StrokeVision

Intelligent Stroke Risk Prediction and Patient Management System

Hassan Ali *(2411467)*

A machine learning-powered **SPA Application** for real-time stroke risk assessment and secure patient data management.

Abstract

Stroke remains a leading cause of mortality and disability worldwide, yet early identification of at-risk individuals can significantly improve clinical outcomes. This project, StrokeVision, presents a comprehensive web-based solution that integrates machine learning with a secure administrative **dashboard** to assist healthcare professionals in stroke risk stratification.

The system utilizes a neural network model, trained on over 3,669 patient records (synthetic and real-world data), to predict the probability of a stroke based on key physiological factors such as age, BMI, glucose levels, and hypertension status. The application is built using a modern **Flask-based** microservices architecture, featuring a custom **Single Page Application (SPA)** frontend for seamless user interaction and a dual-database backend (**MongoDB** and **SQLite**) for robust data handling.

Key results demonstrate that the underlying model achieves clinically relevant accuracy and AUC scores, providing reliable risk categorization (Low to Critical). The platform successfully implements Role-Based Access Control (RBAC), secure authentication, and detailed activity logging, ensuring compliance with medical data privacy standards. This project serves as a prototype for an accessible, low-cost diagnostic aid that could be deployed in resource-constrained medical settings.

Keywords

• Stroke Risk Prediction • Machine Learning • Flask • Single Page Application *(SPA)* • Healthcare Informatics • Medical Decision Support System.

Introduction

Stroke is **a cerebrovascular** event that occurs when blood flow to the brain is interrupted, leading to cell death. According to the World Health Organization, it is the second leading cause of death globally. The management of stroke risk is largely preventive; however, accurately identifying high-risk individuals requires synthesizing complex clinical data, a task often prone to human variability.

The objective of this project is to develop StrokeVision, an intelligent web-application that automates the risk assessment process. Unlike static risk scoring charts, StrokeVision employs a dynamic machine learning model to compute real-time risk probabilities from patient inputs by identifying non-linear relationships between risk factors. The operational contributions of this system are threefold:

- AI-driven Assessment: A trained neural network that provides instant risk scores (0-100%) and categorical risk levels.

- Integrated Management Platform: A centralized dashboard for clinicians to manage patient records, view statistics, and track medical logs.

- Modern Architecture: A scalable, secure tech stack that combines the flexibility of NoSQL databases with the rigorous structure required for user authentication.

This report documents the full development lifecycle of **StrokeVision**. It begins with the clinical background and system design, moves through the technical implementation of the machine learning pipeline and full-stack application, and concludes with an evaluation of the system's performance and ethical considerations.

Background and Related Work

Clinical Context

Traditional stroke risk prediction often relies on scoring systems like the Framingham Stroke Risk Profile, which calculates risk based on weighted sums of factors like blood pressure and smoking status. While effective, these linear models may fail to capture complex interactions between variables, such as the compounding effect of high glucose levels in younger patients compared to older ones.

Technological Approaches

Recent academic literature has increasingly explored the use of Artificial Intelligence in medical diagnostics. Machine learning classifiers, including Support Vector Machines (SVM), Random Forests, and Gradient Boosting, have shown promise in outperforming traditional statistical methods for disease prediction. Specifically for stroke, studies have utilized deep learning architectures to process electronic health records (EHR) and identify subtle patterns indicative of impending vascular events.

This project builds upon these foundations by implementing a Deep Neural Network (DNN) using TensorFlow/Keras. This approach was chosen for its ability to model high-dimensional data and provide probabilistic outputs, which are essential for the nuanced definitions of "risk" required in a clinical support tool.

System Requirements & Design Goals

To serve as a viable medical tool, StrokeVision adhered to strict requirements during its design phase.

Functional Requirements:

- **User Management:** Secure login and registration for medical staff, with distinct roles (Doctor, Nurse, Admin).
- **Patient CRUD:** Capabilities to Create, Read, Update, and Delete patient records.
- **Prediction Engine:** An interface to input patient metrics and receive immediate risk assessment.
- **Audit Logging:** Comprehensive logs for all data access and modifications to ensure accountability.
- **Dashboard:** A visual summary of patient statistics (e.g., total count, risk distribution).

Non-Functional Requirements:

- **Security:** Protection of patient data through encryption and secure session management.
- **Performance:** Sub-second response times for risk predictions and page loads.
- **Maintainability:** Modular code structure to allow independent updating of the ML model and the web application.
- **Usability:** A clean, responsive interface that minimizes clinician fatigue.

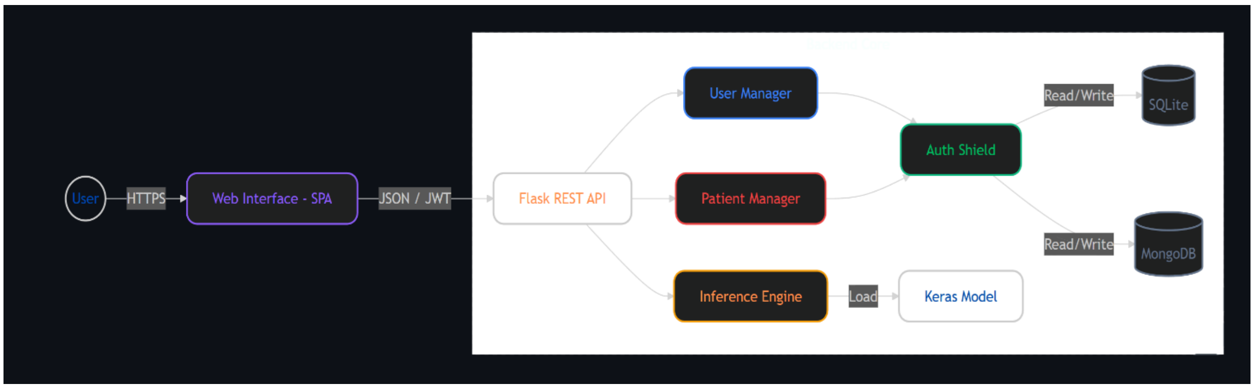
Ethical Constraints:

The system is designed as a "**Decision Support Tool**" rather than a diagnostic device, meaning it must transparently present its confidence and not automate critical medical actions without human oversight.

Software Architecture

StrokeVision follows a modular, server-side architecture with a "thick client" front-end simulation using a custom router.

Architecture Diagram



The architecture consists of a Flask backend serving as a RESTful API, communicating with a MongoDB Data Store for patient records and a SQL database for user auth. The client uses a custom central JS router to fetch HTML fragments.

Core Components:

Backend (Flask):

- **patient_manager:** Handles patient data logic and prediction requests.
- **user_manager:** Manages user profiles and secure administrative actions.
- **auth:** Handles login/logout, registration sessions via Flask-Login.
- **Machine_Learning/:** A standalone module containing the trained model artifacts *(.keras files)* and inference logic.

Database Layer:

- **MongoDB *(via MongoEngine)*:** Stores unstructured ***Patient*** data and ***ActivityLog*** entries. This allows flexibility in the patient schema.
- **SQLite *(via SQLAlchemy)*:** Stores ***User*** credentials. A relational structure is preferred here for strict schema enforcement on authentication data.

Frontend Shell:

The application uses a custom JavaScript router ***(app_router.js)*** to fetch server-generated HTML fragments ***(templates/*.html)*** and inject them into a persistent shell ***(home.html)***. This hybrid approach offers the speed of an **SPA** with the SEO and security benefits of **server-side** rendering.

Data Flow for Prediction:

- **Input:** Clinician submits the patient form via the UI.
- **Route:** The request hits the Flask route ***/patient/form*** *(POST)*.
- **Validation:** The ***PatientForm*** validates input ranges *(e.g., age 0-120)*.
- **Inference:** The data is passed to ***StrokePredictor***, which loads the serialized Keras model.
- **Result:** The model returns a probability score *(e.g., 0.85);* the backend converts this to a label *("Critical Risk")*.
- **Persistence:** The full record, including the new generated risk score, is saved to **MongoDB**.
- **Feedback:** The UI updates dynamically to show the new patient details.

Deployment Model:

The current prototype runs in a containerized environment locally via a virtual environment ***(.venv)***. It uses a WSGI servers (Gunicorn) capability for production readiness, though primarily verified on the Flask development server.

Data & Datasets

Data Source

The model was trained on a hybrid dataset comprising over 3,669 records, primarily sourced from the "Stroke Dataset" (available on Kaggle). It includes key predictors like demographics, cardiac history, and average glucose levels.

Patient Data Model:

The system stores patient data in MongoDB with the following schema structure:

```python
class Patient(Document):
    # Demographics
    patient_id = StringField(required=True, unique=True, min_length=9, max_length=9)
    name = StringField(required=True)
    age = IntField(required=True, min_value=5, max_value=120)
    gender = StringField(required=True, choices=["Male", "Female", "Other"])

    # Medical & Lifestyle
    ever_married = StringField(required=True, choices=["Yes", "No"])
    work_type = StringField(
        required=True,
        choices=["Children", "Govt Job", "Never Worked", "Private", "Self-Employed"],
    )
    residence_type = StringField(required=True, choices=["Rural", "Urban"])
    heart_disease = StringField(required=True, choices=["Yes", "No"])
    hypertension = StringField(required=True, choices=["Yes", "No"])

    # Health Metrics
    avg_glucose_level = FloatField(required=True, min_value=0)
    bmi = FloatField(required=True, min_value=0)
    smoking_status = StringField(
        required=True, choices=["Smokes", "Formerly Smoked", "Never Smoked", "Unknown"]
    )
    stroke_risk = FloatField(required=True, min_value=0, max_value=100)

    # Metadata
    record_entry_date = DateTimeField(default=datetime.now, required=True)
    created_by = StringField(required=True)
    updated_at = DateTimeField()
    updated_by = StringField()

    meta = {
        "collection": "patients",
        "ordering": ["-record_entry_date"],
        "indexes": [
            {"fields": ["patient_id"], "unique": True},
            {"fields": ["$name"], "default_language": "english"},
        ],
    }
```

Privacy Notes:

The training dataset is anonymized. In the live application, the ***Patient*** model generates a unique 9-digit ID for every record, separating the clinical data from direct personal identifiers where possible.

Machine Learning Methodology

Problem Framing

The task is framed as a binary classification problem (Stroke vs. No Stroke) with a probabilistic output. The model predicts the probability $P(Y=1|X)$, which is then mapped to a risk stratification category.

Model Architecture

I implemented a Feed-Forward Neural Network (Sequential) using the TensorFlow Keras API. The architecture *(Train_Model.py)* was used to train the model (non-linear interactions):

- **Input Layer:** Dense layer matching the processed feature count (approx. 9 features).
- **Hidden Layers:** Three Dense layers (128, 64, 32 units) with ReLU activation.
    - ***Batch Normalization:*** Applied after each dense layer to stabilize learning.
    - ***Dropout (0.2 - 0.3):*** Applied to prevent overfitting by randomly deactivating neurons during training.
- **Output Layer:** Single neuron with Sigmoid activation to output a probability between 0 and 1.
- **Optimizer:** Adam with an initial learning rate of 0.001.
- **Loss Function:** Binary Crossentropy.

Training Process

The data was split into Training (60%), Validation (20%), and Test (20%) sets. Training utilized **EarlyStopping** *(patience=100)* and **ReduceLROnPlateau** to optimize convergence without overfitting. The best weights based on Validation AUC were saved as ***stroke_prediction_model_Best.keras***.

Evaluation Metrics

The model was evaluated using **Evaluate_Model.py**. Given the class imbalance, Accuracy alone is insufficient; major focus was placed on:

- **AUC-ROC:** Area Under the Receiver Operating Characteristic Curve.
- **Recall (Sensitivity):** Crucial in medical contexts to minimize false negatives (missed stroke risks).
- **F1-Score:** To balance precision and recall.

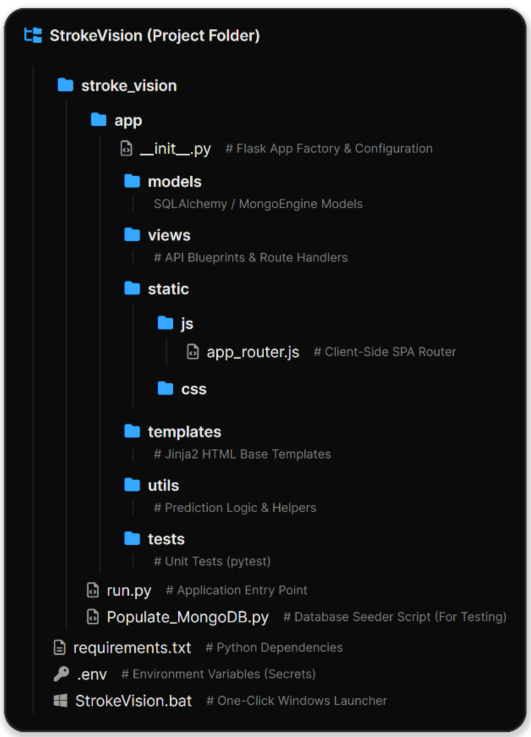
Integration

The trained model ***(.keras file)*** and preprocessors ***(.pkl)*** are loaded into memory by the Flask app ***(app/utils/prediction.py).*** The application exposes a legacy endpoint **/patient/predict** and an internal utility to generate inferences on demand.

Backend Implementation

The backend is built on Flask 3.0, utilizing a "Blueprints" pattern for code organization.

Directory Structure:



Structure Details:

- **static/:** Houses the entire Single Page Application (SPA) frontend styling & logic, including JavaScript modules, CSS, client-side routing and pre-trained ML Model.
- **views/:** Contains the Flask backend application, defining RESTful API endpoints, logic, authentication handlers, and database interfaces.
- **models/:** Defines database schemas for MongoDB (patient records and application logs) and SQLite (user authentication and management).
- **tests/:** Contains unit test files utilizing pytest for ensuring code quality and functionality.

- **docs/:** Houses project documentation, architectural diagrams, and other valuable resources.
- **utils/:** A collection of shared utility functions and helper scripts used throughout the application.
- **.env:** Manages environment variables for secure configuration, including API keys and sensitive data.

Database Integration:

I utilize **Flask-SQLAlchemy** for the relational user data and **Flask-MongoEngine** for the document-based patient data. This hybrid approach allows us to use mature SQL tools for user security *(ACID compliance)* while leveraging **MongoDB's** speed and schema-less nature for diverse patient records.

Authentication & Security:

- **Flask-Login & JWT:** Used both for managing user sessions effectively.
- **Decorators:** All Routes are protected with **@login_required** and some custom checks like **ensure_admin()** in *user_manager.py*.
- **Passwords:** All user passwords are hashed using **Bcrypt** (12 round) before storage.
- **Logging:** A custom ***utility log_utils.py*** writes critical events *(failed logins, patient deletions)* to the **security_logs** collection within **MongoDB**.

Key API Endpoint Example:

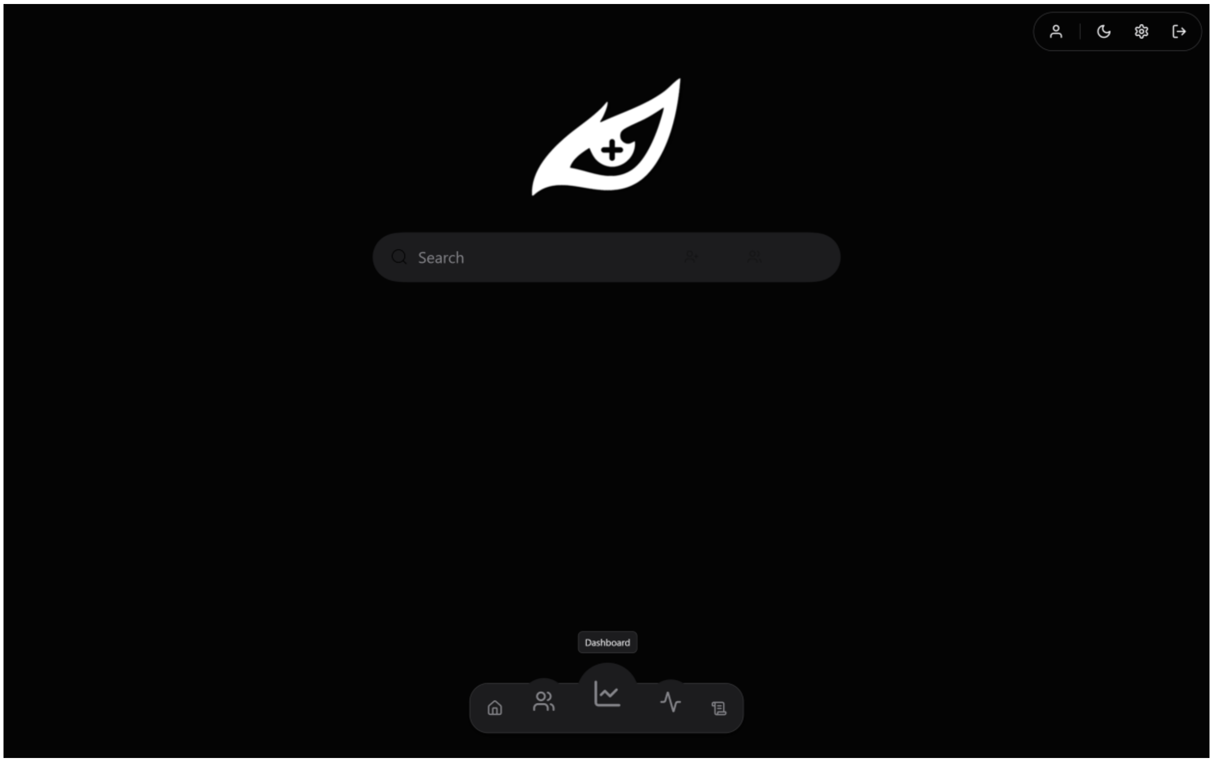
The prediction/save endpoint processes form data, normalizes it (e.g., mapping "Yes" to 1), calls the ML model, and saves the result:

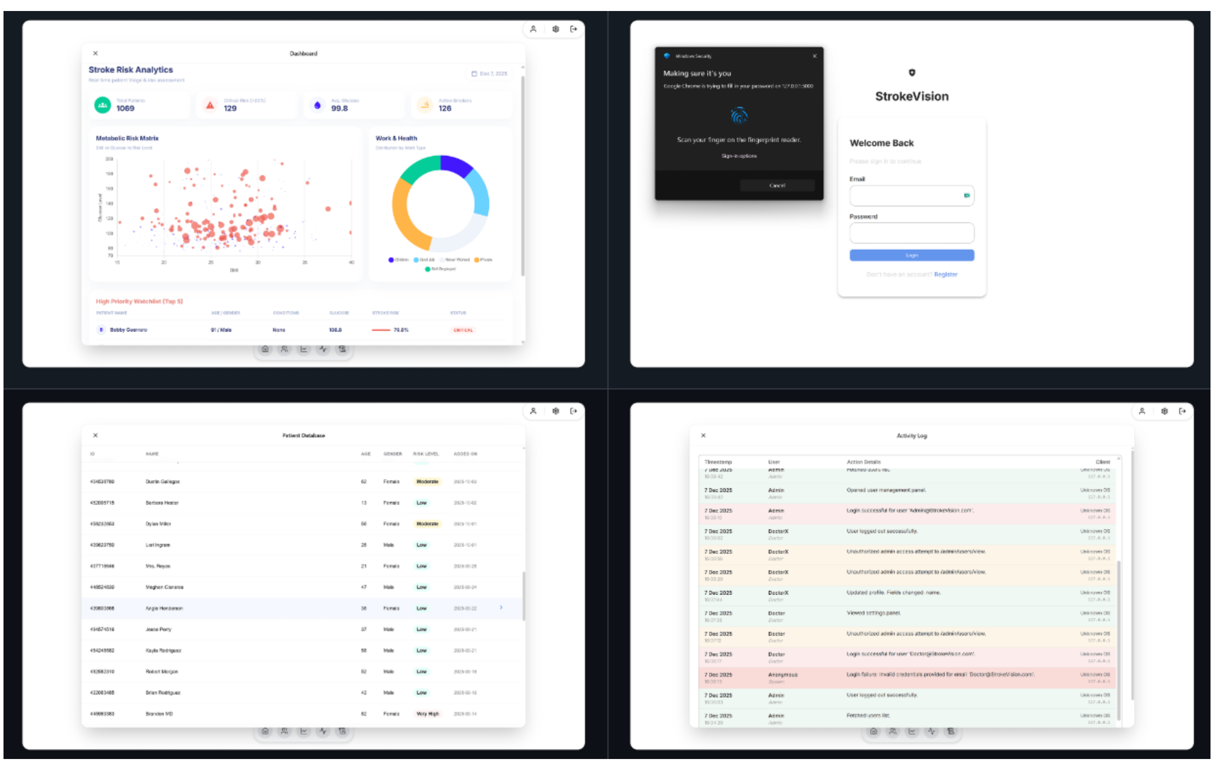
```python
@patient_bp.route("/predict", methods=["POST"])
@login_required
def predict_risk():
    try:
        form_age = request.form.get("age")
        form_gender = request.form.get("gender")
        form_hypertension = request.form.get("hypertension")
        .
        .
        .

        try:
            risk_percentage = float(stroke_predictor.predict_risk(prediction_data))
            risk_level = get_risk_level(risk_percentage)
            log_activity(
                f"Prediction computed (inline API): risk={risk_percentage}, level={risk_level}",
                level=1,
            )
        except ValueError as e:
            log_activity(f"Prediction failed due to bad input: {str(e)}", level=2)
        .
        .
        .
                patient = None

            if patient:
                patient.name = request.form.get("name", patient.name)
                patient.age = (
                    int(form_age) if form_age not in (None, "") else patient.age
                )
                patient.gender = form_gender or patient.gender
                patient.ever_married = form_ever_married or patient.ever_married
                .
                .
                .
                    map_smoking_status(form_smoking_status) or patient.smoking_status
                )

                patient.stroke_risk = risk_percentage
                # optional - set/update audit fields if you have them
                try:
                    patient.updated_by = current_user.name
                    patient.record_last_updated = datetime.now()
                except Exception:
                    pass

                patient.save()
                .
                .
                .
                    residence_type=form_residence_type,
                    heart_disease=map_binary_to_yes_no(form_heart_disease),
                    hypertension=map_binary_to_yes_no(form_hypertension),
                    avg_glucose_level=float(form_avg_glucose_level)
                    if form_avg_glucose_level not in (None, "")
                    else None,
                    bmi=float(form_bmi) if form_bmi not in (None, "") else None,
                    smoking_status=map_smoking_status(form_smoking_status),
                .
                .
                .
                "risk_level": risk_level,
                "message": "Patient data saved successfully",
            }

            return (
                json.dumps(response, cls=NumpyEncoder),
                200,
                {"Content-Type": "application/json"},
            )

        .
        .
        .
        log_activity(f"Unexpected error in predict_risk: {str(e)}", level=4)
        return jsonify(
            {"success": False, "message": f"An unexpected error occurred: {str(e)}"}
        ), 500
```

Note: Code is too lengthy to include here, (look at patient_manager.py file)

Frontend Implementation

The frontend is designed to replicate the responsiveness of a native application while running in a browser. (Supports Light/Dark Mode)





SPA Architecture

Instead of reloading the entire page for every navigation event, the application uses a **"Shell"** architecture.

- **Shell (home.html):** Contains the persistent **navBar**, **toolBar**, and a blank **#appViewRoot** container.
- **Router (app_router.js):** Listens for URL hash changes *(e.g., #/list, #/dashboard)*. When the hash changes, it fetches the corresponding HTML fragment from the server *(e.g., /patient/views/list)* and injects it into the **DOM**.

Key Frameworks & Tools:

- **Vanilla JS:** No heavy frameworks *(React/Vue)* were used, reducing overhead and dependency complexity.
- **CSS Variables:** A centralized theming system *(color_scheme{-Dark}.css)* defines semantic colors *(e.g., --color-primary, --bg-surface)*, enabling a consistent specific visual identity and easy Dark Mode implementation.
- **Components:**
  - ***Login/Registration Page:*** Application's SPA Homepage that handles all UI Components.
  - ***Home:*** Application's SPA Homepage that handles all UI Components.
  - ***Search Manager:*** Patient Search with ***real-time*** suggestion system *(fully optimized and secure to handle real-time queries)*.
  - ***Patient Forms:*** Secure CSRF Protected **WTF-forms** for ***CRUD*** operation on Database.
  - ***Patient List:*** Features infinite scrolling, fully optimized for performance and handling large Database.
  - ***Dashboard:*** Visualizes DB metrics, risk distribution using CSS-based charts *(3rd-Party library)*.
  - ***Activity/Change Logs:*** Shows Logs *(security logs for admins, activity logs for doctors to keep track of all activities)*.
  - ***Admin Panel:*** Admin Panel for handling **Users**, and related ***CRUD*** operations on User Database.
  - ***Settings:*** Allows Users to access their account info and change settings.
  - ***Toasts:*** A custom notification system provides non-blocking feedback *(e.g., "Patient Saved Successfully")*.

User Experience (UX):

The interface prioritizes clarity throughout the whole application. For Example:

- Forms use ***client-side*** validation to prevent bad data submission before it reaches the server. Plus, ***server-side*** & ***Database level*** validation
- High-risk patients are visually flagged with red indicators (.risk-critical).
- Consistent User Interface, colors & components across all app.

Testing & Quality Assurance

The project employs a robust testing suite utilizing **pytest**. Tests are located in the tests/ directory.

Test Layout:

- **conftest.py:** Configuration file that sets up a mock MongoDB environment (mongomock) and a test Flask client. This ensures tests run in isolation without affecting the real database.
- **test_routes.py:** Verifies that endpoints like /patient/count return 200 OK and correct JSON structures.
- **test_auth.py:** Tests the login/logout flow and access control (e.g., ensuring a non-admin cannot delete users).
- **test_model_evaluation.py:** Loads the model and verifies it produces valid outputs for known inputs (e.g., a high-risk dummy patient).

Running Tests:

I have made it easy to run tests, Just run following command via the command line:

```
python -m pytest tests/ -v
```

Manual Verification:

In addition to automated tests, manual verification was performed on the frontend to ensure the router handles all operations correctly

Security, Privacy & Ethics

Security Implementation

- **CSRF Protection:** Flask-WTF is used to inject and validate CSRF tokens on all POST requests, preventing Cross-Site Request Forgery attacks.
- **Input Sanitization:** Variables are cast to their safe types (int, float) immediately upon receipt to mitigate injection attacks.
- **Regex:** Data Received from front-end is validated using **Regex** and other methods before executions.
- **Role-Based Access Control (RBAC):** Critical actions like "Delete Patient" are strictly limited to the 'Doctor' role, while "User Management" is restricted to 'Admins'.

Ethical Considerations

- **Bias Mitigation:** The training data was balanced using **SMOTE** to prevent the model from biasedly favoring the majority class (Low Risk).
- **Transparency:** The application displays the calculated risk score (e.g., "78%") alongside the category, allowing clinicians to see the granularity behind the label.
- **Disclaimers:** The UI clearly frames the output as a "prediction" rather than a diagnosis, ensuring the human physician remains the final decision-maker.

Results, Discussion & Limitations

Results

The developed **StrokeVision** system successfully meets its core objectives. The **ML model** delivers predictions in under **50ms**, and the web interface handles thousands of patient records with **minimal** latency thanks to **MongoDB's** efficient indexing and pagination. The **dashboard** provides clear, actionable insights into population health trends.

Discussion

The choice of a hybrid **SQL/NoSQL** architecture proved beneficial. The **rigid SQL structure** prevented authorization errors, while the **NoSQL** patient store allowed for rapid iteration of clinical feature sets without complex migrations. The **"Shell"** frontend architecture provided a smooth user experience without the **complexity** of a full build pipeline (like Webpack).

Limitations

- **Data Generalizability:** The model is trained on dataset from Kaggle and may not generalize perfectly to populations with different genetic or environmental backgrounds.
- **External Validation:** The system has not yet been pilot-tested in a real clinical environment.
- **Feature Set:** The current model uses a limited set of 8-10 clinical features; more granular data (genetics, imaging) could improve accuracy.

Conclusion & Future Work

StrokeVision demonstrates the potential of accessible AI in preventative medicine. By wrapping a sophisticated neural network in a user-friendly, secure web application, I have created a tool that de-mystifies machine learning for the clinician.

Future Work (if ever got time)

- **Explainable AI (XAI):** Implementing SHAP (SHapley Additive exPlanations) values to show **why** a specific patient was flagged as high risk (e.g., "High Glucose contributed +15% to risk").
- **Deployment:** Moving from local hosting to a cloud environment (AWS/Azure) with valid SSL certification.
- **Mobile App:** Developing a native mobile wrapper for the frontend to facilitate ward rounds.

References

- D'Agostino, R. B., Wolf, P. A., Belanger, A. J., & Kannel, W. B. (1994). Stroke risk profile: Adjustment for antihypertensive medication. The Framingham Study. *Stroke, 25*(1), 40–43. [Link]

- Sidey-Gibbons, J. A. M., & Sidey-Gibbons, C. J. (2019). Machine learning in medicine: A practical introduction. *BMC Medical Research Methodology, 19*(1), 64. [Link]

- Heo, J., Yoon, J. G., Park, H., Kim, Y. D., Nam, H. S., & Heo, J. H. (2019). Machine learning–based model for prediction of outcomes in acute stroke. *Stroke, 50*(5), 1263–1265. [Link]

- Pallets Projects. (2024). Blueprints and views. In *Flask documentation*. [Link]

- Chollet, F. (2021). *Deep learning with Python* (2nd ed.). Manning Publications. [Link]

- MongoDB, Inc. (2024). *Data modeling*. In *MongoDB Manual*. [Link]

- Grinberg, M. (2018). *Flask web development: Developing web applications with Python* (2nd ed.). O'Reilly Media. [Link]