Technical Specification & Deployment Guide:

A Language-Agnostic "Reading Profile" Model for Dyslexia Screening

Author: Project Lead, AI Engineering Date: November 17, 2025 Model Version: 3.0 (Profile Model, 89% Accuracy)

ABSTRACT

This document details the development, architecture, and critical preprocessing pipeline for the "Reading Profile" model, a deep learning system for the language-agnostic screening of dyslexia. The project's core hypothesis is that dyslexic reading exhibits universal, kinematic signatures that can be captured from eye-movement data, independent of the language being read. The final model is a multi-input classifier that achieves this by analyzing a participant's complete reading profile across three distinct diagnostic tasks (`Syllables` *,* `MeaningfulText` *,* `PseudoText` *). This system is built on a foundational LSTM encoder, pre-trained via self-supervision on the 150-hour OneStop dataset, and fine-tuned on the labeled ETDD70 dataset. This document details the iterative development journey, the final architecture, a precise data preprocessing pipeline essential for deployment, and the final model performance, which achieved 89% validation accuracy.*

I. INTRODUCTION & DEVELOPMENT ENVIRONMENT

The objective of this project was to develop a high-accuracy, scalable, and language-agnostic model for dyslexia screening. The primary data source for fine-tuning was the ETDD70 dataset, a rich collection of eye-tracking data from 70 Czech children (dyslexic and non-dyslexic) performing three specific reading tasks.

The development environment was Google Colab, leveraging GPU acceleration (Tesla T4) for all model training phases.

Core Libraries:

- **Data Ingestion:** `pymovements` (for initial dataset loading).
- **Data Manipulation:** `pandas` , `numpy` .
- **Modeling:** `tensorflow` (Keras), `scikit-learn` .

This document summarizes the iterative development process and provides a definitive guide for the engineering team tasked with deploying this model in a real-world screening application.

II. CORE COMPONENTS & DEPLOYMENT ASSETS

To run the model, the backend system requires **three** essential files, which represent the culmination of this project's work.

1. `gaze_encoder_pretrained.h5` **(The "Foundation")**
    - **What it is:** A foundational LSTM-based autoencoder that was pre-trained on the massive, 150-hour OneStop dataset.
    - **Its Role:** This model is **not called directly**. It is loaded *inside* the final `dyslexia_profile_model` and serves as a shared, frozen feature extractor. Its pre-training

on a self-supervised task (masked gaze modeling) taught it the universal, language-agnostic "grammar" of human eye movements (the physics of saccades, the rhythm of fixations, etc.). This transfer learning step is the primary reason for the model's high accuracy and data efficiency.

2. `scaler.pkl` **(The "Key")**
    - **What it is:** A `StandardScaler` object from Scikit-learn, saved as a pickle file.
    - **CRITICAL IMPORTANCE:** This is the most vital asset for preprocessing. The `gaze_encoder` was pre-trained on data normalized with this *exact* scaler. All new, real-world data from a Tobii tracker **must** be transformed with this same scaler to match the numerical range and distribution the model expects. **Failure to use this scaler will result in 100% incorrect and meaningless predictions.**
3. `dyslexia_profile_model.h5` **(The "Brain")**
    - **What it is:** The final, trained Keras model. It contains the complete multi-input architecture and all the learned weights for both the frozen encoder and the final, fine-tuned classifier head. This is the single file you will call `model.predict()` .

III. MODEL ARCHITECTURE & RATIONALE

The final model architecture was the result of a rigorous, iterative process.

A. Architectural Rationale: Why the "Reading Profile" Model?

During development, we explored simpler architectures and conclusively rejected them.

1. **Separate Models (Rejected):** Training one model for `Syllables` , one for `MeaningfulText` , etc., was rejected. This approach fragments an already small dataset (70 participants), leading to high overfitting risk. More importantly, it **loses the most powerful diagnostic signal**: the *comparative difference* in reading behavior *between* these tasks.
2. **Task-as-a-Feature (Rejected):** A single model that takes a 20-fixation sequence and is simply *told* the task type (e.g., via a one-hot feature) was also deemed insufficient. This signal is too weak and forces the model to learn the complex comparative logic only implicitly.

The "Reading Profile" Model (Our Champion): We designed a multi-input model that takes all three task-specific sequence sets from a single participant as three parallel inputs. These inputs are processed by the *same shared encoder*. The resulting profile vectors are then concatenated and fed to a final classifier head. This architecture **explicitly learns from the comparative differences** between the three tasks, which mimics a clinical diagnosis and ultimately yielded the best performance.

B. Final Model Architecture

The model was built using the Keras Functional API. The data flow is as follows (see `model.summary()`):

1. **Three Inputs:** The model has three separate inputs, one for each task. Each input has a shape of `(None, 100, 20, 5)` .
    - `input_syllables`
    - `input_meaningful`
    - `input_pseudo` *(See Section V for an explanation of these dimensions)*

2. **Shared Encoder:** The `gaze_encoder_pretrained.h5` model is loaded and frozen (`trainable=False`). It is then wrapped in a `TimeDistributed` layer. This *same* layer (`shared_gaze_encoder`) is applied to all three inputs. It processes each of the 100 `(20, 5)` sequences, outputting a 64-dimensional embedding for each.
   - *Output Shape per Branch:* `(None, 100, 64)`
3. **Profile Aggregation:** A `GlobalAveragePooling1D` layer is applied to each of the three branches. This aggregates the 100 sequence-embeddings into a single, 64-dimensional "profile vector" for each task, representing the average reading behavior for that task.
   - *Output Shape per Branch:* `(None, 64)`
4. **Concatenation:** The three 64-dim profile vectors are concatenated into a single, 192-dimensional vector (`concatenated_profile`). This vector represents the participant's complete, comparative reading profile.
   - *Output Shape:* `(None, 192)`
5. **Classifier Head:** This final profile vector is fed through a small, trainable classification head to produce the final prediction.
   - `Dense(64, activation='relu')`
   - `Dropout(0.5)`
   - `Dense(1, activation='sigmoid')` (The final prediction)

C. Final Performance

The model was initially trained for ~50 epochs, achieving 78% accuracy. After further training for **250 epochs**, the model converged to a superior performance, achieving a final **validation accuracy of 0.89** and an F1-score of **0.88**.

IV. DEVELOPMENT JOURNEY & DEBUGGING

The path to this model involved solving several critical bugs. Understanding these is key to understanding the preprocessing pipeline.

- **The `KeyError` Bug:** The `_fixations.csv` files from `pymovements` had hidden leading/trailing whitespace in their column names (e.g., `' fix_y'`). Our code was failing with `KeyError: 'fix_y'`.
  - **Solution:** We added `.columns = df.columns.str.strip()` after loading any CSV to programmatically clean the column names.
- **The "Packet Sizing Hell" (Inhomogeneous Shape `ValueError`):** When building the final dataset, we had 70 participant "packets." Each packet contained three arrays of sequences, e.g., `(18, 20, 5)`, `(21, 20, 5)`, and `(35, 20, 5)`. Calling `np.array()` on this list of 70 packets failed because the first dimension was not uniform.
  - **Solution:** We had to find the `max_len` (max sequences for any task, which was **100**) and manually "pad" each participant's sequence array. The final pipeline creates a `np.zeros((100, 20, 5))` container and copies the real sequences (e.g., the 35) into it: `padded_array[:35] = sequences`. This ensures every tensor in the final batch has the identical shape of `(70, 100, 20, 5)`.

- **The `scaler.pkl` Logic Error:** A subtle bug was found and fixed. The `scaler.pkl` was trained on features that were *already* spatially normalized. The pipeline must first convert pixels to a `[0, 1]` scale and *then* apply the `scaler.pkl`. This two-stage normalization is non-obvious and mandatory.

V. DEPLOYMENT: REAL-TIME INFERENCE PIPELINE

This is the **step-by-step guide for the backend team** to implement the inference logic. The system must replicate this process *exactly*.

System Requirement: A high-quality eye-tracker (e.g., Tobii 4C/5L). The application GUI **must** enforce a screen resolution of **1680x1050** to match the training data's coordinate system.

1. On Server Start:

- Load `dyslexia_profile_model.h5` into memory: `model = load_model(...)`.
- Load `scaler.pkl` into memory: `scaler = pickle.load(...)`.

2. During Screening Session (For one participant):

- Initialize three empty lists: `syl_sequences = []`, `mean_sequences = []`, `pse_sequences = []`.
- **Run Task 1 (Syllables):**
  - Stream gaze data `(x, y, timestamp)` from the Tobii.
  - Run a fixation detection algorithm (e.g., I-VT) on the stream to get a list of fixation events.
  - This list becomes a DataFrame. Run this DataFrame through the **Data Processing Pipeline (Steps A-E)**:
    - **A. Calculate 5 Core Features:** `fixation_duration` (from timestamps), `fixation_x`, `fixation_y`, `saccade_amplitude_in` (from pixel distance), `saccade_velocity_in` (from ampl / time).
    - **B. Clean:** Filter fixations (duration 80-1000ms).
    - **C. Spatial Normalization:**
      - `x_norm = fixation_x / 1680.0`
      - `y_norm = fixation_y / 1050.0`
      - `amp_norm = saccade_amplitude_in / 1960.53` (the diagonal)
    - **D. Feature Scaling (The "Key"):**
      - Create feature matrix in this exact order: `['fixation_duration', 'x_norm', 'y_norm', 'amp_norm', 'saccade_velocity_in']`.
      - `scaled_features = scaler.transform(feature_matrix)`.
    - **E. Sequencing (Sliding Window):**
      - Iterate through `scaled_features` with `SEQUENCE_LENGTH=20` and `STEP=5`.
      - Store all resulting `(20, 5)` sequences in the `syl_sequences` list.
- **Run Task 2 (MeaningfulText):** Repeat the entire process (A-E), storing results in `mean_sequences`.
- **Run Task 3 (PseudoText):** Repeat the entire process (A-E), storing results in `pse_sequences`.

3. On Session Complete (Making the Prediction):

- **Padding (The "Sizing Hell" Solution):**
    - The model expects an input of shape `(100, 20, 5)`.
    - `tensor_syl = manual_pad(syl_sequences, max_len=100)`
    - `tensor_mean = manual_pad(mean_sequences, max_len=100)`
    - `tensor_pse = manual_pad(pse_sequences, max_len=100)` *(The `manual_pad` function must create a `np.zeros((100, 20, 5))` and copy the sequences into it).*
- **Input Formatting:** The model expects a batch size of 1. The final input must be a dictionary:

```
X_input = {
    'input_syllables': np.array([tensor_syl]),
    'input_meaningful': np.array([tensor_mean]),
    'input_pseudo': np.array([tensor_pse])
}
```

- **Run Inference:** `prediction_probability = model.predict(X_input)[0][0]`