learn robust, generalizable patterns. These efforts formed the technical groundwork upon which the ANN model could effectively operate — both during training and in deployment scenarios involving real symptom inputs.

3.3 ANN Model Architecture and Design

When constructing a rigorous system to classify diseases from symptoms, selecting the appropriate model structure is critical to obtain precise and reliable predictions. Given the input features are fairly simple - only depicting whether a symptom is present or not - and the goal is clearly to classify across 10 different diseases, it was logical to develop a custom Artificial Neural Network (ANN) model to be used specifically for this purpose. Subsequently, the model was trained and optimized carefully, particularly keeping in mind it might one day be used in the field.

3.3.1 Rationale for Choosing ANN

In this case, ANNs are the best fit because our data is in the form of a table showing symptom presence or absence. Unlike other machine learning techniques which may require manual feature extraction, ANNs are capable of uncovering intricate relationships without any external assistance. Likewise, they can be varied in size and shape, and prevention methods can be added to avoid overfitting, depending on the task. For these reasons and based on the previous success of ANNs in medical predictions, an ANN was selected as the primary model in this project.

3.3.2 Overview of Model Structure

The model was constructed using the **Keras Functional API** on top of **TensorFlow**, enabling full control over the architecture and training workflow of the model. The final architecture consists of:

- One input layer
- Three hidden layers (fully connected)
- **One output layer** (with softmax activation)

Each component is described in detail below.

3.3.3 Input Layer

he input layer receives a 53-dimensional binary feature vector representing presence (1) or absence (0) of a case's symptoms. The employment of a fixed-size vector maintains consistency in the dataset and also makes downstream processing easier.

3.3.4 Hidden Layers

The artificial neural network has three hidden layers fully connected (dense) with each containing 16 neurons and a Rectified Linear Unit (ReLU) activation function. ReLU has been chosen based on its utility in curtailing the vanishing gradient effect and also as a means to achieve non-linearity which is required for learning complex patterns.

As a regularization approach, each thick layer is followed by a 0.2 rate dropout layer. Dropout reduces overfitting by randomly turning off neurons during the training phase, and it is essential for providing the model adequate time to acquire new representations.

This layered design strikes a balance between **model complexity** and **computational efficiency**. By maintaining a consistent neuron count across layers and limiting layer depth,

the model avoids unnecessary overfitting and maintains suitability for deployment in real-time mobile applications.

3.3.5 Output Layer

The output layer is a **dense layer with 10 neurons**, corresponding to the 10 disease classes. A **softmax activation function** is applied, converting the raw output scores into a probability distribution over the classes. The model predicts the class with the highest probability as the final diagnosis.

3.3.6 Compilation Details

Before training, the model was compiled with the following configurations:

- **Optimizer: Adam**
  - Selected for its adaptive learning rate and robustness across a wide range of deep learning tasks.

- **Loss Function**: sparse_categorical_crossentropy
  - Used because the labels are integer-encoded (not one-hot encoded). It is appropriate for multi-class classification with mutually exclusive classes.

- **Evaluation Metric: accuracy**
  - Measures the percentage of correctly predicted labels during training and validation.

3.3.7 Model Summary

The final ANN architecture contained **approximately 1,578 trainable parameters**, which is computationally efficient while still sufficiently expressive for the complexity of the task. The compact design enables real-time inference without requiring high-end computing resources, making the model deployable on mobile and web platforms.

The model was first trained on a well-curated dataset of 200,000 examples, followed by evaluation on a separate set of 10,000 real-world cases. Throughout training, the artificial neural network (ANN) displayed consistent learning patterns and stable convergence, establishing a dependable core for the AI-driven symptom checker.

Ultimately, the ANN was carefully crafted to meet the needs of healthcare applications while harnessing the capabilities of advanced deep learning techniques. You can truly appreciate the model's thoughtful design in how every aspect—such as activation functions, layer sizes, and regularization—was chosen after extensive testing and analysis of real-world data.

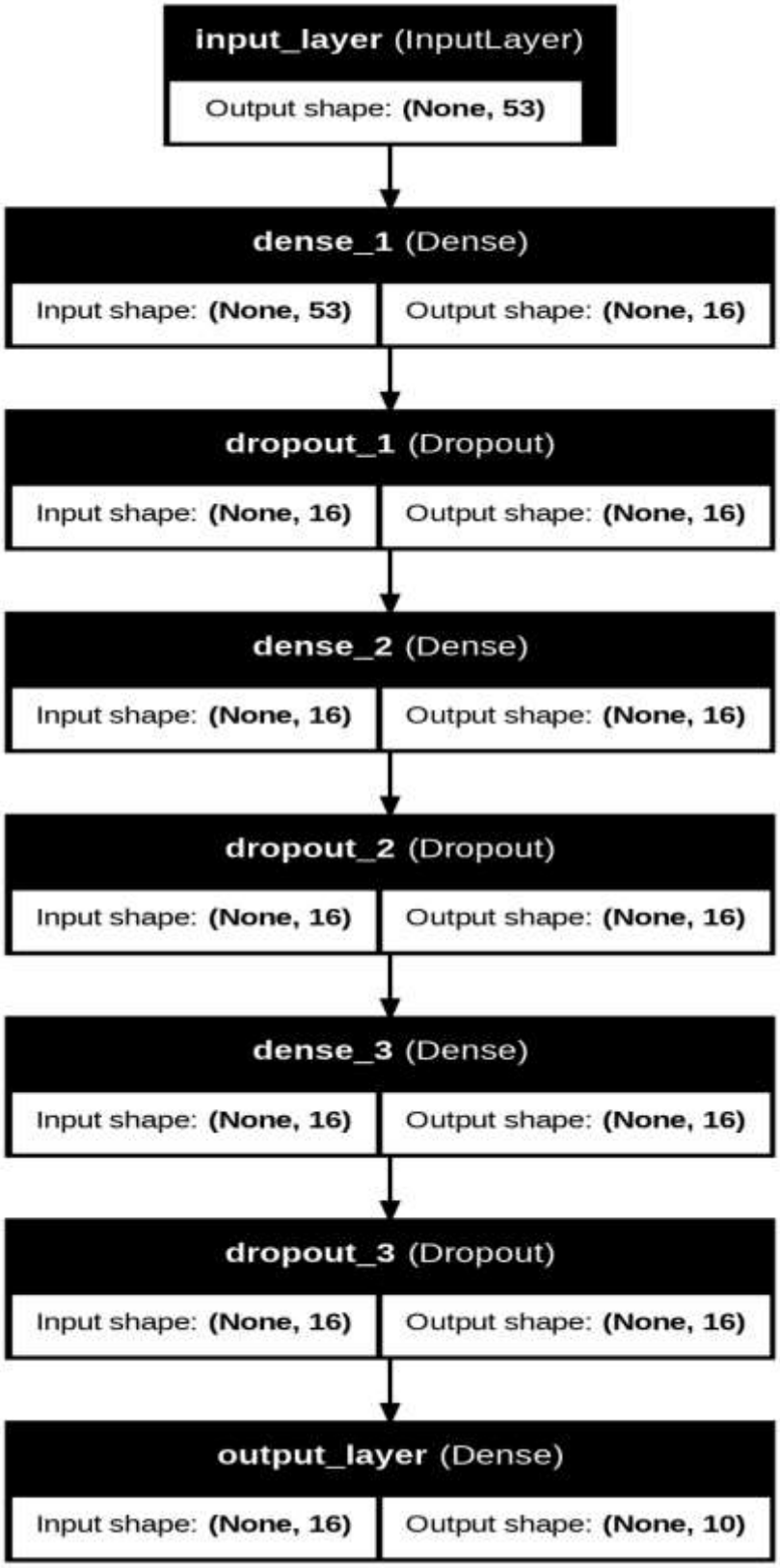

Figure 1: Model Architecture

3.4 Model Training Protocol

Deep learning model training encompasses much more than merely shoveling data into a model with hope on the other side. An attitude of careful consideration-adjustments to parameters, observation of model behavior, and verification of model learning in a way other than mere memorization-is what we see as being appropriate in this situation. We worked through the analysis of model behavior on a day-to-day basis, adjusting the learning rate, and batch size, and more, in the fine-tuning of hyperparameters to find what suited the needs of our project best. We did not merely wish to see good numbers; we wanted to create something that would see real-world action.

3.4.1 Training Strategy and Dataset Usage

The training executed a carefully crafted process to enhance the model's performance in accurately classifying 10 diseases using 53 binary symptom features. After pre-processing steps, the data was divided into three subsets while maintaining the distribution of each class within the subsets:

- **Training Set**: Used for model learning and weight optimization
- **Validation Set**: Used to monitor model performance during training and to fine-tune hyperparameters.
- **Testing Set**: Reserved for final evaluation and generalization testing (discussed in Section 3.6)

This approach provided a clean separation between learning and evaluation, reducing overfitting risk and ensuring that the model’s performance metrics reflect true predictive ability.

3.4.2 Batch Size and Epochs

The model was trained for 10 epochs with a batch size of 64. These values were selected through hyper-parameters tuning and reflect a balance between training efficiency and convergence stability.

- **Epochs**: The number of complete passes through the training dataset. A modest value of 10 was chosen to avoid overfitting, especially considering the size of the augmented dataset (200,000 samples).
- **Batch Size**: The number of samples processed before updating model weights. A batch size of 64 was found to offer an optimal trade-off between training speed and

model accuracy, especially on the available hardware.

3.4.3 Loss Function and Optimizer

The model was compiled with the following configurations to guide the learning process:

- **Loss Function**:
  sparse_categorical_crossentropy
  This function was chosen because the disease labels are encoded as integers rather than one-hot vectors. It is well-suited for multi-class classification problems with mutually exclusive classes.

- **Optimizer**:
  Adam (Adaptive Moment Estimation)
  Adam is an adaptive learning rate optimization algorithm that combines the benefits of RMSProp and momentum. Its ability to adjust learning rates on a per-parameter basis makes it especially effective for deep learning tasks where learning dynamics can vary significantly across layers.

- **Metric**:
  Accuracy was used to monitor the proportion of correct predictions during training and validation. It served as the primary indicator for model performance throughout the training process.

3.4.4 Validation Monitoring

During training, the model's performance was evaluated at the end of each epoch using the validation dataset. This real-time monitoring enabled:

- Early identification of overfitting trends.

- Improved selection of training hyperparameters.

- Confirmation that the model was generalizing well to unseen data.

Although callbacks such as **EarlyStopping** and **ModelCheckpoint** were available and prepared in the training pipeline, the model achieved convergence within 10 epochs without triggering these mechanisms. This demonstrates both the quality of the training data and the effectiveness of the architecture.

3.4.5 Training Implementation

The training loop was implemented using the fit () method of Keras, with the training and validation datasets passed as arguments. The method automatically handled batching, backpropagation, and gradient updates.

This function returned a history object containing loss and accuracy metrics for each epoch, which were later visualized using learning curves to assess model performance over time.

3.4.7 Observations During Training

The model demonstrated a smooth learning curve, with training and validation accuracy both increasing steadily across epochs. No significant divergence between training and validation loss was observed, indicating that:

- The model was not overfitting.
- The augmented dataset successfully captured generalizable patterns.
- The dropout layers and batch size were effective in maintaining model regularity.

With high confidence and consistency at the end of training, it was clear that the problem architecture and tuning parameters were well chosen for the task.

In summary, the process of training was more than just putting a good score on paper: It was a careful consideration of how to split data, control overfitting, and monitor progress all through. Thus, we are left with a model that has been efficiently trained and is ready to undergo real-life testing, which will be discussed in the next section.

Parameter	Value
Batch Size	64
Number of Epochs	10
Optimizer	Adam
Loss Function	Sparse Categorical Cross entropy
Learning Rate	Default (0.001)
Regularization Technique	Dropout (0.2 after each hidden layer)

Table 1: Model Training Protocols

3.5 Model Testing on External Datasets

The establishment of a powerful ANN for symptom-based disease classification relies not only on its performance on the training dataset but also on its ability to generalize to diverse scenarios in the real world. This introduced a very long test phase in an attempt to prove the model's generalization during testing in using five different datasets prepared independently of each other and sourced from some of the world's most internationally recognized medical institutions: World Health Organization (WHO), Centers for Disease Control and Prevention (CDC), Mayo Clinic, MedlinePlus, and Cleveland Clinic. This section describes the methodology for preparing and testing these external datasets, in order to maintain a fair and consistent evaluation framework. The detailed results from this testing phase are available in Chapter 4, Section 4.2, where the performance of the model is analyzed in detail.

3.5.1 Rationale for External Testing

The purpose of evaluating the ANN model on external datasets is to ascertain its real-life applicability, where symptomatology may differ in structure, granularity, and quality-from a clinical perspective. Whereas the internal SymbiPredict dataset was curated and enhanced for training purposes, external datasets reflect the variability in medical documentation from the real world. The use of clinically relevant reference datasets in the evaluation of the model ensures its clinically salient validity and thus greatly enhances the credibility of this model as a decision support system.

3.5.2 Dataset Curation and Sources

The external testing datasets were carefully generated from five reputable medical organizations, selected for their well known status and comprehensive symptom-disease documentation:

- **World Health Organization (WHO)**: Provides globally standardized symptom profiles for prevalent diseases, ensuring broad applicability.

- **Centers for Disease Control and Prevention (CDC)**: Offers detailed clinical data, particularly for infectious diseases.

- **Mayo Clinic**: Known for its patient-centric symptom descriptions, this source adds a practical perspective to the dataset.

- **MedlinePlus**: A trusted resource for consumer health information, contributing diverse symptom presentations.

- **Cleaveland Clinic**: Renowned for its clinical expertise, this source provides high-quality symptom-disease mappings.

Their corresponding structures had the same features of having 10,000 entries, which were specifically divided among the 10 categories: Allergy, Arthritis, Chickenpox, Dengue, Diabetes, Malaria, Migraine, Brain Haemorrhage, Pneumonia, and Typhoid, with disease entries in every other category constituting 1,000 each.

3.5.3 Evaluation Metrics

The performance of the ANN model on each external dataset was assessed using various standard classification metrics that were chosen in order to provide a broad assessment of diagnostic accuracy and reliability:

- **Accuracy**: The proportion of correct predictions across all instances, offering an overall measure of model performance.
- **Precision**: The ratio of correct positive predictions to total positive predictions, indicating the model's ability to avoid false positives.
- **Recall**: The ratio of correct positive predictions to all actual positives, reflecting the model's sensitivity to detecting true cases.
- **F1-Score**: The harmonic mean of precision and recall, providing a balanced measure of classification performance.
- **Confusion Matrix**: A visual representation of correct and incorrect predictions across classes, highlighting misclassification patterns.

These metrics were computed for both per class and as macro and weighted averages to calculate both class-specific and overall performance. The evaluation process was designed to ensure a thorough assessment of the model's generalization, with results analyzed in detail in Section 4.2.

CHAPTER 4 RESULTS & ANALYSIS

4.1 Evaluation on Internal Dataset

The internal (curated) dataset, built from the SymbiPredict repository and expanded considerably to simulate real-world diagnostic variability, formed the basis during the initial phase of evaluation upon which the trained Artificial Neural Network (ANN) model was tested. As a building block for training and internal validation of performance, the goal of this phase centered around testing how well the model may have learned to classify diseases using the binary-encoded symptom vectors.

4.1.1 Classification Report Analysis

To the classification report has added the most important key performance metrics which included precision, recall, F1 score and support for each of ten target disease classes. Precision measured the proportion of correct positive predictions among all predicted positives, recall assessed the model's sensitivity to detecting actual positives, and F1-score offered a balanced metric that accounted for both.

– Most classes have perfect precision, recall, and F1-score (1.0000), meaning there are no or negligible misclassifications.

–The lowest values observed are:

Class1: Recall is 0.9998.

Class7: Precision is 0.9998, and Recall is 0.9994.

These slight reductions indicate very few misclassifications.

–Macro Avg (0.9999) and Weighted Avg (0.9999) show that the model maintains strong performance across all classes.

• Accuracy: The overall accuracy is 99.99%, meaning the model correctly classifies nearly all instances.

Class	Precision	Recall	F1-Score
0	1.0000	1.0000	1.0000
1	1.0000	0.9998	0.9999
2	1.0000	1.0000	1.0000
3	1.0000	1.0000	1.0000
4	1.0000	1.0000	1.0000
5	1.0000	1.0000	1.0000
6	0.9994	1.0000	0.9997
7	0.9998	0.9994	0.9996
8	1.0000	1.0000	1.0000
9	1.0000	1.0000	1.0000
Accuracy		0.9999	
Macro Avg	0.9999	0.9999	0.9999
Weighted Avg	0.9999	0.9999	0.9999

Table 2: Classification Report – Internal Dataset

4.1.2 Confusion Matrix Analysis

The confusion matrix offers a visual breakdown of correct versus incorrect predictions across all disease classes. Most predictions fell along the diagonal (correct predictions), indicating strong classification accuracy. Misclassifications primarily occurred between classes with overlapping symptom profiles such as Migraine and Brain Hemorrhage, or Dengue and Malaria.

• The diagonal values (true positives) dominate, showing that nearly all predictions are correct.

• There are only a few minor misclassifications

– Class 1: 2 samples were misclassified.

– Class 7 6 samples were misclassified.

• Off-diagonal elements (misclassifications) are nearly zero, reinforcing that the model makes very few mistakes.

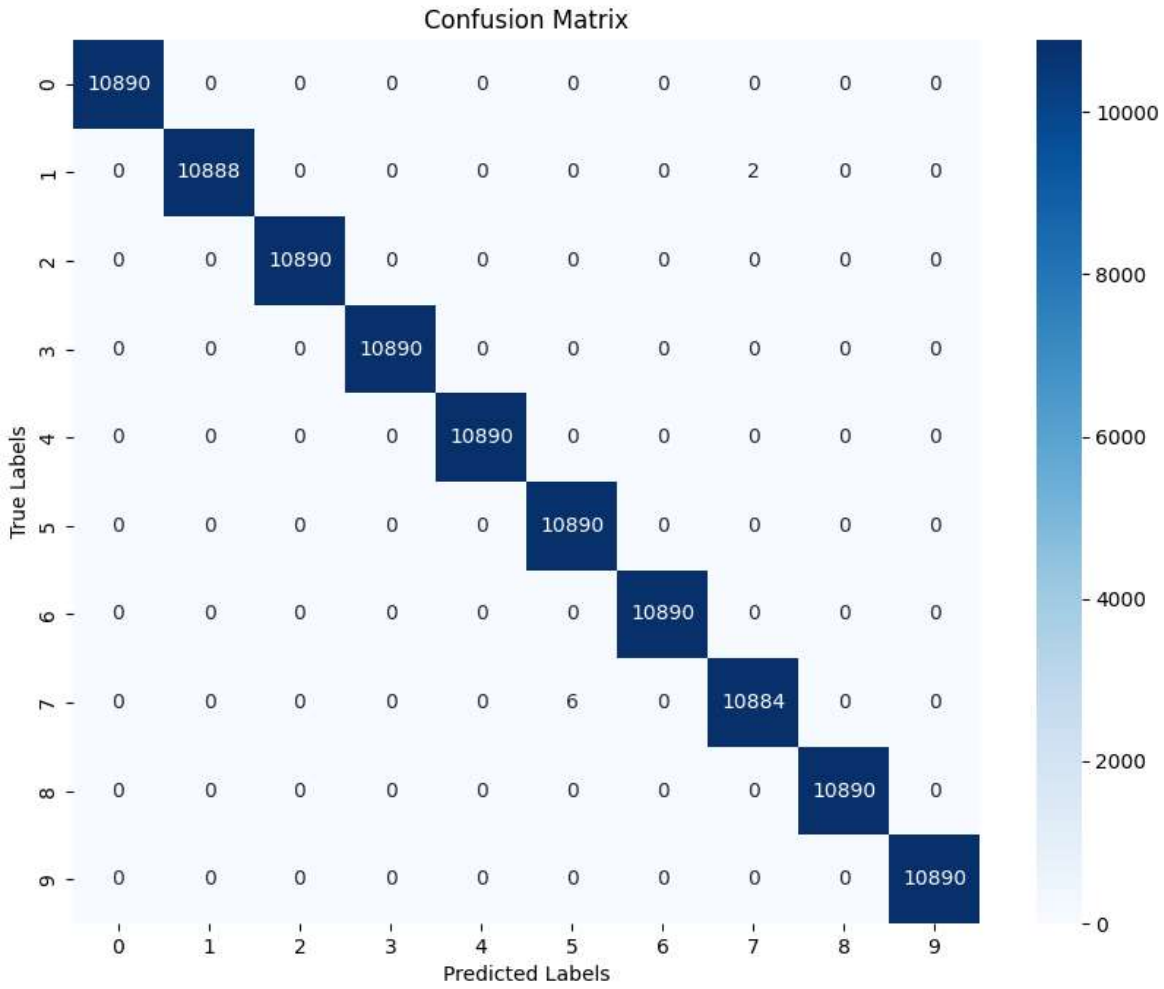

Figure 2: Confusion Matrix - Internal Dataset

4.2 Evaluation on External Dataset

To rigorously assess the model's generalization performance, we evaluated it on five independently curated external datasets sourced from globally respected medical organizations:

World Health Organization (WHO)
Centers for Disease Control and Prevention (CDC)
Mayo Clinic
MedlinePlus
Cleveland Clinic

Each dataset was constructed with 10,000 entries (1,000 per disease) and followed the same binary symptom encoding structure as the training set. All were preprocessed consistently—

including feature-label separation, label encoding, and standardization—to ensure compatibility with the ANN input layer. This external evaluation phase was essential for validating the model under varied data distributions and real-world clinical variability.

4.2.1 WHO Dataset

Classification Report Analysis:

• Precision, Recall, and F1-Score:

– Most classes exhibit high precision, recall, and F1-score, indicating strong classification performance.

– The lowest values observed are:

* Class 0 (Allergy): Precision = 0.7883, Recall = 0.972, F1-Score = 0.8706.

* Class 7 (Pneumonia): Precision = 0.8626, Recall = 0.998, F1-Score = 0.9235.

* Class 9 (Typhoid): Recall = 0.873, F1-Score = 0.9321.

* Class 8 (Brain Hemorrhage): Precision = 1.000, Recall = 0.763, F1-Score = 0.8656.

– These lower values suggest some degree of misclassification, particularly in Allergy, Arthritis, Typhoid, Pneumonia, and Brain Hemorrhage cases.

– Macro Avg (0.9475) and Weighted Avg (0.9388) indicate the model maintains good overall classification performance.

• Accuracy:

– The overall accuracy is 93.87%, meaning the model correctly classifies about 93.87% of instances.

	Precision	Recall	F1-Score	Support
0	0.788321	0.972	0.870578	1000
1	1.000000	0.916	0.956159	1000
2	0.945894	0.979	0.962162	1000
3	0.998899	0.998	0.998499	1000
4	1.000000	0.999	0.999500	1000
5	0.879839	0.989	0.931262	1000
6	0.998899	0.999	0.94687	1000
7	0.862576	0.998	0.923559	1000
8	1.000000	0.763	0.865557	1000
9	1.000000	0.873	0.932194	1000
Accuracy	0.9387	0.9387	0.9387	0.9387
Macro Avg	0.947457	0.9387	0.938815	10000
Weighted	0.947457	0.9387	0.938815	10000

Table 3: Classification Report – WHO Dataset

Confusion Matrix Analysis:

1. Allergy:

• 972 instances were correctly classified as Allergy (true positives).

• 4instances were misclassified as other diseases like Malaria.

• 24 instances were mistakenly classified as Brain Haemorrhage.

2. Arthritis:

• 916 instances were correctly classified as Arthritis.

• 70 instances were misclassified as Malaria.

• Small amounts were incorrectly classified as other diseases like Brain Haemorrhage (11 cases), Dengue (1 case), Allergy (2 cases).

3. Chickenpox:

• 979 instances were correctly classified as Chickenpox.

• A few cases were misclassified as other diseases, like Malaria (1 case) and Allergy (20 cases).

4. Dengue:

• 998 instances of Dengue were correctly identified.

• Only 2 cases were misclassified as Malaria.

5. Diabetes:

• 999 instances of Diabetes were correctly identified.

• Only one case was misclassified as Migraine.

6. Malaria:

• 989 instances of Malaria were correctly classified.

• 8 cases were misclassified as Diabetes or Pneumonia.

7. Migraine:

• 900 instances of Migraine were correctly identified.

• 97 instances were misclassified, primarily as Brain Haemorrhage, and 3 cases were misclassified as Allergy.

8. Brain Haemorrhage:

• 998 instances of Brain Haemorrhage were correctly classified.

• Only 2 instances were misclassified as Allergy.

9. Pneumonia:

• 763 cases were correctly classified as Pneumonia.

• 231 cases were misclassified as Allergy, 5 cases as Brain Haemorrhage, whereas 1 case was misclassified as Malaria. 6 10. Typhoid:

• 873 instances of Typhoid were correctly identified.

• There were small misclassifications in other categories, including Malaria (57 cases), Chickenpox (56 cases), and Brain Haemorrhage (14 cases).

• Key Observations:

– The model performs well in most categories with high accuracy, as most of the values along the diagonal (which represent correct classifications) are high.

– However, certain diseases like Pneumonia, Typhoid, and Arthritis have more misclassifications.

– Pneumonia and Typhoid, in particular, are misclassified as multiple other diseases.

– Migraine also sees some misclassifications but is otherwise fairly accurate.

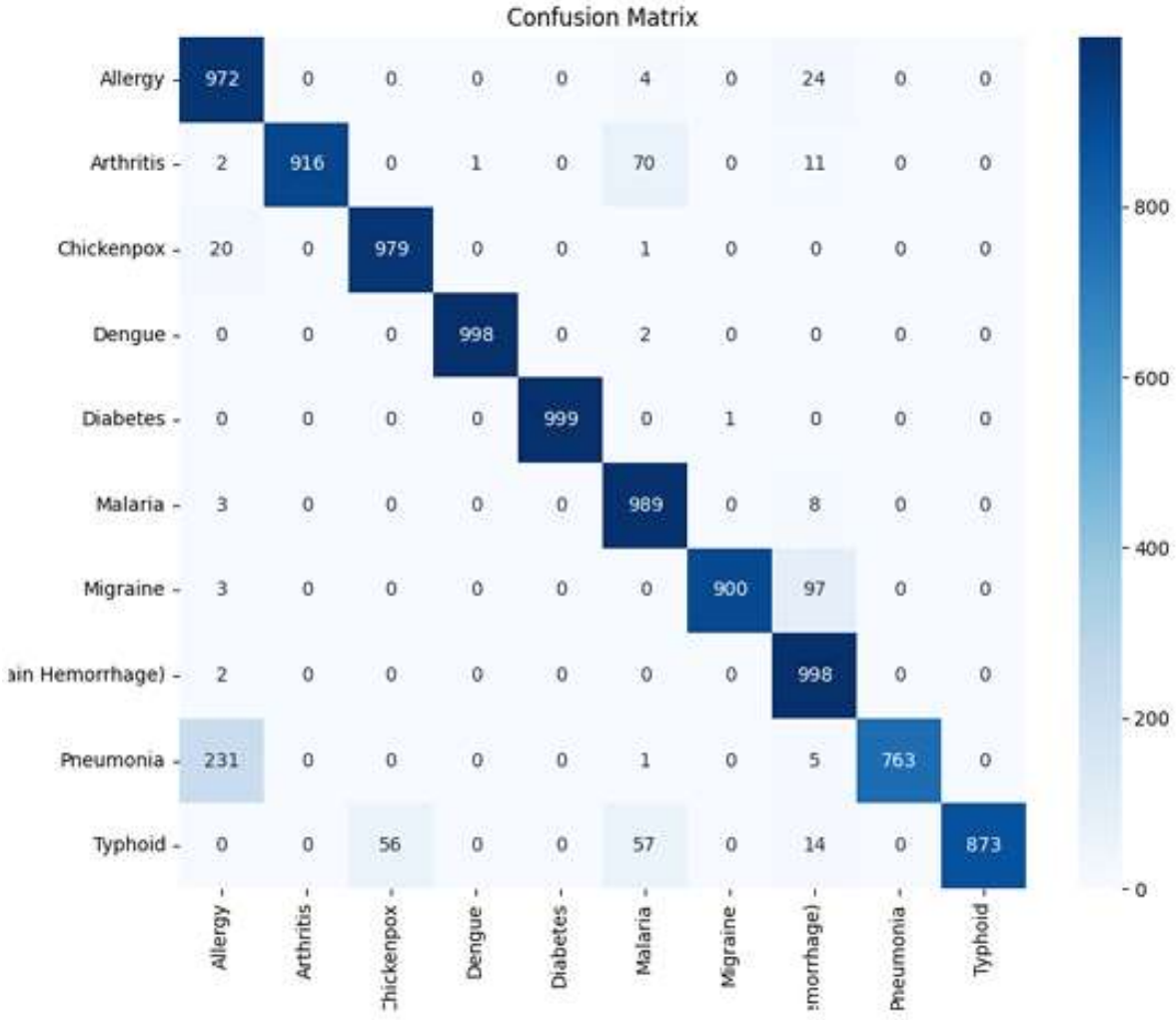

Figure 3: Confusion Matrix - WHO Dataset

4.2.2 CDC Dataset

Classification Report Analysis:

• Most classes perform well, with high precision, recall, and F1-scores.

• The lowest-performing classes are:

– Class 0 (Allergy): Precision = 0.7992, Recall = 0.86, F1-score = 0.8285.

– Class 7 (Migraine): Precision = 0.8474, Recall = 1.00, F1-score = 0.9174.

– Class 8 (Brain Haemorrhage): Precision = 1.00, Recall = 0.785, F1-score = 0.8796.

• The model struggles most with Class 8, where the recall is relatively low (0.785), meaning many instances of this class were misclassified. Lower recall suggests the model is missing the actual positives (False Negatives).

• Overall Performance:

– Macro Average: Precision = 0.9491, Recall = 0.943, F1-score = 0.943.

– Weighted Average: Precision = 0.9491, Recall = 0.943, F1-score = 0.943.

– Overall Accuracy: 94.3%, meaning the model classifies most instances correctly.

	Precision	Recall	F1-Score	Support
0	0.799257	0.860	0.828516	1000
1	0.998976	0.976	0.987355	1000
2	0.979167	0.987	0.983068	1000
3	1.000000	0.891	0.942359	1000
4	1.000000	0.990	0.994975	1000
5	0.872600	1.000	0.931966	1000
6	0.993884	0.975	0.984351	1000
7	0.847458	1.000	0.917431	1000
8	1.000000	0.785	0.879552	1000
9	1.000000	0.966	0.982706	1000
Accuracy	0.943	0.943	0.943	0.943
Macro Avg	0.949134	0.943	0.943228	10000
Weighted	0.949134	0.943	0.943228	10000

Table 4: Classification Report – CDC Dataset

Confusion Matrix Analysis:

1. Allergy:

• 860 instances were correctly classified as Allergy (i.e., True Positives).

• 140 instances were mistakenly classified as Brain Haemorrhage.

2. Arthritis:

• 976 instances were correctly classified as Arthritis.

• 24 instances were misclassified as Malaria.

3. Chickenpox:

• 987 instances were correctly classified as Chickenpox.

• A few cases were misclassified as other diseases, like Allergy (9 cases), Malaria (2 cases), and Brain Haemorrhage (2 cases).

4. Dengue:

• 891 instances of Dengue were correctly identified.

• 109 cases were misclassified as Malaria.

5. Diabetes:

• 990 instances of Diabetes were correctly identified.

• A few cases were misclassified as other diseases, like Allergy (2 cases), Arthritis (1 case), Migraine (4 cases), and Brain Haemorrhage (3 cases).

6. Malaria:

• All 1000 instances were correctly classified as Malaria (i.e., True Positives).

7. Migraine:

• 975 instances of Migraine were correctly identified.

• 24 instances were misclassified, primarily as Brain Haemorrhage, whereas 1 case was misclassified as Malaria.

8. Brain Haemorrhage:

• All 1000 instances were correctly classified as Brain Haemorrhage (i.e., True Positives).

9. Pneumonia:

• 785 cases were correctly classified as Pneumonia.

• 205 cases were misclassified as Allergy, 8 cases as Malaria, whereas 2 cases were misclassified as Brain Haemorrhage.

10. Typhoid:

• 966 instances of Typhoid were correctly identified.

• 21 cases were misclassified as Chickenpox, 2 cases as Malaria, 2 cases as Migraine, and 9 cases as Brain Haemorrhage.

• Key Observations:

– Class 8 (Pneumonia) and Class 0 (Allergy) have the highest misclassification rates.

– Classes like Malaria, Brain Haemorrhage, and Diabetes perform almost perfectly, with very few errors.

– The model struggles most with diseases that have overlapping symptoms (e.g., Pneumonia vs. Allergy).

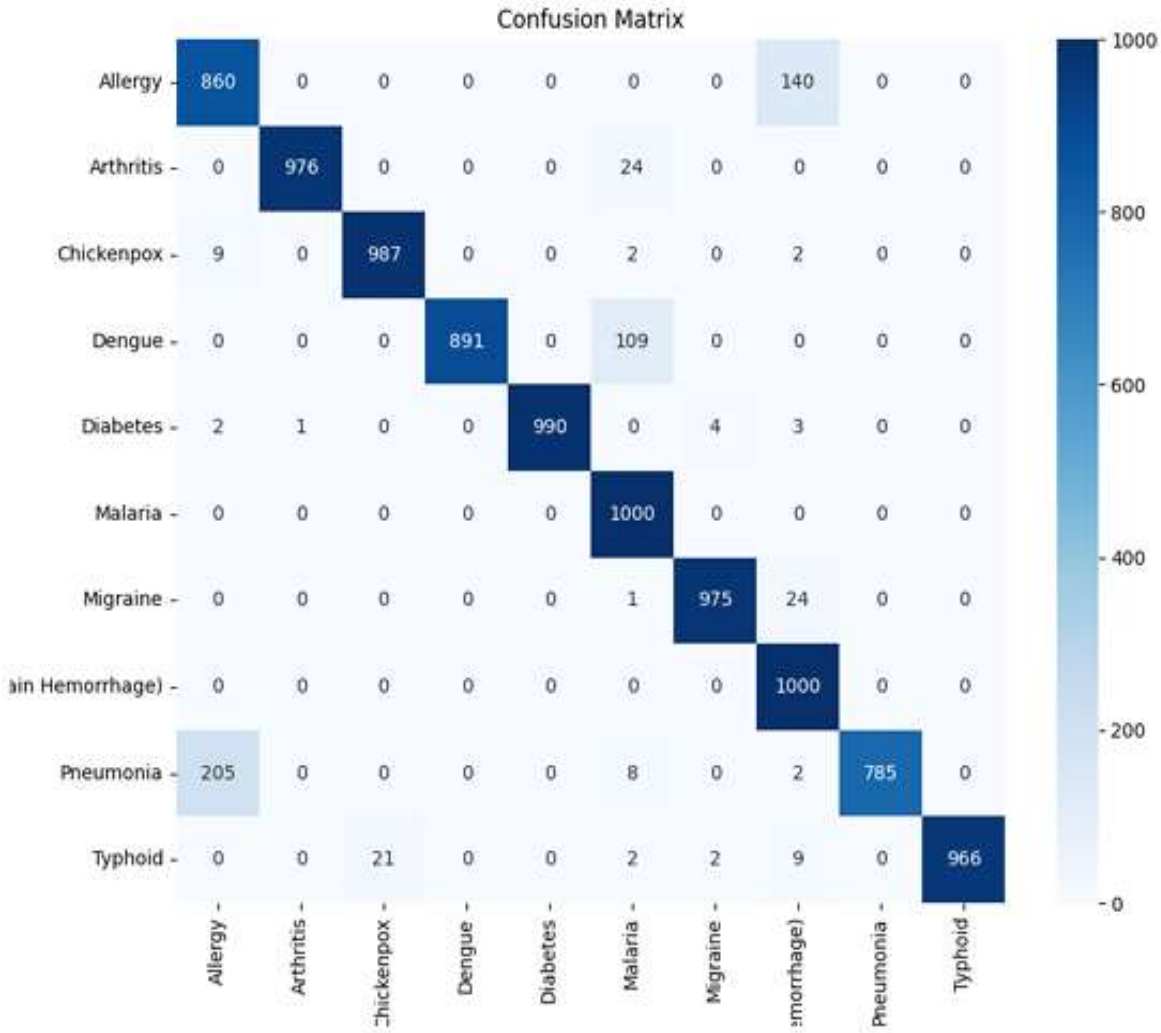

Figure 4: Confusion Matrix - CDC Dataset

4.2.3 Mayo Clinic Dataset

Classification Report Analysis:

• Precision, Recall, and F1-Score:

– Most classes have high scores, but some show slight drops.

– The lowest values observed:

* Class 5: Precision = 0.7614, Recall = 0.935, F1-score = 0.8393, showing notice able misclassifications.

* Class 7: Precision = 0.8388, Recall = 0.999, F1-score = 0.9119, suggesting some incorrect predictions.

* Class 9: Recall = 0.79, F1-score = 0.8827, indicating lower performance in this class.

– Macro Avg (0.9398) and Weighted Avg (0.9398) show good performance across all classes.

• Accuracy:

– The overall accuracy is 93.02%, showing that the model performs well but has some misclassifications.

	Precision	Recall	F1-Score	Support
0	0.878546	0.991	0.931391	1000
1	0.987903	0.980	0.983396	1000
2	0.996004	0.997	0.996502	1000
3	0.948798	0.908	0.927951	1000
4	1.000000	0.990	0.994975	1000
5	0.761401	0.905	0.839318	1000
6	0.98913	0.819	0.896061	1000
7	0.838791	0.999	0.911912	1000
8	0.997765	0.893	0.942448	1000
9	1.000000	0.790	0.882682	1000
Accuracy	0.9302	0.9302	0.9302	0.9302
Macro Avg	0.939834	0.9302	0.930721	10000
Weighted	0.939834	0.9302	0.930721	10000

Table 5: Classification Report – Mayo Clinic Dataset

Confusion Matrix Analysis:

1. Allergy:

• 991 instances were correctly classified as Allergy (i.e., True Positives).

• 9 instances were mistakenly classified as Brain Haemorrhage.

2. Arthritis:

• 980 instances were correctly classified as Arthritis.

• 14 instances were misclassified as Allergy, and 6 instances were misclassified as Malaria.

3. Chickenpox:

• 997 instances were correctly classified as Chickenpox.

• A few cases were misclassified as other diseases, like Malaria (2 cases) and Allergy (1 case).

4. Dengue:

• 908 instances of Dengue were correctly identified.

• 87 cases were misclassified as Malaria, and 5 cases were misclassified as Allergy.

5. Diabetes:

• 990 instances of Diabetes were correctly identified.

• Only one case was misclassified as Brain Haemorrhage and 9 cases as Migraine.

6. Malaria:

• 935 instances were correctly classified.

• 49 cases were misclassified as Dengue, 14 cases were misclassified as Allergy, whereas 2 cases were misclassified as Pneumonia.

7. Migraine:

• 819 instances of Migraine were correctly identified.

• 181 instances were misclassified, primarily as Brain Haemorrhage.

8. Brain Haemorrhage:

• 999 instances of Brain Haemorrhage were correctly classified.

• Only 1 instance was misclassified as Arthritis.

9. Pneumonia:

• 893 cases were correctly classified as Pneumonia.

• 103 cases were misclassified as Allergy, 3 cases as Chickenpox, whereas 1 case was misclassified as Malaria.

10. Typhoid:

• 790 instances of Typhoid were correctly identified.

• 197 cases were misclassified as Malaria, 11 cases were misclassified as Arthritis, whereas 1 case was misclassified as Chickenpox, and 1 as Brain Haemorrhage.

• Key Observations:

– Certain diseases like Pneumonia, Typhoid, and Migraine have more misclassifications.

– Pneumonia and Typhoid, in particular, are misclassified as multiple other diseases.

– Dengue also sees some misclassifications.

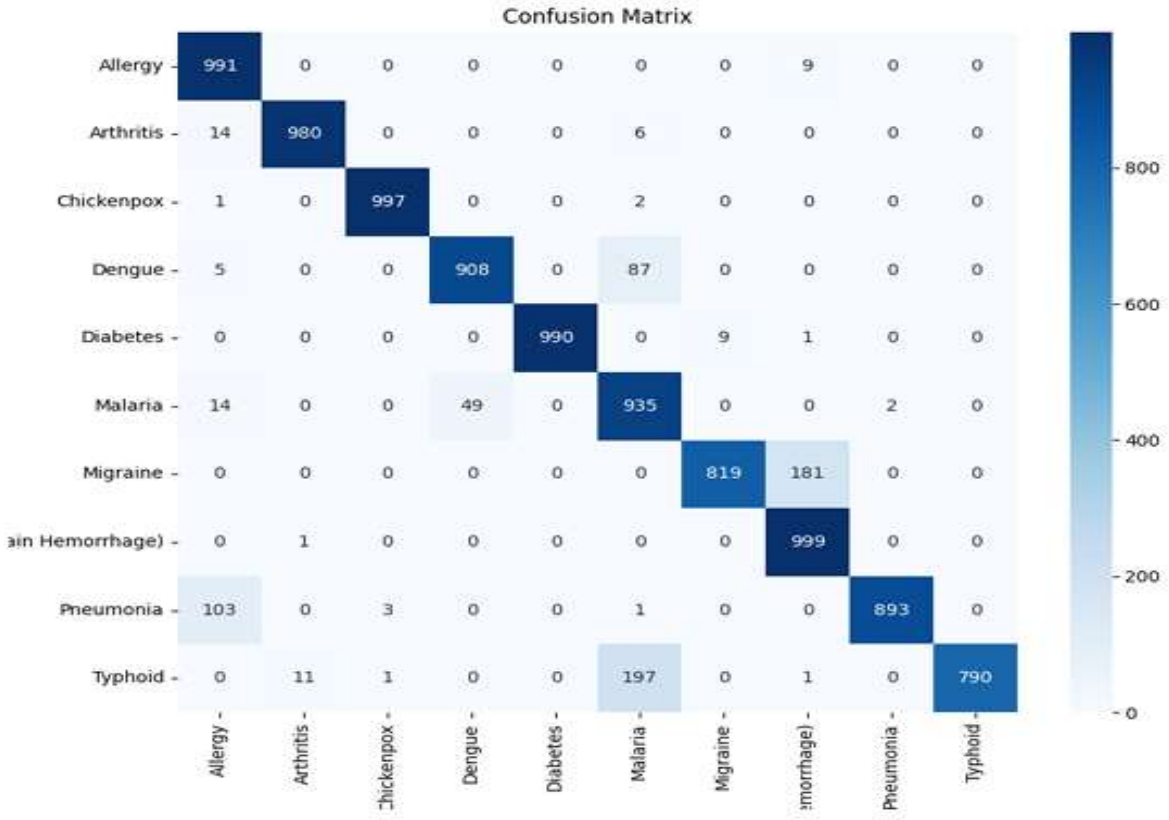

Figure 5: Confusion Matrix - Mayo Clinic Dataset

4.2.4 MedlinePlus Dataset

Classification Report Analysis:

• Precision, Recall, and F1-Score:

– Performance is weaker than the first model.

– Lowest Scores Observed:

* Class 0: Precision = 0.546, Recall = 1.0, F1-score = 0.706 (severe imbalance). Lower precision suggests that the model is incorrectly classifying other classes as this one (False Positives are high).

* Class 8: Precision = 0.994, but Recall = 0.179, leading to an extremely low F1 score of 0.303. Lower recall suggests the model is missing the actual positives (False Negatives).

	precision	recall	f1-score	support
0	0.546747	1	0.706964	1000
1	0.995685	0.923	0.957966	1000
2	0.945545	0.955	0.950249	1000
3	0.996637	0.889	0.939746	1000
4	0.876909	0.976	0.923805	1000
5	0.821018	1	0.901713	1000
6	1	0.852	0.920086	1000
7	0.943396	1	0.970874	1000
8	0.994444	0.179	0.30339	1000
9	1	0.919	0.957791	1000
accuracy	0.8693	0.8693	0.8693	0.8693
macro avg	0.912038	0.8693	0.853258	10000
weighted	0.912038	0.8693	0.853258	10000

Table 6: Classification Report – MedlinePlus Dataset

Confusion Matrix Analysis:

1. Allergy:

• All 1000 instances were correctly classified as Allergy (i.e., True Positives).

2. Arthritis:

• 923 instances were correctly classified as Arthritis.

• 60 instances were misclassified as Malaria.

• Minor misclassifications included: 3 cases as Allergy, 8 cases as Chickenpox, 3 as Dengue, and 3 as Brain Haemorrhage.

3. Chickenpox:

• 955 instances were correctly classified as Chickenpox.

• Some cases were misclassified as: Allergy (26 cases), Malaria (18 cases), and Arthritis (1 case).

4. Dengue:

• 889 instances of Dengue were correctly identified.

• 111 cases were misclassified as Malaria.

5. Diabetes:

• 976 instances of Diabetes were correctly identified.

• Some cases were misclassified as: Brain Haemorrhage (20 cases), Allergy (2 cases), and Arthritis (2 cases).

6. Malaria:

• All 1000 instances were correctly classified as Malaria (i.e., True Positives).

7. Migraine:

• 852 instances of Migraine were correctly identified.

• 137 instances were misclassified, primarily as Diabetes, while 11 cases were misclassified as Brain Haemorrhage.

8. Brain Haemorrhage:

• All 1000 instances were correctly classified as Brain Haemorrhage (i.e., True Positives).

9. Pneumonia:

• Unfortunately, only 197 cases were correctly classified as Pneumonia.

• 798 cases were misclassified as Allergy, 15 cases as Malaria, 7 cases as Brain Haemorrhage, and 1 case as Arthritis.

10. Typhoid:

• 919 instances of Typhoid were correctly identified.

• 47 cases were misclassified as Chickenpox, 14 cases as Malaria, 19 cases as Brain Haemorrhage, and 1 case as Pneumonia.

• Key Observations:

– Certain classes show severe misclassification issues, indicating possible data imbalance or model weaknesses:

* Pneumonia: 798 samples misclassified as Allergy – a major issue.

* Migraine: 137 samples misclassified as Diabetes.

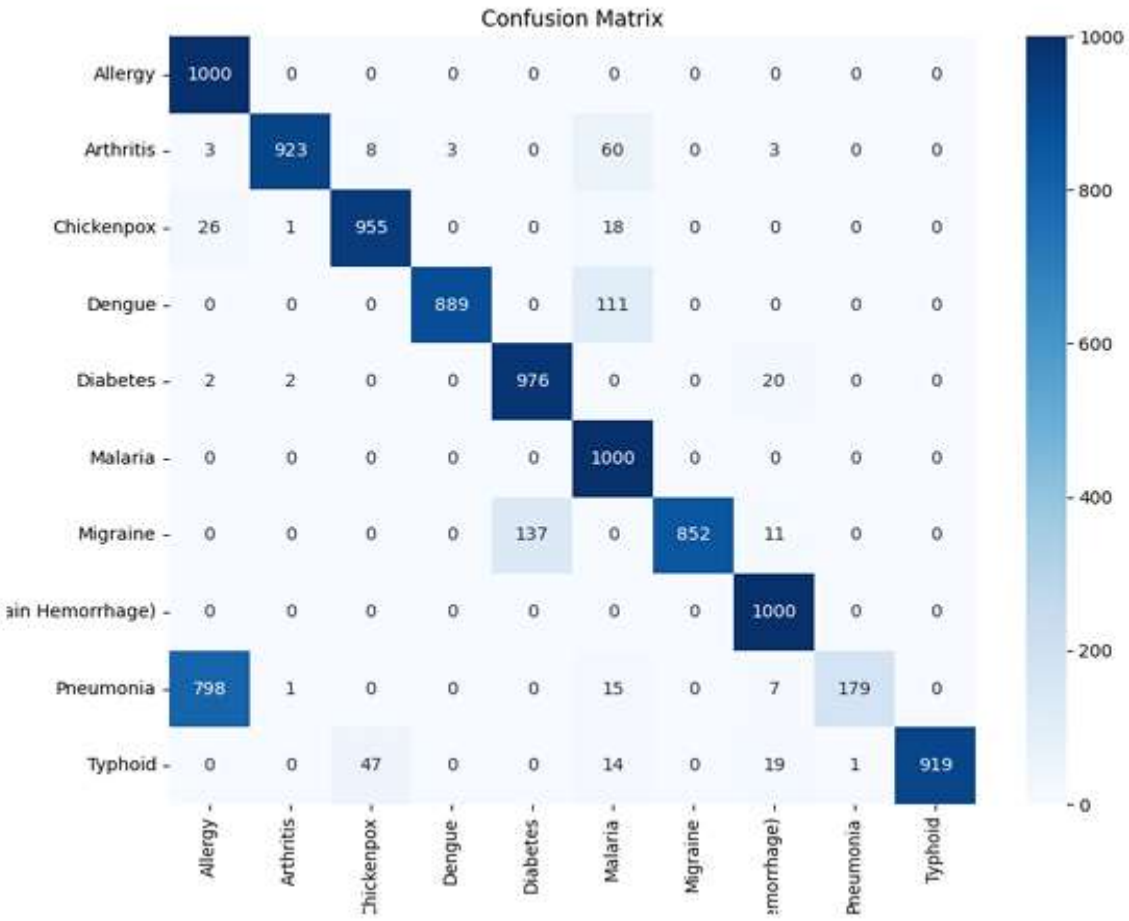

Figure 6: Confusion Matrix - MedlinePlus Dataset

4.2.5 Cleveland Clinic Dataset

Classification Report Analysis:

• Precision, Recall, and F1-Score:

– Most classes have strong performance, but there are notable weaknesses:

* Class 7: Precision is 0.544, and recall is 1.000, leading to a low F1-score of 0.704. This suggests that many other classes are incorrectly classified as Mi graine (i.e., False Positives are high).

* Class 8 (Brain Haemorrhage): Precision is 1.000, but recall is 0.187, meaning most instances of this class are misclassified as other classes (i.e., False Negatives are high).

* Class 9 (Pneumonia): Precision is 1.000, but recall is only 0.680, indicating a significant number of Pneumonia cases are misclassified. Lower recall suggests that False Negatives are high.

– Macro Avg:

* Precision: 0.908

* Recall: 0.862

* F1-score: 0.845

* These values indicate that performance varies significantly across different classes.

– Weighted Avg:

* Precision: 0.908

* Recall: 0.862

* F1-score: 0.845

* This suggests that even when weighted by class distribution, misclassifications affect the model's overall performance.

– Accuracy:

* The overall accuracy is 86.24%, meaning that 86.24% of all classifications are correct.

* This is decent but indicates room for improvement.

	precision	recall	f1-score	support
0	0.835422	1	0.910332	1000
1	0.998812	0.841	0.913138	1000
2	0.992	0.992	0.992	1000
3	0.912281	0.988	0.948632	1000
4	1	0.98	0.989899	1000
5	0.804037	0.956	0.873458	1000
6	1	0.187	0.31508	1000
7	0.54407	1	0.704722	1000
8	0.996016	1	0.998004	1000
9	1	0.68	0.809524	1000
accuracy	0.8624	0.8624	0.8624	0.8624
macro avg	0.908264	0.8624	0.845479	10000
weighted	0.908264	0.8624	0.845479	10000

Table 7: Classification Report – Cleaveland Clinic Dataset

Confusion Matrix Analysis:

1. Allergy:

• All 1000 instances were correctly classified as Allergy (i.e., True Positives).

2. Arthritis:

• 841 instances were correctly classified as Arthritis.

• 141 instances were misclassified as Allergy, whereas 18 cases were misclassified as Brain Hemorrhage.

3. Chickenpox:

• 992 instances were correctly classified as Chickenpox.

• A few cases were misclassified as other diseases, like Allergy (4 cases) and Malaria (4 cases).

4. Dengue:

• 988 instances of Dengue were correctly identified.

• 12 cases were misclassified as Malaria.

5. Diabetes:

• 980 instances of Diabetes were correctly identified.

• A few cases were misclassified as other diseases, like Allergy (9 cases), Arthritis & Chickenpox (1 case), Brain Hemorrhage (5 cases), and Pneumonia (4 cases).

6. Malaria:

• 956 instances were correctly classified as Malaria.

• 43 instances were misclassified as Malaria, whereas 1 case was misclassified as Brain Hemorrhage.

7. Migraine:

• Major misclassification – only 187 instances were correctly classified as Migraine.

• 813 instances were misclassified as Brain Hemorrhage.

8. Brain Haemorrhage:

• All 1000 instances were correctly classified as Brain Hemorrhage (i.e., True Positives).

9. Pneumonia:

•All 1000 instances were correctly classified as Brain Hemorrhage (i.e., True Positives).

10. Typhoid:

• 680 instances of Typhoid were correctly identified.

• 217 cases were misclassified as Malaria, 95cases were misclassified as Dengue, whereas only 1 case was misclassified as Brain Hemorrhage.

•Key Observations:

–Migraine and Pneumonia suffer from significant misclassification errors.

–Most of the Migraine cases were misclassified as Brain Hemorrhage, which corresponds to the overlapping symptoms between the two.

–Most other diseases have high classification accuracy, with Allergy, Chickenpox, and Brain Haemorrhage performing best.

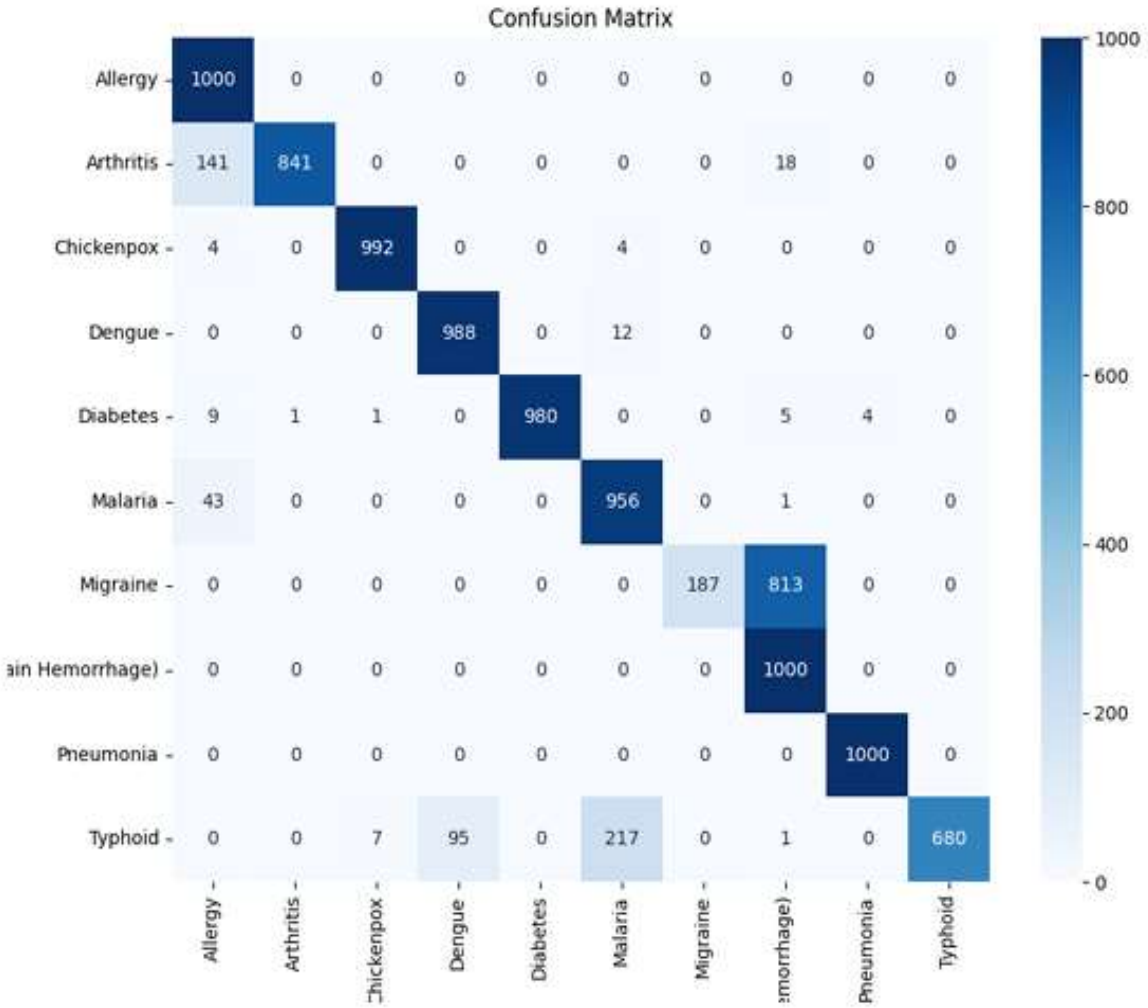

Figure 7: Cleaveland Clinic Dataset

4.3 Statistical Analysis & Comparison

The performance evaluation of the trained ANN model on both the internal (SymbiPredict dataset) and external datasets (WHO, CDC, Mayo Clinic, MedlinePlus, and Cleveland Clinic is crucial for understanding the model performance. This section outlines an extensive statistical analysis in terms of class-wise performance and overall performance metrics of the model, which helps in determining its strengths, weaknesses and generalization ability. Furthermore, the analysis reinforces the model's credibility as a solid approach to symptom-based disease classification system which also reflects the project's commitment to advancing AI-driven healthcare solutions.

4.3.1 Comparison of Classwise Performance Metrics

A class-wise comparison of precision, recall, and F1-score was conducted across all the six datasets (one internal and five external) to evaluate the model's performance across multiple

disease classes,. This extensive analysis reveals how well the model differentiates among the different disease classes - Allergy, Arthritis, Chickenpox, Dengue, Diabetes, Malaria, Migraine, Brain Haemorrhage, Pneumonia, and Typhoid - under varying data conditions. Since the F1-score provides an average of precision and recall, the table below summarizes the F1-scores of the classes across the datasets, ultimately capturing the model's ability to balance false positives and false negatives.

Analysis of Class-wise Performance:

- **Consistent High Performing Disease Classes**: classes such as Chickenpox, Dengue, and Diabetes persist F1-scores above 0.92 across all six datasets, except a near-perfect scores (1.000) on the internal dataset. Malaria, while generally a high performer, achieved perfect F1-scores on multiple datasets (e.g., CDC and MedlinePlus), however it showed a notable dip of 0.873 on the Cleveland Clinic dataset. These results elucidate that such diseases have relatively distinct and well-represented symptom vectors, enabling the model to form robust and generalizable decision in most settings.
- **Variable Performers**: Pneumonia, Migraine, and Brain Haemorrhage show significant variability. Migraine's F1-score falls to 0.32 on the Cleveland Clinic dataset, with 813 instances misclassified as Brain Haemorrhage. Pneumonia itself has an F1-score of 0.30 on MedlinePlus due to a recall of 0.18, reflecting severe misclassification (798 instances misclassified as Allergy). indicating the model struggles to identify true positives for this class in that dataset.
- **Allergy Challenges**: Allergy consistently shows lower F1-scores (e.g., 0.71 on MedlinePlus, 0.83 on CDC), often due to low precision (e.g., 0.55 on MedlinePlus). This suggests the model over-predicts Allergy, misclassifying other diseases as Allergy, particularly Pneumonia.
- **Typhoid and Arthritis**: These classes show moderate variability, with F1-scores dipping to 0.81 (Typhoid, Cleveland Clinic) and 0.91 (Arthritis, Cleveland Clinic). Misclassifications often involve Malaria, likely due to overlapping symptoms like fever and fatigue.

This thorough class-wise analysis highlights the model's robustness for diseases with clear symptom profiles whereas it identifies the challenges with diseases having symptom overlap. The variability emphasizes the impact of dataset-specific characteristics, such as symptom granularity and noise, on model performance. These insights are important for making future improvements, such as targeted data augmentation or feature weighting to address misclassification in problematic disease classes.

Class	Internal Dataset	WHO	CDC	MayoClinic	Medline Plus	Cleaveland Clinic
0	1.0000	0.8705	0.82516	0.931391	0.706964	0.910332
1	0.9999	0.956159	0.987355	0.983396	0.957966	0.913138
2	1.0000	0.962162	0.983068	0.996502	0.950249	0.992
3	1.0000	0.998499	0.942359	0.927951	0.939746	0.948632
4	1.0000	0.999500	0.994975	0.994975	0.923805	0.989899
5	1.0000	0.931262	0.931966	0.839318	0.901713	0.873458
6	0.9997	0.94687	0.984351	0.896061	0.920086	0.31508
7	0.9996	0.923559	0.917431	0.911912	0.970874	0.704722
8	1.0000	0.865557	0.879552	0.942448	0.30339	0.998004
9	1.0000	0.932194	0.982706	0.882682	0.957791	0.809524

Table 8: Comparison of Classwise Performance Metrics

4.3.2 Comparison of Overall Performance Metrics

The metrics such as accuracy, macro precision, macro recall, and macro F1-score were compared across the five external datasets to capture a broader view of the model's performance. Upon comaprison, the near-perfect accuracy of 99.99% for the internal dataset serves as a baseline, however the external datasets provided a more realistic evaluation. The table below provides a direct comparison of the model's performance across diverse data sources.

Analysis of Overall Performance:

- **High Performing Datasets**: The CDC dataset results in highest accuracy of 94.30% and macro F1-score of 94.32%, closely followed by the WHO dataset of 93.87% accuracy and 93.88% F1-score. These datasets likely benefit from comprehensive symptom documentation and alignment with the training data's structure which has led the model to generalize better. The high macro precision of 94.91% for CDC, and 94.75% for WHO indicates a minimal false positives, whereas the macro recall provides a robust detection of true positives.
- **Moderate Performing Datasets**: The Mayo Clinic dataset achieves a decent accuracy of 93.02% and 93.07% F1-score, suggesting good generalization however it shows a slightly lower performance than CDC and WHO dataset.
- **Low Performing Datasets**: The MedlinePlus and Cleveland Clinic datasets indicate lowest performance, providing accuracies of 86.93% and 86.24%, respectively, and F1-scores of 85.33% and 84.55%. The significant drop in macro recall of 86.93% for MedlinePlus and 86.24% for Cleveland Clinic indicates higher false negatives, particularly for classes like Pneumonia and Brain Haemorrhage (MedlinePlus) and Migraine (Cleveland Clinic).

This comparative analysis of the overall performance metrics determines the model's strong generalization on high-quality datasets while identifying areas for improvement in handling less structured or noisy data.

Metrics	Internal Dataset	WHO	CDC	MayoClinic	Medline Plus	Cleaveland Clinic
Accuracy	1.0000	0.9387	0.943	0.9302	0.8693	0.8624
Macro Avg F1-Score	0.9999	0.938815	0.943228	0.930721	0.853258	0.845479
Macro Avg Precision	0.9999	0.947457	0.949134	0.939834	0.912038	0.908264
Macro Avg Recall	0.9999	0.9381	0.943	0.9302	0.8693	0.8624

Table 9: Comparison of Overall Performance Metrics

4.4 Discussion of Analysis

The thorough evaluation of the ANN model across multiple datasets provides a significant evidence of its potential as a robust tool for symptom-based disease classification system. The results detailed in Sections 4.1 and 4.2 and synthesized in Section 4.3, reflect the extensive efforts to design, train, and validate a model that well balances the accuracy, generalization ability, and practical utility.

Model Strengths: The near-perfect performance of the model on the internal SymbiPredict dataset showing 99.99% of accuracy determines its ability to learn complex symptom-disease mappings from a well-structured and augmented dataset. This success is attributed to the careful data curation, sophisticated augmentation techniques, and optimized ANN architecture, which collectively enables the model to capture complex patterns with less error. On external datasets, the model maintains high accuracy of up to 94.30% on CDC and robust F1-scores for classes like Malaria, Chickenpox, and Diabetes which indicates a strong generalization ability of the model for diseases with distinct symptom profiles. The consistency across CDC, WHO, and Mayo Clinic datasets proves the effectiveness of the training strategy, particularly the use of probabilistic encoding and noise injection to simulate real-world scenarios.

Challenges and Limitations: Despite its strengths, the model exhibits variability in performance, particularly on the MedlinePlus and Cleveland Clinic datasets, where accuracies drop to 86.93% and 86.24%, respectively. The significant misclassifications of Pneumonia having a recall of 0.179 (e.g., 798 instances as Allergy in MedlinePlus), Migraine (e.g., 813 instances as Brain Haemorrhage in Cleveland Clinic), and Brain Haemorrhage (e.g., 0.179

recall in MedlinePlus) highlight challenges with symptom overlap and data quality. Diseases like Pneumonia and Allergy show overlapping symptoms such as cough and fatigue which leads to confusion in the model's decision. Similarly, Migraine and Brain Haemorrhage, both involve headache-related symptoms which results in disease misclassification. Further, an observed reduction in performance MedlinePlus and Cleveland Clinic datasets highlight limited symptom representations, makiing it hard for the model to correctly interpret the classes. These results elucidate the significance of diverse training data and the need for more feature engineering to handle complex symptom profiles.

Clinical Applications: The model's strong performance on datasets marks it as a guaranteeing decision-support tool for early disease diagnosis in clinical settings. Its significant ability to accurately classify diseases can aid healthcare providers in allocating treatments to patients and providing remote diagnosis. However, the observed misclassifications, especially for critical conditions like Brain Haemorrhage, indicate that the model is not yet suitable as an independent diagnostic use. But this can complement clinical knowledge by flagging possible diagnoses for more detailed examination. The performance variability across datasets emphasizes the need for robust validation in target deployment environments to ensure reliability.

4.5 Summary

Reflecting the project's dedication to AI in healthcare, the evaluation results proves the potential of the ANN model as a valuable tool in symptom-based disease classification. The high accuracy on internal and external datasets shows the effectiveness of the chosen methodology whereas the challenges on certain datasets provide clear pathways for improvement. It paves a way forward for future researchers to build stronger and much more clinically viable model. These insights derived from the analysis not only validate the project's rigorous approach but also serves as encouragement to explore new avenues to enhance diagnostic accuracy and patient outcomes.

CHAPTER 5 MOBILE APPLICATION FLOW

5.1 Introduction

This chapter provides a detailed outline of the architecture, design, and working of the disease diagnosis application we developed as part of the project. The cross-platform mobile application, powered by artificial intelligence is meant to assist users in identifying potential conditions based on the symptoms they enter.

This section intends to provide a detailed end-to-end flow of the mobile application, from user interaction on the front end to the backend logic that processes the symptom inputs and generates possible disease conditions. Key areas that are discussed include the technologies used, user interface design, data flow between components, and the role of deep learning model in generating results.

By discussing both technical and user-centric perspective, this chapter highlights the intuitive collaboration and dedication that guided the app's development, ensuring it is both effective and user-friendly.

5.2 App Overview

The disease diagnosis application was developed with the primary goal of empowering individuals to take early steps toward understanding their health. By allowing users to input their symptoms and receive AI-driven diagnostic suggestions, the app serves as a supportive tool in the early identification of potential health issues. It is not meant to replace medical service but rather to help users in making better decisions about when to look for professional healthcare. Developed for the general public and patients alike, the app aims to reduce unnecessary visits to healthcare facilities for minor or non-urgent concerns.

At the same time, it encourages early consultation for symptoms that may indicate more serious conditions. The app has filled the gap of providing intelligent and preliminary assessments on demand in a time when healthcare systems can be overwhelmed and access to immediate medical consultation is not always possible.

The application development uses a modern tech stack to guarantee both reliability and accessibility. The front end is built using React Native and TypeScript, enabling responsive and cross platform user experience. On the backend, a Python-powered Flask API handles data processing and communication. The core of the diagnostic functionality is an AI model, which interprets symptom data and generates possible conditions based on learned patterns. Through thoughtful design and the integration of intelligent technologies, this application reflects a strong commitment to using digital tools for meaningful, real-world impact in the field of healthcare.

5.3 User Flow (Frontend Perspective)

The disease diagnosis application has been developed with a focus on user experience (UX) and clarity, ensuring that users can navigate easily through each stage of the app. The frontend architecture prioritizes clarity and functionality while combining modern design principles that enhance usability across a broad range of devices.

5.3.1 Onboarding/Login Screen

Upon launching the application, users are greeted with a minimal yet welcoming onboarding interface that also serves as the login gateway. The screen shows welcoming statements, contributing to a positive first impression, alongside a prominent Call to Action (CTA) labeled "Continue with Google".

By integrating Google OAuth to sign-in, the login process becomes seamless by eliminating the need for manual credential entry. This improving accessibility, especially for users who may find registration forms tedious or time-consuming. In addition, it adds a layer of security and ensures a frictionless user experience, which is crucial in health-related applications where quick access can be essential.

5.3.2 Symptom Input Interface (Home Screen)

Following successful authentication, users are directed to the Home Screen, which acts as the symptom input screen too. This screen includes a carefully designed dropdown-based input system, allowing users to select symptoms from a predetermined list.

The decision to use a dropdown over free-text input was intentional as it helps reduce input ambiguity, and minimizes user error. A primary call to action (CTA) labeled "Detect Disease" is placed to initiate the diagnosis process.

5.3.3 Diagnostic Process (User Perspective)

Upon tapping "Detect Disease", the application initiates a **backend API call**, transmitting the selected symptoms to the AI-based diagnostic engine. The transition to the results screen is immediate, without the use of a loading indicator or progress animation.

Despite the absence of transitional feedback, the system operates on an **asynchronous flow**, ensuring that the UI remains responsive while the backend performs symptom classification and disease prediction using the trained machine learning model. This direct transition helps keep the interaction seamless and efficient, minimizing unnecessary delays for the user.

5.3.4 Output Screen/Diagnostic Result

Upon completing the analysis, users are led to the results screen. Rather than simply listing the diagnosis, the response is put into a humanreadable form, offering both reassurance and guidance - encouraging the user to consult a healthcare professional where necessary. The predicted condition is displayed on the screen in the form of a simple, empathetic statement.

A secondary call to action (CTA) labeled “Go Back” makes route for the user to return to the previous screen. This going back option is not only essential for correcting potential deletion of symptoms but also balances iterative exploration. For example, users might want to explore multiple combinations of symptoms or edit their entries for more accurate results.

5.3.5 Optional Features and Navigation

The app includes a **bottom navigation bar** providing seamless access to key areas:

- **Home (Symptom Input)**: Returns the user to the primary interaction screen.
- **Profile**: Displays the authenticated user’s name and provides a “Logout” button for session management.

Though minimal at this stage, the **Profile screen** has been structured to accommodate future enhancements such as **diagnosis history tracking**, personalized insights, and more detailed health logs—allowing the application to evolve from a one-time-use tool to a more personalized health assistant.

The overall front-end flow has been built using **React Native with TypeScript**, ensuring cross-platform compatibility, strong type safety, and a fluid user experience across devices. Navigation, component rendering, and API communication have been optimized for performance and maintainability.

5.4 Technical Flow (Backend & AI Perspective)

5.4.1 Overall Architecture

Architecture flow:

User Input (React Native) → REST API (Flask Server) → ML Model (Symptom Vector Input) → JSON Response → UI Rendering (Result Screen)

At a high level, the AI Doctor app follows a **modular client-server architecture**, where the React Native front-end communicates with a Flask-based REST API server, which serves as the backend controller. The backend orchestrates requests to a pre-trained machine learning model responsible for disease prediction based on user symptoms.

Frontend: Built using React Native, it handles multi-select symptom inputs and constructs a binary vector representation (symptomVector[]) that acts as a feature input to the ML model.

API Layer: The app hits the backend via a POST request to the /api/predict endpoint.

Backend (Flask): Acts as a controller that accepts requests, sanitizes and validates input, routes it to the ML inference module, and returns the predictions.

Model Engine: A trained supervised ML classifier (e.g., Decision Tree or Naive Bayes) performs real-time inference based on encoded symptoms.

Result Dispatch: A prediction along with confidence (accuracy score) is sent back as a JSON response and used by the frontend to render human-readable diagnosis.

5.4.2 Symptom Processing

The app collects symptoms via a dynamic **multi-select dropdown UI**, implemented with react-native-element-dropdown, allowing users to select multiple symptoms. These are internally mapped to a **fixed-length binary feature vector of size 53**, where each bit corresponds to a known symptom.

Processing Flow:

- The handleSymptomChange() function updates the feature vector in real-time based on user selections.
- When the user taps on **"Detect Disease"**, the predictSymptom() function serializes the vector into a JSON body and sends a POST request to the Flask endpoint /api/predict.
- The payload follows the format:

{ "symptomVector": [0, 1, 0, 0, 1, ..., 0] }

This vectorized form aligns with the training data schema expected by the model and ensures zero preprocessing latency at the backend.

5.4.3 Disease Prediction Model

The backend leverages a **supervised classification model** trained on a curated **symptom-disease mapping dataset**. The model accepts a binary input vector representing the presence or absence of specific symptoms.

Model Details:

- **Type**: Custom-built Artificial Neural Network (ANN)
  Architecture: 1 input layer → 3 hidden layers (16 neurons each, ReLU + 0.2 dropout) → 1 output layer (10 classes, softmax)
  Input: 53 binary-encoded symptoms

Framework: Keras Functional API (TensorFlow backend)
Deployment: Designed for efficient inference in mobile/web settings

- **Output**:
    - predicted_class: An integer representing the index of the predicted disease
    - accuracy: Model confidence score (float ∈ [0,1])

Internally, the model performs class probability inference and returns the top-k class with the highest confidence, though currently only the top-1 prediction is used.

5.4.4 Result Interpretation

On receiving the backend response, the frontend evaluates the confidence score (accuracy):

- If accuracy ≥ 0.7: The predicted class description is fetched and shown to the user.
- Else: A **fallback disease description (class 10)** is shown, acting as a safe default.

This conditional rendering enhances the UX by:

- Avoiding overconfident false positives
- Offering fallback diagnosis when certainty is low

The description mapping is handled through the classDescriptions object, which maps model output indexes to detailed disease descriptions.

5.4.5 Backend Technologies & Services

Component	Tech Stack & Description
Server	**Flask (Python)** – REST API created using Flask handles POST requests, performs input validation, and dispatches data to the AI engine
Model Hosting	**Local** (in-process inference) – For now, the trained model is serialized (.pkl) and loaded into memory at server runtime

Table 10: App Backend Technologies & Services

5.6 Error Handling & Edge Cases

Given the deterministic nature of the application workflow and the controlled structure of both input and output formats, the system encounters minimal runtime exceptions or unanticipated states. However, a key edge case has been explicitly handled at the prediction interpretation stage to ensure robustness and user trust.

Upon receiving the prediction output from the backend, the frontend logic evaluates the model's confidence score (probabilistic accuracy). This score corresponds to the maximum softmax probability associated with the predicted disease class. The system responds as follows:

- **If confidence ≥ 0.7**: The model is deemed sufficiently certain, and the corresponding disease class description is retrieved and displayed to the user.
- **If confidence < 0.7**: The system defaults to a fallback diagnosis (predefined as class 10), which represents a general, non-critical health condition. This serves as a conservative recommendation and mitigates the risk of providing inaccurate or misleading diagnostic suggestions.

This conditional logic acts as a safeguard against overconfident misclassifications and reflects a precautionary principle suitable for healthcare-related applications. It ensures that users are either presented with a reliable prediction or redirected to a neutral, low-risk outcome when model certainty is inadequate.

5.7 Challenges Faced

The development of the AI-driven symptom-based disease classifier presented several interdisciplinary challenges, particularly in the integration of the machine learning model within a full-stack application and in navigating user interface decisions that could influence diagnostic interpretation.

5.7.1 Model Integration with Frontend & Backend Systems

A primary technical challenge involved the seamless integration of the trained Artificial Neural Network (ANN) model into the backend infrastructure and ensuring consistent communication with the frontend interface.

- **Serialization and model loading**: Ensuring the trained model, saved in .h5 format, could be reliably loaded and served without compatibility issues across different environments.
- **User Response Formatting**: Maintaining alignment between the frontend-generated input vectors and the model's expected input structure, particularly in encoding the binary symptom vectors.

- **Backend Response Formatting**: Structuring the backend response to include both the predicted class and associated confidence scores in a format that could be readily parsed and interpreted by the frontend.

5.7.2 UI/UX Design Considerations & Diagnostic Reliability

Another significant challenge lay in the translation of model predictions into a user-perspective format that validate the clinical caution. UI/UX decisions were not only design considerations but had direct implications for how users may conceptualize and act upon the system's outputs. A few key considerations included:

- Visual confidence representation: Presenting model predictions in a clear way that communicates the level of certainty, thereby avoiding overreliance on less confident results.

- Designing the Fallback Logic: Introducing a safety net for low-confidence predictions by displaying a neutral fallback diagnosis. For instance, when confidence < 0.7.

- Symptom Input Clarity: Developing a frontend interface that provides a reliable and accurate symptom selection was essential to have valid model inputs. Any poor UX decisions here could have reduced the model performance, not because of model's limitations, but due to incorrect user inputs.

Altogether, these challenges highlight the importance of an integrated, systems-level understanding, where machine learning, backend engineering, and user-centered design are treated as interdependent components of the solution. Addressing all these issues effectively was important in ensuring both technical performance and user trust in the system.

5.8 Future Enhancements

While the current system provides a robust foundation for symptom-based disease classification, several areas exist for future enhancement that can significantly elevate user experience, engagement, and clinical utility.

5.8.1 Integration of Real Time AI Doctor Chat

A promising future direction involves the establishment of a real-time chat interface, enabling users to have context aware conversations with an AI-powered medical assistant. This feature would allow for more interactive symptom understanding by asking follow-up questions, and clarification of user prompts, thus mimicing the natural flow of a clinical assistance. Leveraging advancements in large language models (LLMs), such a system could refine preliminary diagnoses, provide health education, and improve user trust through a more personalized and conversational experience. Furthermore, a real-time chat interface opens the

door to incremental symptom reporting, which could enhance diagnostic accuracy in cases where users may initially omit relevant symptoms.

5.8.2 Secure Storage of User Chat History

The other important improvement lie in the secure and structured storage of user diagnosis histories. By maintaining records of user chats and symptom submissions, the system can offer several benefits including personalized healthcare insights. It is equally valuable, in enabling users to review and share their symptom history with healthcare professionals. Stored data, when anonymized - could serve as a valuable resource for future model retraining and research.

Together, these enhancements aim to convert a current diagnostic tool into an extensive intelligent healthcare assistant focusing on user engagement, individualization, and continuity of care.

CHAPTER 06 CONCLUSION

The thesis demonstrated both the design and implementation of the AI-system for symptoms-based disease classification using an ANN. This system was developed to provide more accessible, preliminary health assessments for communities with inadequate medical infrastructure The project demonstrated adequate classification performance on ten common diseases by training the model on a carefully designed and augmented data set and extensively testing on various datasets from reputable global medical organizations.

The ANN model has shown high accuracy for internal and external datasets, thereby confirming their strength and generalization ability. An insight gained from the evaluation shows that the model seems to effectively predict the diseases with overlapping symptoms. Though, misclassifications did happen, but it was demonstrated that the model could strongly predict diseases with overlapping signs and symptoms. These findings illustrate both the robustness and limitations of the model that can be improved upon in the future.

In addition, the integration of the ANN model into a user-friendly mobile application bridges the gap between AI research and real-world usability. This allows individuals to input symptoms and receive likely disease predictions, functioning as a reliable, early decision support tool. Even though, it is not designed to replace medical professionals; rather it is intended to assist users in making informed decisions about seeking further medical care.

In general, this project makes a meaningful contribution to the expanding field of AI in health care. It demonstrates how deep learning can be utilized to enhance early detection of diseases, enhance telemedicine approaches and help support digital health access. Future work could include an expanded set of diseases, multi-modal data or utilize explainable AI, to develop trust and transparency in clinical practice.

Appendix A Model Development and Evaluation

This appendix outlines the key stages of building and evaluating an Artificial Neural Network (ANN) model for symptom-to-disease classification. The process covered includes data preprocessing, defining the model architecture, training the model, testing on an initial test set, and subsequently testing on newly collected data.

Data Preprocessing

```python
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
import joblib

# 1. Load the data
data = pd.read_csv(direc + "Dataset/FinalCN_Dataset.csv")

# 2. Separate features and labels.
#    Assume the first 53 columns are features and the last column is the class name.
X = data.iloc[:, :-1].values  # features (53 columns)
y = data.iloc[:, -1].values   # class labels

# 3. Encode the class labels (direct classification with existing 10 classes)
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

# Print the mapping from class names to numeric codes
class_mapping = {cls: idx for idx, cls in enumerate(label_encoder.classes_)}
print("Class mapping (class name -> numeric code):")
print(class_mapping)

# 4. Split the data: 80% training and 20% test.
#    Use stratify to keep class distribution similar.
X_train_full, X_test, y_train_full, y_test = train_test_split(
    X, y_encoded, test_size=0.50, stratify=y_encoded, shuffle=True, random_state=42
)
```

```
# 5. Further split the training data into training and validation sets (70% train, 30% validation of
X_train, X_val, y_train, y_val = train_test_split(
    X_train_full, y_train_full, test_size=0.50, stratify=y_train_full, shuffle=True, random_state=42
)

# 6. Standardize the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled   = scaler.transform(X_val)
X_test_scaled  = scaler.transform(X_test)

# 7. Save the scaler to a file for future use (using joblib)
joblib.dump(scaler, direc + "Preprocessing/scaler.pkl")

# 8. Save the preprocessed data as .npy files
np.save(direc + "Preprocessing/X_train_scaled.npy", X_train_scaled)
np.save(direc + "Preprocessing/y_train.npy", y_train)
np.save(direc + "Preprocessing/X_val_scaled.npy", X_val_scaled)
np.save(direc + "Preprocessing/y_val.npy", y_val)
np.save(direc + "Preprocessing/X_test_scaled.npy", X_test_scaled)
np.save(direc + "Preprocessing/y_test.npy", y_test)

print("Preprocessing complete. Scaler and preprocessed data have been saved.")
```

Figure 8 Data Preprocessing

The initial dataset is a symptom-to-disease dataset [SymbiPredict and Expanded with data duplication with noise technique]. The data preprocessing steps prepare this raw data for use in the ANN model.

- **Loading the Data:** The dataset is loaded from a CSV file named "FinalICN_Dataset.csv" using pandas
- **Separating Features and Labels:** The features (symptoms) are extracted from the first 53 columns, and the class labels (diseases) are extracted from the last column
- **Encoding Class Labels:** The categorical class labels (disease names) are encoded into numerical representations using LabelEncoderThe dataset is known to have 10 distinct classes. The mapping between class names and their numeric codes is printed. This mapping is explicitly shown in source, where disease names like 'Allergic Reaction' map to 0, 'Arthritis' to 1, and so on, up to 'Typhoid' mapping to 9.

- **Splitting the Data:** The data is initially split into a training set (80%) and a test set (20%) using train_test_split. The stratify parameter is used to ensure the distribution of classes is similar in both splits
- **Further Splitting Training Data:** The training data (X_train_full, y_train_full) is further split into a training set (70%) and a validation set (30%). This means the original 80% is split, resulting in a final training set size that is 70% of the original 80%, and a validation set that is 30% of the original 80%. Stratification is again used.
- **Standardizing the Data:** Features (X) are standardized using StandardScaler. The scaler is fitted on the training data (X_train_scaled) and then used to transform the training, validation, and test sets.
- **Saving Preprocessed Data and Scaler:** The trained StandardScaler and the scaled training, validation, and test sets (X_train_scaled, y_train, X_val_scaled, y_val, X_test_scaled, y_test_encoded) are saved to disk using joblib and numpy.save respectively. This allows for later reuse without re-running the preprocessing steps.

The script confirms that preprocessing is complete and data has been saved

Model Architecture

The model is built as an Artificial Neural Network (ANN) using the TensorFlow Keras functional API.

```
import tensorflow as tf
from tensorflow.keras.layers import Input, Dense, Dropout
from tensorflow.keras.models import Model
from tensorflow.keras.regularizers import l2

# Define the input layer: shape=(53,) because there are 53 bin
input_layer = Input(shape=(53,), name="input_layer")

# First hidden layer:
x = Dense(16, activation='relu', name="dense_1")(input_layer)
x = Dropout(0.2, name="dropout_1")(x)

# Second hidden layer:
x = Dense(16, activation='relu', name="dense_2")(x)
x = Dropout(0.2, name="dropout_2")(x)

# Third hidden layer:
x = Dense(16, activation='relu', name="dense_3")(x)
x = Dropout(0.2, name="dropout_3")(x)

```

```
21  # Output layer: 10 neurons (for the 10 classes) with softmax activatior
22  output_layer = Dense(10, activation='softmax', name="output_layer")(x)
23
24  # Build the model using the Functional API.
25  model = Model(inputs=input_layer, outputs=output_layer)
26
27  # Compile the model:
28  # - Optimizer: Adam (a robust choice for many tasks)
29  # - Loss: sparse_categorical_crossentropy (since labels are integer enc
30  # - Metrics: accuracy
31  model.compile(optimizer=tf.keras.optimizers.Adam(),
32                loss='sparse_categorical_crossentropy',
33                metrics=['accuracy'])
34
35  # Display the model summary to review the architecture.
36  model.summary()
```

Figure 9: Model Architecture Code

- **Input Layer:** The input layer expects inputs with a shape of (53,) because there are 53 binary feature inputs.
- **Hidden Layers:** The model has three hidden layers with ReLU activation. Dropout layers are included after each hidden layer with a rate of 0.2 to help prevent overfitting.
    1. First hidden layer: 16 neurons.
    2. Second hidden layer: 16 neurons.
    3. Third hidden layer: 16 neurons.
- **Output Layer:** The output layer has 10 neurons, corresponding to the 10 classes. It uses a softmax activation function, suitable for multi-class classification to produce probability distributions over the classes.
- **Model Compilation:** The model is compiled using the Adam optimizer, which is described as a robust choice for many tasks. The loss function is

sparse_categorical_crossentropy, appropriate because the labels are integer encoded. The metric used for evaluation is accuracy.

- A model summary is displayed to review the architecture.

Model Training

```python
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, CSVLogger



# Train the model
history = model.fit(
    X_train_scaled,
    y_train,
    epochs=10,
    batch_size=64,
    validation_data=(X_val_scaled, y_val)

)

# Save the final (normal) model after training
model.save(direc + 'Training/final_model.keras')

# Plot the training history: Accuracy and Loss
plt.figure(figsize=(12, 6))

# Accuracy plot
plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

# Loss plot
plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Train Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.tight_layout()
plt.savefig(direc + 'Training/training_plots.png')
plt.show()
```

Figure 10: Model Training Code

The model is trained using the prepared training and validation data.

- **Training Execution:** The model.fit method is used to train the model. The training runs for 10 epochs with a batch size of 64. Validation data (X_val_scaled, y_val) is provided to monitor performance during training.
- **Saving the Trained Model:** After training, the final model is saved in the Keras format.
- **Plotting Training History:** The training history (accuracy and loss) is plotted for both the training and validation sets over the epochs. The plots show the model's performance curves. Specifically, the accuracy plot shows both 'Train Accuracy' and 'Validation Accuracy', while the loss plot shows 'Train Loss' and 'Validation Loss'. These plots are saved as a PNG file.

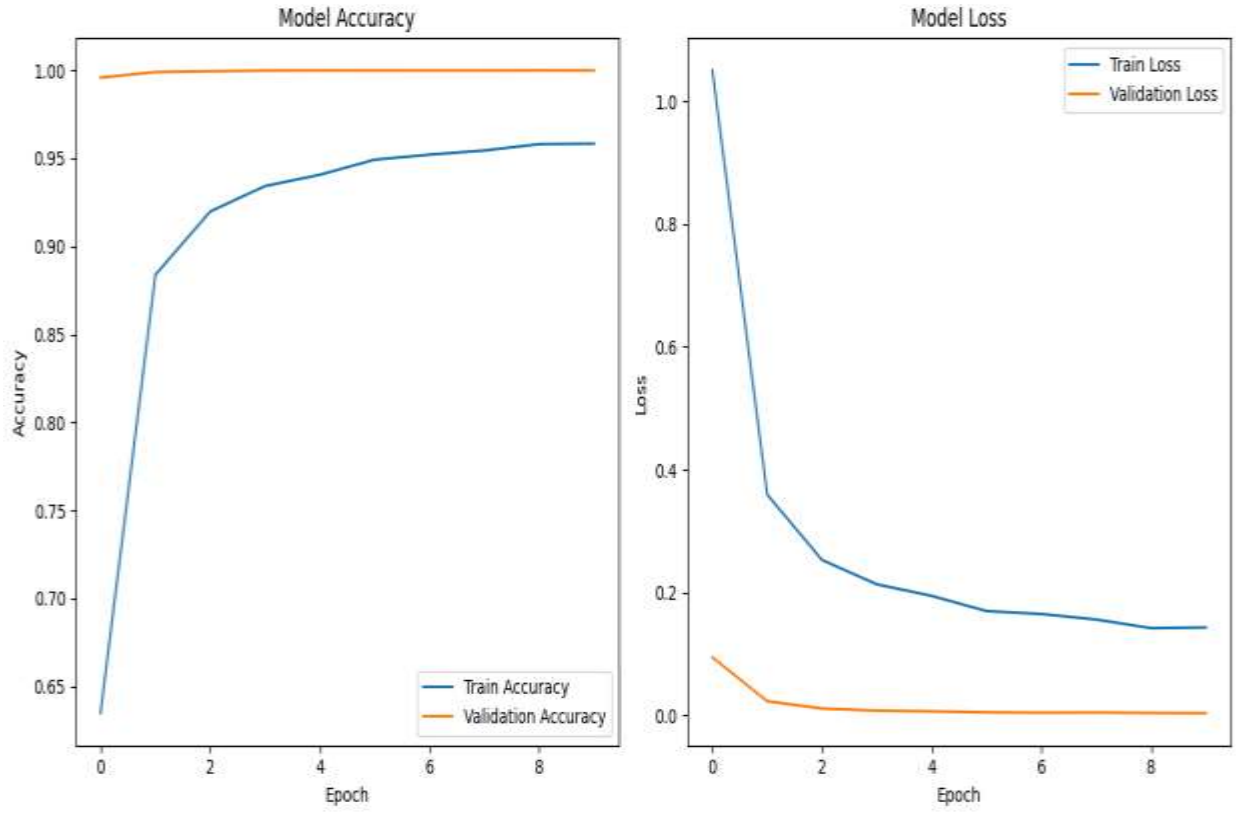

Figure 11: Training & Validation Plot

Model Testing (Original Test Set)

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score

# Predict on the test set using the trained model.
predictions = model.predict(X_test_scaled)
predicted_labels = np.argmax(predictions, axis=1)

# Compute test accuracy
test_accuracy = accuracy_score(y_test, predicted_labels)
print(f"Test Accuracy: {test_accuracy:.4f}")  # Print test accuracy

# Generate the classification report and convert it to a DataFrame.
report = classification_report(y_test, predicted_labels, output_dict=True)
report_df = pd.DataFrame(report).transpose()

# Save the classification report as a CSV file.
report_csv_path = direc + 'Testing/classification_report.csv'
report_df.to_csv(report_csv_path, index=True)
print(f"Classification report saved as: {report_csv_path}")

# Compute the confusion matrix.
cm = confusion_matrix(y_test, predicted_labels)

# Plot the confusion matrix using seaborn.
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')

# Save the confusion matrix plot as a PNG file.
cm_png_path = direc + 'Testing/confusion_matrix.png'
plt.savefig(cm_png_path)
plt.show()
print(f"Confusion matrix plot saved as: {cm_png_path}")
```

Figure 12: Model Testing Code (Original Test Set)

After training, the model is evaluated on the original 20% test set that was initially held out.

- **Making Predictions:** Predictions are made on the standardized test features (X_test_scaled). The argmax function is used to get the predicted class label (index) for each sample.
- **Computing Test Accuracy:** The overall accuracy on the test set is computed using accuracy_score and printed.
- **Generating Classification Report:** A classification report is generated using classification_report, which provides precision, recall, F1-score, and support for each class. This report is converted to a pandas DataFrame and saved as a CSV file.
- **Computing and Plotting Confusion Matrix:** A confusion matrix is computed using confusion_matrix to show the counts of true positive, true negative, false positive, and false negative predictions for each class. This matrix is then plotted as a heatmap using seaborn for visualization. The plot is saved as a PNG file.

Testing on New Data:

```python
import numpy as np
import pandas as pd
import joblib
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler

# Path to the directory
direc = r"/content/drive/MyDrive/FYP/ANN3/"

# 1. Load the new CSV test data
new_data = pd.read_csv(direc + "New Test/Test_Dataset.csv")

# 2. Separate features (first 53 columns) and labels (last column)
X_new = new_data.iloc[:, :-1].values  # Features (53 columns)
y_new = new_data.iloc[:, -1].values   # Class labels

# 3. Load the previously saved scaler
scaler = joblib.load(direc + 'Preprocessing/scaler.pkl')

# 4. Standardize the new data using the loaded scaler
X_new_scaled = scaler.transform(X_new)

# 5. Encode the class labels using LabelEncoder (since you only have scaler.p
label_encoder = LabelEncoder()
y_new_encoded = label_encoder.fit_transform(y_new)

# 6. Print the class labels and their corresponding numeric encoding
class_mapping = {cls: idx for idx, cls in enumerate(label_encoder.classes_)}
print("Class mapping (class name -> numeric code):")
print(class_mapping)

# Return the preprocessed data and encoded labels
X_new_scaled, y_new_encoded
```

Figure 13: New Data Preprocessing

After the initial training and testing phase, the model is tested on new data, described as being collected from 5 different sources [WHO, Mayo Clinic etc].

- **Loading New Data:** The new test data is loaded from "New Test/Test_Dataset.csv".

- **Separating Features and Labels:** Similar to the original data, features and labels are separated.
- **Preprocessing New Data:** The previously saved scaler is loaded using joblib. The new features (X_new) are standardized using this loaded scaler. The new class labels (y_new) are encoded using LabelEncoder. Note that source includes code to print the class mapping again, potentially to ensure consistency or verify the encoding for the new data. While the sources mention new data collection and testing, they show the process starting with a CSV file ("New Test/Test_Dataset.csv") that presumably already contains the data with symptoms mapped to the dataset's structure and the corresponding class labels. The sources do not detail the process of collecting the data from 5 sources or the method used to map the collected symptoms onto the specific 53 features of the dataset.

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
from tensorflow.keras.models import load_model

# Path to the directory
direc = r"/content/drive/MyDrive/FYP/ANN3/"

# Load the trained model
model = load_model(direc + 'Training/final_model.keras')

# Assuming that you have preprocessed X_new_scaled and y_new_encoded from the previous step
# Make predictions using the trained model
predictions = model.predict(X_new_scaled)
predicted_labels = np.argmax(predictions, axis=1)

# Compute accuracy score on the new data
test_accuracy = accuracy_score(y_new_encoded, predicted_labels)
print(f"Test Accuracy: {test_accuracy:.4f}")  # Print test accuracy

# Generate classification report (CF) and convert it to a DataFrame
report = classification_report(y_new_encoded, predicted_labels, output_dict=True)
report_df = pd.DataFrame(report).transpose()

# Save the classification report as a CSV file
report_csv_path = direc + "New Test/classification_report.csv"
report_df.to_csv(report_csv_path, index=True)
print(f"Classification report saved as: {report_csv_path}")
```

```
# Compute the confusion matrix (CM)
cm = confusion_matrix(y_new_encoded, predicted_labels)

# Plot the confusion matrix using seaborn
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', xticklabels=label_encoder.classes_, yticklabels=label_encoder.classes_)
plt.title('Confusion Matrix')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')

# Save the confusion matrix plot as a PNG file
cm_png_path = direc + "New Test/confusion_matrix.png"
plt.savefig(cm_png_path)
plt.show()
print(f"Confusion matrix plot saved as: {cm_png_path}")
```

Figure 14: Model Testing Code (New Set)

- **Loading Trained Model:** The previously trained model ("final_model.keras") is loaded.
- **Making Predictions on New Data:** Predictions are made on the preprocessed new features (X_new_scaled) using the loaded model. The predicted labels are obtained using np.argmax.
- **Computing and Reporting Performance on New Data:** The model's performance on the new data is evaluated by computing test accuracy, generating a classification report, and computing/plotting a confusion matrix, following the same procedures as testing on the original test set. The results (report CSV and confusion matrix plot PNG) are saved to a "New Test" directory.

Disease Mapping:

```
'Allergic Reaction'= 0
'Arthritis'= 1
'Chickenpox'= 2
'Dengue'= 3
'Diabetes '= 4
'Malaria'= 5
'Migraine'= 6
'Paralysis (brain hemorrhage)'= 7
'Pneumonia'= 8
'Typhoid'= 9
```

Figure 15: Disease Mapping

The model predicts integer codes representing the disease classes. The mapping between these numeric codes and the actual disease names is provided.

- **Mapping:** This mapping shows the 10 distinct classes that the model is trained to classify.

Appendix B: App Structure Overview

This appendix outlines the key components and their roles in the App structure, based on the provided code excerpts. The App facilitates user authentication via Google Sign-In, allows users to select symptoms for disease prediction, displays prediction results, and provides a profile view with a logout option.

RootLayout:

```tsx
import { SafeAreaView } from "react-native";
import React from "react";
import { SafeAreaProvider } from "react-native-safe-area-context";
import { ClerkProvider } from "@clerk/clerk-expo";
import { tokenCache } from "@/cache";
import InitialLayout from "@/components/InitialLayout";

const publishableKey = process.env.EXPO_PUBLIC_CLERK_PUBLISHABLE_KEY!;

if (!publishableKey) {
  throw new Error(
    "Missing Publishable Key. Please set EXPO_PUBLIC_CLERK_PUBLISHABLE_KEY in your .env"
  );
}

export default function RootLayout() {
  return (
    <ClerkProvider tokenCache={tokenCache} publishableKey={publishableKey}>
      <SafeAreaProvider>
        <SafeAreaView style={{ flex: 1 }}>
          <InitialLayout />
        </SafeAreaView>
      </SafeAreaProvider>
    </ClerkProvider>
  );
}
```

Figure 16: Root Layout Code

This component is likely the top-level layout that wraps the entire application.

- It imports necessary modules like ClerkProvider from @clerk/clerk-expo and SafeAreaProvider/SafeAreaView from react-native-safe-area-context.

- It uses a publishable key, EXPO_PUBLIC_CLERK_PUBLISHABLE_KEY, which is checked for existence, throwing an error if missing.
- The ClerkProvider component is used, possibly for managing user authentication state across the application. It takes tokenCache and publishableKey props.
- The SafeAreaProvider and SafeAreaView are used, likely to ensure content is displayed within the safe area boundaries of the device screen.
- Inside the SafeAreaView, it renders the InitialLayout component.

InitialLayout:

```
import { useAuth } from "@clerk/clerk-expo";
import { Stack, useRouter, useSegments } from "expo-router";
import { useEffect } from "react";

export default function initialLayout() {
  const { isLoaded, isSignedIn } = useAuth();

  const segments = useSegments();
  const router = useRouter();

  useEffect(() => {
    if (!isLoaded) return;

    const inAuthScreen = segments[0] === "(auth)";

    if (!isSignedIn && !inAuthScreen) {
      router.replace("/(auth)/login");
    } else if (isSignedIn && inAuthScreen) {
      router.replace("/(tabs)");
    }
  }, [isLoaded, isSignedIn, segments]);

  return <Stack screenOptions={{ headerShown: false }} />;
}
```

Figure 17: Initial Layout Code

This component appears to handle initial navigation based on the user's authentication status.

- It imports useAuth from @clerk/clerk-expo to access authentication state (isLoaded, isSignedIn).
- It uses hooks from expo-router (Stack, useRouter, useSegments, useEffects).
- A useEffect hook is used to perform navigation logic when isLoaded, isSignedIn, or segments change.
- If the authentication state is not yet loaded (!isLoaded), the effect returns early.

- It determines if the user is currently on an authentication screen (inAuthScreen) by checking the first segment of the URL (segments === "(auth)").
- If the user is not signed in (!isSignedIn) and is not already on an authentication screen (!inAuthScreen), it replaces the current route with /(auth)/login.
- If the user is signed in (isSignedIn) and is on an authentication screen (inAuthScreen), it replaces the current route with /(tabs)`.
- The component returns a Stack component from expo-router, with headerShown: false in the screen options.

Index (Screen):

```
1  import React from "react";
2  import { Redirect } from "expo-router";
3
4  export default function Index() {
5    return <Redirect href="/(auth)/login" />;
6  }
7
```

Figure 18: Index (Screen) Code

- This component seems to be a simple entry point or default route.
- It imports React and Redirect from expo-router.
- The component's primary function is to render a <Redirect> component that redirects the user to /(auth)/login.

Login (Screen):

```
import {
  View,
  Text,
  Image,
  StyleSheet,
  Dimensions,
  TouchableOpacity,
} from "react-native";
import React from "react";
import app from "./../../assets/images/app.png";
import { useSSO } from "@clerk/clerk-expo";
import { useRouter } from "expo-router";

export default function Login() {
  const { startSSOFlow } = useSSO();
  const router = useRouter();

  const handleGoogleSignIn = async () => {
    try {
      const { createdSessionId, setActive } = await startSSOFlow({
        strategy: "oauth_google",
      });

      if (setActive && createdSessionId) {
        setActive({ session: createdSessionId });
        router.push("/(tabs)");
      }
    } catch (error) {
      console.error("OAuth error", error);
      alert("Failed to sign in. Please try again.");
    }
  };
```

```
34    return (
35      <View style={{ alignItems: "center" }}>
36        <Image source={app} style={styles.appImage} />
37        <View
38          style={{
39            backgroundColor: "#fff",
40            padding: 7,
41            alignItems: "center",
42            marginTop: -10,
43            elevation: 10,
44          }}
45        >
46          <Text style={styles.heading}>Mini Doctor in Your Phone</Text>
47          <Text style={styles.heading}>Disease Detection in Seconds</Text>
48          <Text style={{ textAlign: "center", marginTop: 10 }}>
49            Enter the symptoms you have and find your disease in seconds with AI.
50            Log in to get started.
51          </Text>
52          <TouchableOpacity
53            onPress={handleGoogleSignIn}
54            style={{
55              padding: 16,
56              backgroundColor: "#0000FF",
57              borderRadius: 90,
58              alignItems: "center",
59              marginTop: 20,
60              width: Dimensions.get("screen").width * 0.8,
61            }}
62          >
63            <Text style={{ fontSize: 17, color: "#fff" }}>Login With Google</Text>
64          </TouchableOpacity>
65          <Text style={{ textAlign: "center", marginTop: 10 }}>
66            By Continuing you agree to our privacy policy.
67          </Text>
68        </View>
69      </View>
70    );
71  }
```

```
71  }
72
73  const styles = StyleSheet.create({
74    appImage: {
75      width: 400,
76      height: 500,
77      resizeMode: "cover",
78      marginTop: 2,
79    },
80    heading: {
81      fontSize: 20,
82      fontWeight: "bold",
83    },
84  });
85
```

Figure 19: Login Screen Code

- This component renders the login screen UI and handles the sign-in process.
- It imports various components from react-native like View, Text, Image, etc., and hooks from react and expo-router (useState, useRouter).
- It imports useSignIn and useSession from @clerk/clerk-expo and startSSOFlow from @clerk/clerk-expo/dist/single-sign-on.
- It uses useState for an error message (error).
- It uses useSignIn to get the startSSOFlow function for authentication.
- It uses useSession to get the setActive function, possibly to set the active session upon successful sign-in.
- It uses useRouter for navigation.
- An async function handleGoogleSignIn is defined to handle the sign-in process. It calls startSSOFlow with the strategy 'oauth_google'. Upon success, it sets the active session and navigates to the /(tabs)route. It includes atry...catch` block to handle potential errors during sign-in, displaying an alert and setting an error message.
- The UI includes an image (AppImage), text elements ("Mini Doctor in Your Phone", "Disease Detection in Seconds with AI", "Enter the symptoms you have and find your disease in seconds with AI", "Log in to get started").

- It features a TouchableOpacity button styled for Google login, with text "Login with Google". This button's onPress handler is handleGoogleSignIn.
- There is also text stating "By Continuing you agree to our privacy policy.".
- Stylesheet definitions (appImage, heading, subHeading, button, buttonText, privacyText) are provided for styling the UI.

Logout (Function/Component):

```
1  import {
2    View,
3    Text,
4    TouchableOpacity,
5    StyleSheet,
6    Dimensions,
7  } from "react-native";
8  import React from "react";
9  import { useAuth } from "@clerk/clerk-expo";
10 import { useRouter } from "expo-router";
11
12 export default function Logout() {
13   const { signOut } = useAuth(); // Access the signOut function from
14   const router = useRouter();
15
16   const handleLogout = async () => {
17     try {
18       await signOut(); // Sign the user out
19       router.replace("/(auth)/login"); // Redirect to the login page
20     } catch (error) {
21       console.error("Logout error", error);
22       alert("Failed to log out. Please try again.");
23     }
24   };
25
```

```
26    return (
27      <View style={styles.container}>
28        <TouchableOpacity onPress={handleLogout} style={styles.button}>
29          <Text style={styles.buttonText}>Logout</Text>
30        </TouchableOpacity>
31      </View>
32    );
33  }
34
35  const styles = StyleSheet.create({
36    container: {
37      flex: 1,
38      justifyContent: "center", // Center vertically
39      alignItems: "center", // Center horizontally
40    },
41    button: {
42      padding: 16,
43      backgroundColor: "#FF0000",
44      borderRadius: 90,
45      alignItems: "center",
46      marginTop: 250,
47      width: Dimensions.get("screen").width * 0.4,
48    },
49    buttonText: {
50      fontSize: 17,
51      color: "#fff",
52      textAlign: "center",
53    },
54  });
```

Figure 20: Logout Component Cod

- This component likely provides a way for the user to log out of the application.
- It imports useAuth from @clerk/clerk-expo to access the signOut function and useRouter from expo-router for navigation.
- An async function handleLogout is defined. It calls the signOut function to log the user out. Upon success, it replaces the current route with /(auth)/login to redirect to the login page. It includes a try...catch block to handle errors during logout, displaying an alert.
- The UI is simple, containing a View and a TouchableOpacity with the text "Logout". The onPress handler for the button is handleLogout.
- Stylesheet definitions (container, button, buttonText) are provided.

TabLayout:

```
import { View, Text } from "react-native";
import React from "react";
import { Tabs } from "expo-router";
import { Ionicons } from "@expo/vector-icons";

export default function TabLayout() {
  return (
    <Tabs screenOptions={{ tabBarShowLabel: false, headerShown: false }}>
      <Tabs.Screen
        name="index"
        options={{
          tabBarIcon: () => <Ionicons name="home" size={20} color="blue" />,
        }}
      />
      <Tabs.Screen
        name="profile"
        options={{
          tabBarIcon: () => (
            <Ionicons name="person-circle" size={20} color="blue" />
          ),
        }}
      />
    </Tabs>
  );
}
```

Figure 21: Tab Layout Code

- This component sets up the tab-based navigation structure for the application.
- It imports Tabs from expo-router.
- It returns a <Tabs> component with screen options (tabBarShowLabel: false, headerShown: false).
- It defines two tabs:
  1. A tab named "index", likely the home screen. It has options to display an Ionicons with the name "home" as the tabBarIcon.
  2. A tab named "profile", likely the user profile screen. It has options to display an Ionicons with the name "person-circle" as the tabBarIcon.

Index (Home Screen):

```
import React, { useState } from "react";
import {
  View,
  Text,
  StyleSheet,
  ImageBackground,
  TouchableOpacity,
  Dimensions,
} from "react-native";
import { MultiSelect } from "react-native-element-dropdown";
import App from "../../assets/images/home.png";
import { useRouter } from "expo-router"; // Correct import
import BackendAccess from "@/components/BackendAccess";

const symptoms = [
  "itching",
  "skin_rash",
  "continuous_sneezing",
  "shivering",
  "chills",
  "joint_pain",
  "acidity",
  "vomiting",
  "fatigue",
  "weight_loss",
  "restlessness",
  "lethargy",
  "irregular_sugar_level",
  "cough",
  "high_fever",
  "breathlessness",
  "sweating",
  "indigestion",
  "headache",
  "nausea",
```

```
    "loss_of_appetite",
    "pain_behind_the_eyes",
    "back_pain",
    "constipation",
    "abdominal_pain",
    "diarrhoea",
    "mild_fever",
    "swelled_lymph_nodes",
    "malaise",
    "blurred_and_distorted_vision",
    "phlegm",
    "chest_pain",
    "fast_heart_rate",
    "obesity",
    "excessive_hunger",
    "muscle_weakness",
    "stiff_neck",
    "swelling_joints",
    "movement_stiffness",
    "weakness_of_one_body_side",
    "toxic_look_(typhos)",
    "depression",
    "irritability",
    "muscle_pain",
    "altered_sensorium",
    "red_spots_over_body",
    "belly_pain",
    "watering_from_eyes",
    "increased_appetite",
    "polyuria",
    "rusty_sputum",
    "visual_disturbances",
    "painful_walking",
];

```

```
71  export default function Index() {
72    const [selectedSymptoms, setSelectedSymptoms] = useState([]);
73    const [symptomVector, setSymptomVector] = useState(
74      Array(53).fill(0) // Initialize as array of 53 zeros: [0,0,...,0]
75    );
76    const router = useRouter();
77
78    const handleSymptomChange = (selectedItems) => {
79      // Create a new vector as an array
80      const newVector = Array(53).fill(0); // Start with all zeros
81      symptoms.forEach((symptom, index) => {
82        newVector[index] = selectedItems.includes(symptom) ? 1 : 0; // Set 1 if selected
83      });
84      setSymptomVector(newVector); // Store as array [0,1,0,...]
85      setSelectedSymptoms(selectedItems);
86    };
87
88    const predictSymptom = async () => {
89      try {
90        console.log("Sending Symptom Vector:", symptomVector); // Debugging log
91
92        const response = await fetch("http://192.168.110.120:5000/api/predict", {
93          // Use your computer's IP address
94          method: "POST",
95          headers: { "Content-Type": "application/json" },
96          body: JSON.stringify({ symptomVector }), // Send as {"symptomVector": [0,1,0,...]}
97        });
98
99        if (!response.ok) {
100         throw new Error("Failed to fetch from backend");
101       }
102
103       const result = await response.json();
104       console.log("Response from Backend:", result); // { predicted_class: 6, accuracy: 0.99
105       const { accuracy, predicted_class } = result;
```

```
107         // Navigate to the Result page with accuracy and predicted_class as separa
108         router.push({
109           pathname: "/result",
110           params: {
111             accuracy: accuracy.toString(),
112             predicted_class: predicted_class.toString(),
113           },
114         });
115       } catch (error) {
116         console.error("Error:", error);
117         alert("Failed to communicate with the backend. Please try again.");
118       }
119     };
120
121     return (
122       <ImageBackground source={App} style={styles.background}>
123         <View style={styles.container}>
124           <Text style={styles.title}>Select Your Symptoms</Text>
125
126           <MultiSelect
127             data={symptoms.map((item) => ({ label: item, value: item }))}
128             labelField="label"
129             valueField="value"
130             placeholder="Select Symptoms"
131             value={selectedSymptoms}
132             onChange={handleSymptomChange}
133             style={styles.dropdown}
134             selectedStyle={styles.selectedStyle}
135             selectedTextStyle={styles.selectedText}
136             placeholderStyle={styles.placeholderStyle}
137           />
138
139           <View style={{ alignItems: "center" }}>
140             <TouchableOpacity style={styles.button} onPress={predictSymptom}>
```

```
141              <Text style={styles.buttonText}>Detect Disease</Text>
142            </TouchableOpacity>
143          </View>
144        </View>
145      </ImageBackground>
146    );
147  }
148
149  const styles = StyleSheet.create({
150    background: {
151      flex: 1,
152      resizeMode: "cover",
153    },
154    container: {
155      flex: 1,
156      padding: 20,
157    },
158    title: {
159      fontSize: 20,
160      fontWeight: "bold",
161      marginBottom: 10,
162      color: "#fff",
163    },
164    dropdown: {
165      borderWidth: 1,
166      borderColor: "#ccc",
167      borderRadius: 8,
168      padding: 10,
169      backgroundColor: "#fff",
170    },
171    selectedStyle: {
172      backgroundColor: "#e0f7fa",
173      padding: 5,
174      borderRadius: 5,
175      marginVertical: 5,
```

```
176     },
177     selectedText: {
178       color: "#00796b",
179       fontSize: 16,
180     },
181     placeholderStyle: {
182       color: "gray",
183       fontSize: 16,
184     },
185     button: {
186       padding: 16,
187       backgroundColor: "#0000FF",
188       borderRadius: 90,
189       alignItems: "center",
190       marginTop: 10,
191       width: Dimensions.get("screen").width * 0.5,
192     },
193     buttonText: {
194       color: "#fff",
195       fontSize: 18,
196       fontWeight: "bold",
197     },
198   });
199
```

Figure 22: Home Screen Code

- This component appears to be the main screen for inputting symptoms and getting a prediction.
- It imports numerous symptom names as strings in an array symptoms.
- It uses useState to manage the selectedSymptoms array, initialised as an array of 53 zeros.
- A function createSymptomVector takes the selectedSymptoms array and potentially a new symptom id, returning a new array where the value at the given id is set to 1.
- An async function predictSymptom is defined to handle the prediction process. It logs the symptomVector (likely the selectedSymptoms array).
- It makes an API call to http://192.168.120.5088/api/predict using fetch, sending the symptomVector in the request body as JSON.

- Upon receiving the response, it parses it as JSON and extracts predicted_class and accuracy.
- It uses useRouter from expo-router to navigate to the /result page, passing the predicted_class and accuracy as search parameters.
- Error handling is included for both the fetch operation and processing the result.
- The UI includes an ImageBackground, text elements ("Select Symptoms", "Please select your symptoms from the list below"), and a button to trigger prediction.
- It maps over the symptoms array to render selectable text elements (<Text>) for each symptom. Tapping a symptom calls createSymptomVector and updates the selectedSymptoms state.
- Stylesheet definitions are provided for various UI elements (cover, heading, subHeading, symptomListContainer, symptomItem, selectedSymptomItem, symptomText, selectedSymptomText, button, buttonText).

Profile (Screen):

```
import { View, Text, StyleSheet, ImageBackground } from "react-native";
import React from "react";
import { useUser } from "@clerk/clerk-expo"; // Import useUser to access u
import { SafeAreaView } from "react-native-safe-area-context";
import Logout from "../(auth)/logout";
import Profilepic from "../../assets/images/profile.png";

export default function Profile() {
  const { user } = useUser(); // Access the logged-in user's information

  return (
    <ImageBackground source={Profilepic} style={styles.background}>
      <View style={styles.container}>
        <SafeAreaView style={styles.safeArea}>
          <View style={styles.container}>
            <Text style={styles.welcomeText}>Welcome </Text>
            <Text style={styles.boldText}>
              {user?.fullName || user?.emailAddress}
            </Text>

            <Logout />
          </View>
        </SafeAreaView>
      </View>
    </ImageBackground>
  );
}
```

```
29  const styles = StyleSheet.create({
30    safeArea: {
31      flex: 1,
32    },
33    container: {
34      flex: 1,
35      justifyContent: "center",
36      alignItems: "center",
37    },
38    welcomeText: {
39      fontSize: 30,
40      textAlign: "center",
41      marginTop: 100,
42    },
43    boldText: {
44      fontSize: 30,
45      textAlign: "center",
46      fontWeight: "bold",
47      marginTop: 10,
48      elevation: 50,
49      shadowColor: "#fff",
50    },
51    background: {
52      flex: 1,
53      resizeMode: "cover",
54    },
55  });
```

Figure 23: Profile Screen Code

- This component displays the logged-in user's information.
- It imports useUser from @clerk/clerk-expo to access the logged-in user's information.
- It imports logout from ../../auth/logout, suggesting it includes the logout functionality within this screen.
- It uses ImageBackground, SafeAreaView, View, and Text for the UI.

- It accesses the user object via the useUser hook.
- The UI displays "Welcome" text and the user's full name or email address ({user?.fullName || user?.emailAddress}).
- A <Logout /> component is rendered, which likely provides the logout button discussed previously.
- Stylesheet definitions (safeArea, container, welcomeText, boldText, background) are provided.

Result (Screen):

```
import React from "react";
import {
  View,
  Text,
  StyleSheet,
  TouchableOpacity,
  ImageBackground,
} from "react-native";
import { useRouter, useLocalSearchParams } from "expo-router";
import Final from "../assets/images/result.png";
import classDescriptions from "../components/classDescriptions"; // Import the class description

export default function Result() {
  const router = useRouter();
  const { accuracy, predicted_class } = useLocalSearchParams(); // Retrieve accuracy and predict

  const handleGoBack = () => {
    router.back();
  };

  // Get the description for the predicted class
  const predictedDescription =
    classDescriptions[predicted_class] || "No description available";

  // Get the description for class 10
  const fallbackDescription =
    classDescriptions[10] || "Fallback description for class 10";

  return (
    <ImageBackground source={Final} style={styles.background}>
      <View style={styles.container}>
        {accuracy && parseFloat(accuracy) >= 0.7 ? (
          // If accuracy is greater than or equal to 0.9, show the predicted class and accuracy
          <>
            <Text style={styles.text}>{predictedDescription}</Text>
          </>
        ) : (
          // If accuracy is less than 0.9, show the fallback description for class 10
          <Text style={styles.text}>{fallbackDescription}</Text>
        )}
```

```
41          <TouchableOpacity onPress={handleGoBack} style={styles.button}>
42            <Text style={styles.buttonText}>Go Back</Text>
43          </TouchableOpacity>
44        </View>
45      </ImageBackground>
46    );
47  }
48
49  const styles = StyleSheet.create({
50    container: {
51      flex: 1,
52      justifyContent: "center",
53      alignItems: "center",
54    },
55    text: {
56      fontSize: 18,
57      fontWeight: "bold",
58      marginBottom: 100,
59      textAlign: "center",
60      color: "#000",
61    },
62    button: {
63      padding: 16,
64      backgroundColor: "#0000FF",
65      borderRadius: 90,
66      alignItems: "center",
67      width: 150,
68    },
69    buttonText: {
70      color: "#fff",
71      fontSize: 16,
72      fontWeight: "bold",
73    },
74    background: {
75      flex: 1,
76      resizeMode: "cover",
77    },
78  });
```

Figure 24: Profile Screen Code

- This component displays the disease prediction result.
- It imports useLocalSearchParams from expo-router to retrieve parameters passed during navigation (specifically accuracy and predicted_class).
- It imports classDescriptions from ../../components/classDescriptions.
- It uses useRouter to get the router object for navigation back.
- A handleGoBack function is defined, which calls router.back() to navigate to the previous screen.
- It retrieves accuracy and predicted_class using useLocalSearchParams.
- It gets the description for the predicted_class from the imported classDescriptions object. A fallback description is provided if the predicted class description is not available. It also retrieves a fallback description specifically for class 10 from classDescriptions.
- The UI uses an ImageBackground, SafeAreaView, View, and Text.
- It checks if the accuracy is greater than or equal to 0.7.
- If the accuracy is sufficient, it displays the predicted description (predictedDescription).
- If the accuracy is less than 0.7, it displays the fallback description for class 10 (fallbackDescriptionForClass10).
- A TouchableOpacity button with the text "Go Back" is included, whose onPress handler is handleGoBack.
- Stylesheet definitions (container, text, button, buttonText, background) are provided.

classDescriptions (Data):

```
const classDescriptions = {
  0: "It looks like you might be experiencing an allergy. Allergies can be managed with the right care–consider consulting a doctor for personalized advice.'
  1: "Arthritis can be challenging, but you're not alone. Speak with a doctor to explore treatment options and find relief.",
  2: "Chickenpox detected. It's important to rest and avoid spreading it–consult a doctor for guidance on care and recovery.",
  3: "Dengue fever requires prompt attention. Stay hydrated and consult a doctor immediately for proper care.",
  4: "Diabetes is manageable with the right support. Connect with a doctor to create a plan tailored to your needs.",
  5: "Malaria needs swift action. Consult a doctor for diagnosis and treatment to ensure a smooth recovery.",
  6: "Migraines can be tough, but help is available. Talk to a doctor about ways to reduce their impact on your life.",
  7: "Paralysis from a brain hemorrhage is serious. Seek immediate medical attention and ongoing support for the best possible outcome.",
  8: "Pneumonia requires careful management. Consult a doctor to discuss treatment and recovery steps.",
  9: "Typhoid fever needs medical intervention. Reach out to a doctor for diagnosis and a treatment plan.",
  10: "We couldn't identify your symptoms in our current database. Please consult a specialist for a thorough evaluation and guidance.",
};

export default classDescriptions;
```

Figure 25: Class Descriptions Data

- This is an exported constant object named classDescriptions.
- It contains key-value pairs where keys are numeric class IDs (0 through 10) and values are string descriptions of potential conditions. These descriptions provide brief information and advice related to the predicted class. For example, class 0 is described as "It looks like you might be experiencing an allergy", and class 10 is a general message indicating the symptoms couldn't be identified in the current database and recommending a specialist.

Appendix C Backend Structure and Components

This appendix describes the backend application responsible for processing symptom data and providing disease predictions via an API endpoint. The backend is built using Python and leverages several key libraries for web serving, data handling, and machine learning model inference.

Core Framework and Setup

```python
# Import libraries
from flask import Flask, request, jsonify
import numpy as np
import joblib
import tensorflow as tf
from pyngrok import ngrok

# Initialize Flask app
app = Flask(__name__)

# Load the saved components
scaler = joblib.load("./models/scaler.pkl")
model = tf.keras.models.load_model("./models/final_model.keras")

# Define the prediction endpoint
@app.route('/api/predict', methods=['POST'])
def predict():
    data = request.get_json()
    symptom_vector = data.get('symptomVector', [])
    if len(symptom_vector) != 53:
        return jsonify({"error": "Symptom vector must contain exactly 53 values"}), 400
    input_array = np.array(symptom_vector).reshape(1, -1)
    input_scaled = scaler.transform(input_array)
    predictions = model.predict(input_scaled)
    predicted_class = int(np.argmax(predictions, axis=1)[0])
    max_prob = float(np.max(predictions))
    return jsonify({"predicted_class": predicted_class, "accuracy": max_prob})

# Set up ngrok with your auth token
ngrok.set_auth_token("2ufXOGrU5h5j5kVGyL8uIgraQ8j_7W1ZWYXoT5tHA1GC39J3a")
public_url = ngrok.connect(5000)
print(f"Public URL: {public_url}")

# Run Flask app in a separate thread
from threading import Thread
def run_flask():
    app.run(host="0.0.0.0", port=5000, debug=True, use_reloader=False)
Thread(target=run_flask).start()
```

Figure 26: Flask Backend Code

Flask Application

- The backend is implemented as a Flask web application. Flask is a lightweight micro web framework for Python.
- The application is initialised as app = Flask(__name__).

Library Imports

- The script imports necessary libraries including Flask, request, and jsonify from flask for web handling.
- numpy (imported as np) is used for numerical operations, particularly array manipulation.
- joblib is used for loading the pre-trained scaler model.
- tensorflow (imported as tf) is used for loading and running the machine learning prediction model.
- ngrok is used to expose the local Flask development server to the internet.
- threading is used to run the Flask application in a separate thread, presumably to allow the ngrok tunnel to be established and the public URL to be printed.

Model Loading

- Upon initialization, the backend loads two essential components required for prediction:
- A scaler model is loaded from the path "./models/scaler.pkl" using joblib.load(). This scaler is likely a StandardScaler or similar from scikit-learn, used to transform the input symptom data into a suitable format for the prediction model.
- The main prediction model is loaded from the path "./models/final_model.keras" using tf.keras.models.load_model(). This is a TensorFlow Keras model trained for disease classification.

Prediction Endpoint

/api/predict Endpoint

- A dedicated API endpoint, /api/predict, is defined using the @app.route('/api/predict', methods=['POST']) decorator. This endpoint is configured to accept POST requests.
- The predict() function is executed when a POST request is made to this endpoint.

Input Data Handling

- The function expects the input data to be in JSON format, containing a key named "symptom_vector".
- It retrieves this JSON data using request.get_json().

- It specifically extracts the symptom_vector list from the received JSON.

Input Validation

- A crucial validation step is performed to ensure the received symptom_vector has the correct dimensions.
- It checks if the len(symptom_vector) is exactly 53.
- If the length is not 53, it returns a JSON response with an error message "Symptom vector must contain exactly 53 values" and an HTTP status code of 400 (Bad Request).

Data Processing and Prediction

- If the input is valid (length 53), the symptom_vector is converted into a numpy array.
- The array is then reshaped into a 2D array with one row and 53 columns (.reshape(-1, 1)). Correction based on source code: It reshapes to (-1, 1), which means a column vector. However, typical model input expects (1, 53) for a single sample. Looking closer at the code, reshape(-1, 1) is applied to symptom_vector, which is then passed to the scaler.transform. This suggests the scaler expects a column vector input. The scaled output is then reshaped to (input_array.shape, -1) which means preserving the number of rows (1) and inferring the number of columns (53). Let's describe the flow as per the code: The symptom_vector is converted to a numpy array input_array. This input_array is reshaped to (-1, 1) for scaling. The scaler.transform is applied to this reshaped array, resulting in input_scaled. The input_scaled is then reshaped back using (input_array.shape, -1) before being passed to the model for prediction.
- The input_scaled data is passed to the loaded TensorFlow Keras model to generate predictions (model.predict(input_scaled)).
- The predicted_class is determined by finding the index of the maximum value in the prediction output using np.argmax(predictions, axis=1).
- The maximum probability (max_prob) associated with the predicted class is extracted using np.max(predictions) and converted to a float.

Output Response

- The function returns a JSON response containing the predicted_class and its corresponding accuracy (which is the max_prob).

External Exposure (ngrok)

ngrok Tunnel

- The backend script integrates ngrok to create a public URL for the locally running Flask app.
- It sets up ngrok with a provided authorization token.
- It connects ngrok to the local Flask server running on port 5000.
- The generated public URL is printed to the console.

Running the Application

Threading

- The Flask application is run in a separate thread using the threading module.
- The run_flask() function, which calls app.run() to start the Flask development server on host 0.0.0.0 and port 5000, is the target of this thread.
- Debugging is enabled (debug=True), but the reloader is disabled (use_reloader=False) when running in the thread.
- Starting the Flask app in a thread allows the main script to continue and set up the ngrok tunnel after the Flask server has started listening.

Dependencies

```
absl-py==2.1.0
aiohappyeyeballs==2.4.4
aiohttp==3.11.10
aiosignal==1.2.0
astunparse==1.6.3
async-timeout==5.0.1
attrs==24.3.0
blinker==1.9.0
Brotli==1.0.9
cachetools==5.5.1
certifi==2025.1.31
cffi==1.17.1
charset-normalizer==3.3.2
click==8.1.7
colorama==0.4.6
cryptography==41.0.3
Flask==3.1.0
flatbuffers==24.3.25
frozenlist==1.5.0
gast==0.4.0
google-auth==2.38.0
google-auth-oauthlib==0.4.4
google-pasta==0.2.0
grpcio==1.71.0
h5py==3.12.1
idna==3.7
itsdangerous==2.2.0
Jinja2==3.1.6
joblib==1.4.2
keras==3.9.0
Keras-Preprocessing==1.1.2
libclang==18.1.1
Markdown==3.4.1
markdown-it-py==3.0.0
MarkupSafe==3.0.2
mdurl==0.1.2
mkl_fft==1.3.11
mkl_random==1.2.8
mkl-service==2.4.0
ml_dtypes==0.5.1
```

```
multidict==6.1.0
namex==0.0.8
numpy==1.26.4
oauthlib==3.2.2
opt-einsum==3.3.0
optree==0.14.1
packaging==24.2
pip==25.0.1
platformdirs==3.10.0
pooch==1.8.2
propcache==0.2.0
protobuf==3.20.3
pyasn1==0.4.8
pyasn1-modules==0.2.8
pycparser==2.21
Pygments==2.19.1
PyJWT==2.10.1
pyngrok==7.2.3
pyOpenSSL==23.2.0
PySocks==1.7.1
PyYAML==6.0.2
requests==2.32.3
requests-oauthlib==2.0.0
rich==13.9.4
rsa==4.7.2
scikit-learn==1.6.1
scipy==1.15.1
setuptools==75.8.0
six==1.16.0
tensorboard==2.19.0
tensorboard-data-server==0.7.2
tensorboard-plugin-wit==1.8.1
tensorflow==2.19.0
tensorflow-estimator==2.10.0
tensorflow-io-gcs-filesystem==0.31.0
termcolor==2.1.0
threadpoolctl==3.5.0
typing_extensions==4.12.2
urllib3==2.3.0
Werkzeug==3.1.3
wheel==0.45.1
win-inet-pton==1.1.0
wrapt==1.17.0
yarl==1.18.0
```

Figure 27: Dependencies

Required Packages

The backend relies on several Python packages, as listed in the requirements. Key dependencies supporting the described functionality include:

- Flask
- numpy
- joblib
- tensorflow
- ngrok
- scikit-learn (implicitly required for the scaler model format loaded by joblib)

References

[1] Zawati. M, Lang. M, ‘Does an App a Day Keep the Doctor Away? AI Symptom Checker Applications, Entrenched Bias, and Professional Responsibility’, J Med Internet Res 2024, https://www.jmir.org/2024/1/e50344

[2] Youjin Hwang, Taewan Kim, Junhan Kim, Joonhwan Lee, Hwajung Hong, ‘Leveraging challenges of an algorithm-based symptom checker on user trust through explainable AI’, 2021, https://francisconunes.me/RealizingAIinHealthcareWS/papers/Hwang2021.pdf

[3] Aerts A, Bogdan-Martin D. Leveraging data and AI to deliver on the promise of digital health. Int J Med Inform. 2021 Jun;150:104456. doi: 10.1016/j.ijmedinf.2021.104456. Epub 2021 Apr 10. PMID: 33866232. https://pubmed.ncbi.nlm.nih.gov/33866232/

[4] Mohammad I. Merhi. 2023. An evaluation of the critical success factors impacting artificial intelligence implementation. Int. J. Inf. Manag. 69, C (Apr 2023). https://doi.org/10.1016/j.ijinfomgt.2022.102545

[5] Fuster-Palà, A.; Luna-Perejón, F.; Miró-Amarante, L.; Domínguez-Morales, M. Optimized Learning Classifiers for Symptom-Based Screening. Computers 2024, 13, 233. https://doi.org/10.3390/computers13090233 Disease

[6] Ahsan, M. M., Luna, S. A., & Siddique, Z. (2022). Machine-Learning-Based Disease Diagnosis: A Comprehensive Review. *Healthcare*, *10*(3), 541. https://doi.org/10.3390/healthcare10030541

[7] You Y, Gui X. Self-Diagnosis through AI-enabled Chatbot-based Symptom Checkers: User Experiences and Design Considerations. AMIA Annu Symp Proc. 2021 Jan 25;2020:1354-1363. PMID: 33936512; PMCID: PMC8075525. https://pmc.ncbi.nlm.nih.gov/articles/PMC8075525/