Forecasting Student Success through Artificial Intelligence: A Case Study Using Neural Networks

Dongli Liu[1], Wendy Paraizo[1], Daniel Ifejika[1]

[1] Engineering School of Centennial College. 941 Progress Ave, Scarborough, ON M1G 3T8 CA

Abstract

Accurately forecasting student success is crucial for improving retention and outcomes. This study predicts first-year persistence using Artificial Intelligence (AI) and Neural Networks (NN). A real-world dataset is processed to address challenges like missing values and class imbalance. The NN model, optimized with advanced techniques, outperforms traditional methods in predicting persistence. While the results show promise, the study also highlights challenges in model interpretability and generalization, suggesting areas for future exploration in AI-driven educational interventions.

Key words: Artificial Intelligence; Neural Network; Student Success; Predictive Analytics.

Introduction

Student success is a critical concern for educational institutions, influencing graduation rates, career outcomes, and societal contributions. However, predicting student success is complex, as factors such as socio-economic background, prior academic performance, and personal circumstances can significantly affect outcomes. Institutions face challenges in identifying students at risk of underperforming or dropping out, making it essential to develop methods for early intervention.

Programs like the Helping Youth Pursue Education (HYPE)[1][1], [2] initiative at Centennial College have demonstrated the importance of targeted support for students from underserved communities[3]. HYPE's focus on addressing barriers to post-secondary education for youth from disadvantaged backgrounds highlights the need for data-driven strategies to improve retention and success rates in higher education[1].

Artificial Intelligence (AI), particularly Neural Networks (NN), offers promising solutions in education. Neural networks can analyze large datasets, uncovering complex patterns that traditional methods may miss. This study uses AI to predict first-year persistence, a key indicator of student success, using a real-world dataset. The goal is to demonstrate how neural networks can improve prediction accuracy and provide insights for better supporting students, while addressing challenges such as missing data, class imbalance, and insufficient amount of data.

Methodology

Dataset Preprocessing

The dataset used in this study originates from Centennial College, specifically from engineer-

1 The HYPE program at Centennial College has given many young people new confidence to return to school by helping to overcome the economic and social barriers that may have interfered with school attendance in the past and by providing a nurturing, inclusive environment for youth aged 17-29, primarily living in the underserved neighbourhoods of Toronto.

ing school, and contains data on first-year students, including socio-economic status, high school GPA, attendance records, participation in extracurricular activities, and other features hypothesized to influence student success, as shown in Table 1.

Column	**Non Null**	**Dtype**
First Term Gpa	1420	float64
Second Term Gpa	1277	float64
First Language	1326	float64
Funding	1437	int64
School	1437	int64
Fast Track	1437	int64
Coop	1437	int64
Residency	1437	int64
Gender	1437	int64
Previous Education	1433	float64
Age Group	1433	float64
High School Avg Mark	694	float64
Math Score	975	float64
English Grade	1392	float64
FirstYearPersistence	1437	int64

Table 1: Information of the Raw Data

Missing values were handled by applying `dropna` columns with a small portion of null values. For the remaining missing data, `IterativeImputer` with `RandomForestRegressor()` was used to infer values, preserving feature relationships. This method was particularly important for columns like *High School Average Marks*, which had over 50% missing values, as using mean imputation could have introduced significant bias [4].

```
imputer = IterativeImputer(
    estimator=RandomForestRegressor(),
    max_iter=10,
    n_nearest_features=6,
    imputation_order="ascending",
    tol=1)
imputed = imputer.fit_transform(dropped)
```

Listing 1: `sklearn.impute.IterateImputer` is a experimental class that infers values by iteratively modelling a function with other features. It is very useful for *multivariate* problems. [4]

The data is significantly imbalanced, with 1138 positive while only 299 negative instances.To address **class imbalance**, *upsampling* was used instead of *downsampling* to retain valuable information [4]. This approach was also chosen because the dataset is relatively small, and maintaining as much data as possible is crucial for training an effective model.

Input Pipeline

The `tf.data.Dataset` API was utilized to create a descriptive and efficient input pipeline [5]. While the dataset used in this project is not large, the choice to use this API aligns with practical considerations, such as the potential deployment of the model for both *prediction* and *online machine training*[6][2]. By leveraging the capabilities of the API `tf.data.Dataset`, an efficient data processing chain was constructed following the approach detailed by A. Géron.

The dataset was split into three subsets: `train_set`, `val_set`, `test_set` using the `take()` and `skip()` functions provided by the API. The `train_set` comprises 70% of the data, while the `val_set` and `test_set` each consist of 15%.

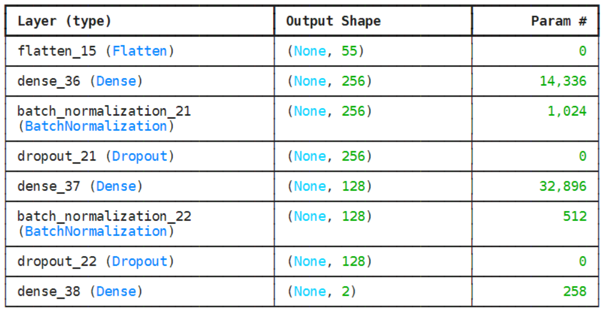
Neural Network Architecture

The neural network model used in this study is designed to predict *first-year persistence*. The architecture consists of an input layer, two hidden layers, and an output layer, with each

1 online machine training: a method of machine learning in which data becomes available in a sequential order and is used to update the best predictor for future data at each step.

hidden layer followed by a `BatchNormalization` layer and a `Dropout` layer for *normalization* and *regularization*. As shown in Figure 1.

The input layer is simply using a `Flatten()` to accept dataset. The hidden layers consist of 256 neurons in the first layer and 128 neurons in the second, chosen based on the experiments in a `for` loop. Each layer uses the *Rectified Linear Unit* (`relu`) activation function, which is known to handle the *vanishing gradient* problem better than traditional `sigmoid` functions. The output layer consists of 2 neuron, using the `softmax` activation function to adapt the `one-hot` format of dataset from the input pipeline, predicting the probability of student persistence. `Dropout` *regularization* techniques were applied in the hidden layers to prevent *overfitting*, with a dropout rate of 0.2. The model is trained using the *adam optimizer*, which combines the advantages of both *AdaGrad* and *RMSProp* for faster convergence and improved performance in *sparse gradients*.



Layer (type)	Output Shape	Param #
flatten_15 (Flatten)	(None, 55)	0
dense_36 (Dense)	(None, 256)	14,336
batch_normalization_21 (BatchNormalization)	(None, 256)	1,024
dropout_21 (Dropout)	(None, 256)	0
dense_37 (Dense)	(None, 128)	32,896
batch_normalization_22 (BatchNormalization)	(None, 128)	512
dropout_22 (Dropout)	(None, 128)	0
dense_38 (Dense)	(None, 2)	258

Figure 1: Architecture of the Selected Model

Model Training

The model was designed to train for 100 epochs, with `EarlyStopping` implemented to prevent *overfitting* if the validation loss does not improve after 10 epochs, and stopped the training at Epoch 45. A `ModelCheckpoint` was set to monitor the `val_accuracy` to capture the best weights.

```
history = model.fit(
  train_set.repeat(),
  epochs=100,
  validation_data=val_set.repeat(),
  steps_per_epoch=1000,
  validation_steps=100,
  callbacks=[mc, es])
```

Listing 2: Train model using infinite-length dataset with finite steps.

During training, the *learning curve*[3] was unstable due to the small dataset and the relatively complex model. To address this issue, the `repeat()` function was used to create an **infinite-length dataset**, allowing for extended training. The parameters `steps_per_epoch` and `validation_steps` were set accordingly to ensure the model received sufficient training despite the dataset's limited size. According to the `accuracy` and `loss` recorded in the training history, this approach significantly smoothed the *learning curve*. The model achieved high accuracy as well as better generalization capability compared to the regular training approach.

`accuracy`	0.9877
`val_accuracy`	1.0000
`loss`	0.0354
`val_loss`	0.0033

Table 2: Best accuracy and loss in training

From Table 2, When the train was early stopped at Epoch 45, the captured model reaches a 100% validation accuracy.

1 learning curve: refers to a graphical representation of the model's performance during training, typically showing metrics like accuracy and loss over training epochs

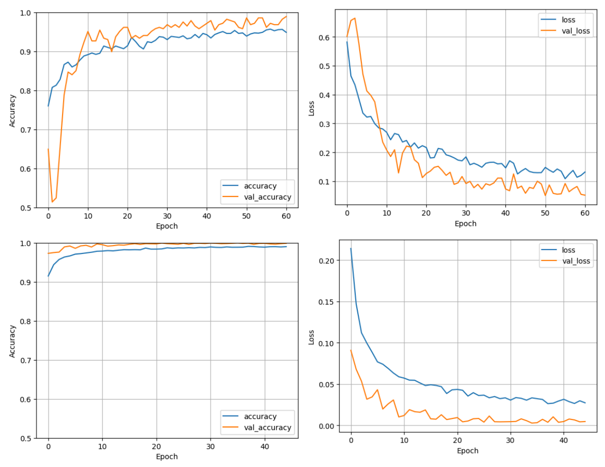


Figure 2: The learning curves show that using an infinite-length dataset (bottom figures) results in higher accuracy, lower loss, and greater stability, with a consistent gap between training and validation curves, indicating better generalization and reduced overfitting.

Evaluation Metrics

The performance of the neural network model was first evaluated by a straightforward `model.evaluate()` function to the unseen `test_set` and reached a astonishing accuracy.

dataset	1000 steps infinite test_set
`loss`	0.003268422558903694
`accuracy`	0.9997187256813049

Table 3: Evaluation with test_set. The repeat() was used again because the single `test_set` consistently produced 100% accuracy.

This outcome reflects the model's excellent performance on unseen test data. In this case, the *confusion matrix*(Figure 3) and *ROC AUC* seems unnecessary because they provide little additional insight when the model consistently predicts the correct class for all instances, resulting in perfect metrics.

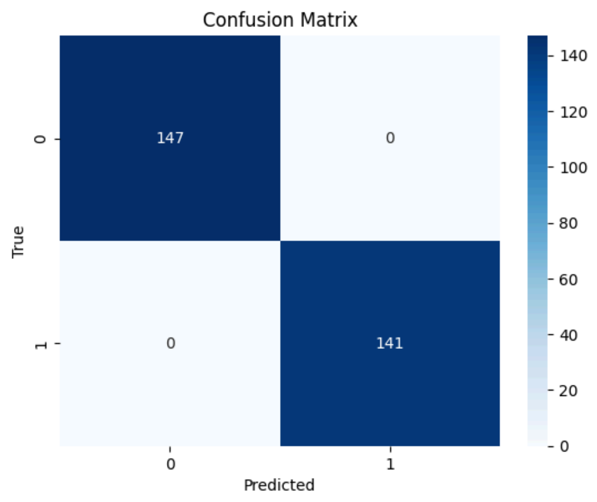


Figure 3: A perfect confusion matrix implies that the model correctly classified all instances, with no false positives or false negatives.

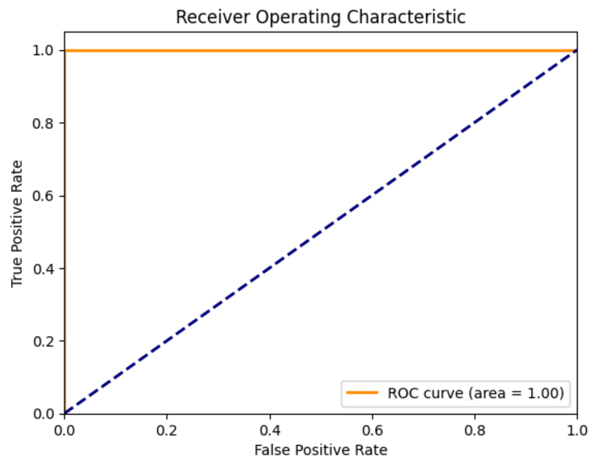


Figure 4: An AUC of 1.00 means that the model has perfect discrimination ability, correctly distinguishing between all positive and negative instances. It indicates that the model's predictions are flawless, with no false positives or false negatives, leading to a perfect classification performance across all thresholds.

Predictive Website

A full-stack website was built to utilize the model, featuring a Node.js frontend and Flask backend. The backend includes a simple `.csv` file as a database layer, used to append data during *online training*. While the website currently only supports prediction, the backend also developed to support two additional functionalities including train_one and train_batch. More information can be found in Appendix.

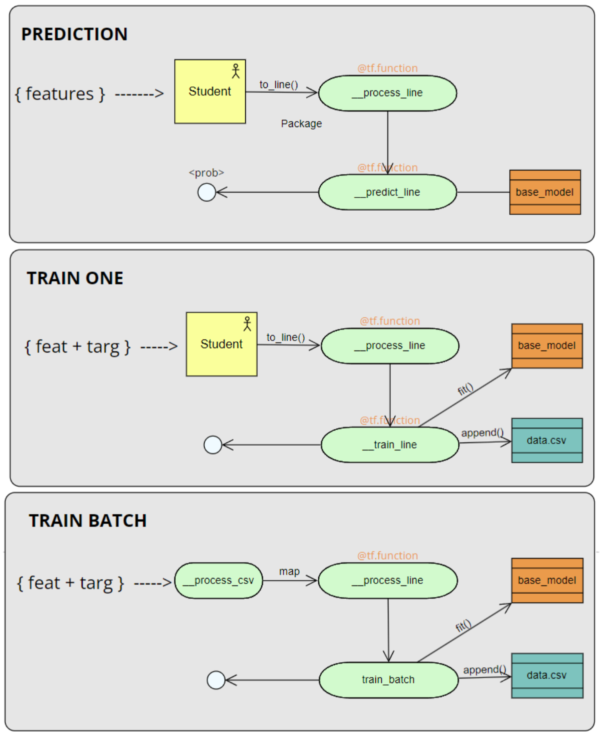


Figure 5: Diagrams showing the process of handling the 3 functionalities in backend. Multiple `@tf.function` was used to effectively handle the flow, matching the need of quick response. If the project have chance to be continued, *Tensorflow Serving* can be used to manage the model versions.

Discussion

Effectiveness in Prediction

The neural network model demonstrated impressive performance in predicting first-year student persistence, with an accuracy of 100% on the validation set and nearly perfect results on the test set. This aligns with the growing body of research suggesting that AI, particularly neural networks, can provide valuable insights into student success predictions, outperforming traditional statistical methods.

Model Interpretability

One of the key challenges highlighted by this study is the interpretability of neural network models. While the model provides high accuracy, understanding the specific factors driving its predictions remains complex. In educational settings, stakeholders such as instructors, counselors, and administrators may require more transparency in decision-making models.

Class Imbalance

Addressing class imbalance through upsampling proved to be effective in maintaining valuable data points, but it also raised questions about potential biases introduced by this technique. While upsampling prevented the loss of information, it might have artificially influenced the distribution of the data, potentially skewing the model's predictions.

Data Quality

Although the missing values were handled gracefully with `IterativeImputer` to preserve relationships between features, the importance of completeness of data is still worth to be emphasized. Missing data is often a recurring issue in real-world datasets, and the choice of imputation technique significantly impacts model accuracy. To handle missing values, other imputation methods such as *k-nearest neighbors* may have similar or better performance. The variance of the data source may also lead to a biased model, for example, all the instances of this dataset are from Engineering school. If the data is more variable, the model will have chance to learn better generalization skill.

Regularization

The model's *early stopping* and *dropout* regularization techniques contributed to minimizing overfitting, which is a common challenge when working with small datasets. Despite the model's high accuracy on unseen test_set, there may still be concerns regarding its generalization to other datasets with different characteristics. Further research is needed to test the model's robustness across a broader set of educational institutions and student demographics.

Model Complexity

Since we did not penalize the model's complexity during the design phase, the neural network evolved into a relatively complex structure. Although it is not overfitting, such complexity may increase computational costs and reduce interpretability. Future work could involve applying regularization, simplifying the architecture, or exploring model pruning to balance performance, efficiency, and scalability.

Model Tuning

Since the model achieved perfect performance without signs of *overfitting* after optimizing the network architecture, further tuning techniques were not explored in this project. However, future studies could consider strategies such as hyperparameter optimization methods like *Grid Search* and *Random Search*, applying regularization techniques like L1 and L2, and experimenting with diverse *units*, *activation* functions, and *loss* functions to enhance the model's robustness and generalizability.

Broader Generalization

Although the model shows excellent performance in this dataset, the effectiveness of applying the model to the broader use case, for example, other institutions or schools are uncertain. In that case, more evaluation should be performed, and more diverse data are probably required to further train the model.

Conclusion

The study reveals neural networks' powerful capability to predict first-year student persistence with near-perfect accuracy. Key findings include the model's strong predictive performance balanced against significant challenges such as interpretability limitations. The research addressed data preprocessing through advanced techniques like upsampling and iterative imputation, while implementing regularization methods such as dropout and early stopping to mitigate overfitting risks. While the model achieved perfect performance on this dataset, and should be helpful to the HYPE program[1], its effectiveness in broader scenarios remains unproven due to potential variations in student populations, institutional contexts, and data quality.

Critical insights emerge from the analysis: while AI can effectively forecast student success, educational stakeholders may require transparent models that enable clear, actionable decision-making. Future work should focus on expanding dataset diversity and advancing explainable AI techniques to bridge the gap between sophisticated predictive capabilities and meaningful educational interventions.

References

[1] P. Armstrong, H. Jafar, D. Aromiwura, J. Maher, A. Bertin, and H. Zhao, *Helping Youth Pursue Education (HYPE): Exploring the Keys to Transformation in Postsecondary Access and Retention for Youth from Underserved Neighbourhoods*. Toronto: Higher Education Quality Council of Ontario, 2017.

[2] C. College, "HYPE (Helping Youth Pursuing Education) Program." [Online]. Available: https://cet.qa.enginess.net/program/hype-helping-youth-pursue-education-program

[3] J. Maher and A. Bertin, "Sustaining the Transformation: Improving College Retention and Success Rates for Youth from Underserved Neighbourhoods," *Journal of*

Global Citizenship & Equity Education, vol. 3, no. 1, pp. 1–20, 2013, [Online]. Available: https://journals.sfu.ca/jgcee

[4] Scikit-Learn, “Scikit-Learn Documentation.” [Online]. Available: https://scikit-learn.org/stable/

[5] Google, “TensorFlow Tutorials and Related Google Documentation.” [Online]. Available: https://www.tensorflow.org/tutorials

[6] Wikipedia, “Online machine learning.” [Online]. Available: https://en.wikipedia.org/wiki/Online_machine_learning

[7] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 3rd ed. O'Reilly Media, Inc., 2022.

Appendix

A.1 Backend Guidelines

1. API Layer (`app.py`)

Endpoints:

- `/predict (POST)`: Accepts a `JSON` payload containing student features, converts the data into a Student object, validates it, and generates predictions, returning rounded predictions as a `JSON` response.
- `/train-one (POST)`: Accepts a single labeled data point for incremental training, enabling real-time model adaptability without requiring batch training.
- `/train-batch (POST)`: Accepts `CSV` files for bulk training, designed for periodic model updates using larger datasets.

2. Core Logic (`core.py`)

Model Management: Dynamically loads models based on the environment (development or production), ensures seamless switching between testing and deployment, and uses TensorFlow’s load_model for robust model handling.

Data Handling: Supports numerical and categorical data processing, where numerical features are normalized using predefined means and standard deviations, and categorical features are encoded using one-hot encoding for model compatibility. Incorporates a CSV-based training pipeline with optimized TensorFlow handling, using the tf.data API for efficient batch processing and multithreading, skipping headers and shuffling data to improve model generalization. Also validates data to ensure consistency and prevent model errors.

Prediction: Processes student features into a compatible input format, generates predictions with TensorFlow, and isolates probabilities for interpretability.

Training: Supports both incremental (train_one) and batch (train_batch) training, encapsulates preprocessing, data augmentation, and error handling during model fitting, and periodically saves the model after successful training.

Validation: Validates input data against metadata (e.g., ranges for numerical fields, allowed values for categorical fields) and prevents inconsistent or invalid inputs from corrupting the model.

3. Student Object

Structure: Encapsulates all student attributes, including academic and demographic features, and converts raw data into normalized, validated inputs for the model.

Validation: Checks numerical ranges and categorical mappings using metadata, ensuring the integrity of predictions and training by rejecting invalid data.

Conversion: Converts JSON payloads into Student objects for easy handling, and generates CSV strings for integration with TensorFlow pipelines.

A.2 Frontend Guidelines

1. `PredictionForm.js`

Renders a form to collect student data and submit predictions.Uses `React`'s `useState` hook for state management to track form inputs and results, implements client-side validation to ensure required fields are filled and numeric inputs are valid, shows a loading spinner and dynamic messages for predictions, integrates with the backend via `axios`, and is customizable with reusable components.

2. `PredictionResults.js`

Displays prediction results. Renders results only when available, and supports future expansions for more metrics.

3. `PredictionForm.css`

Styles the `PredictionForm.js` component. Leverages modern design trends with gradients, rounded borders, and responsive layout, highlights errors with distinct colors and styles, and provides hover effects on buttons for interactivity.