Project Report:
Classification with an Academic Success Dataset

ABOU ORM Daniel

December 24, 2024

1 Introduction to the AI Problem

- **Context:** Predicting whether a student will *Graduate* or *Dropout*.
- **Why AI:** Machine learning techniques allow us to train models that identify patterns in students' data to predict their future status.

2 Introduction to the Dataset

The dataset used in this project comes in two files: `train.csv` (with both features and a target column) and `test.csv` (same features but without the target).

2.1 Columns and Structure

Both files share the same general structure of columns, which capture a variety of demographic, academic, and behavioral characteristics for each student. Key columns include:

- **id**: Unique identifier for each student record.
- **Marital status**, **Application mode**, **Application order**, **Course**: Various indicators about the student's personal status and chosen study program.
- **Previous qualification** and **Previous qualification (grade)**: Information on the student's prior education.
- **Nacionality**, **Mother's qualification**, **Father's qualification**, **Mother's occupation**, **Father's occupation**: Family background and socioeconomic details.
- **Admission grade**: Numeric value reflecting admission performance.
- **Displaced**, **Educational special needs**, **Debtor**, **Tuition fees up to date**: Indicators of special conditions or administrative status.
- **Gender**, **Scholarship holder**, **Age at enrollment**, **International**: Additional personal/academic attributes.
- **Curricular units 1st sem & 2nd sem**: Several columns detailing how many curricular units were enrolled, evaluated, approved, credited, etc.
- **Unemployment rate**, **Inflation rate**, **GDP**: Economic context variables that might affect or correlate with academic outcomes.
- **Target** (*only in* `train.csv`): The label with values such as `Graduate`, `Dropout`, or `Enrolled`.

In train.csv, there are 38 columns in total (including Target), whereas test.csv has 37 columns (no Target). Each row corresponds to a single student's record. Below is a small sample excerpt:

- train.csv sample row (includes the Target):

```
id=0, Marital status=1, Application mode=1, ..., Unemployment rate=11.1, Inflation rate=0.6,
GDP=2.02, Target=Graduate
```

- test.csv sample row (no Target):

```
id=76518, Marital status=1, Application mode=1, ..., Unemployment rate=13.9, Inflation rate=-0.3,
GDP=0.79
```

2.2 Label Definition and Task

Since the goal is to predict whether a student eventually Graduates or Dropouts, the Target column in the training set is crucial. Note that some rows show the Target as Enrolled—these represent ongoing students but are still part of the classification challenge. The models are therefore trained to distinguish potential *graduates* from *dropouts* (and in some cases, Enrolled).

2.3 Data Usage in the Project

The use of train.csv is to train and evaluate my models, applying data preprocessing (e.g., handling missing values, normalizing numerical columns, and encoding categorical features). The final trained model is then applied to the test.csv dataset to generate predictions for Kaggle submission.

3 Methodology (Reference to Code)

This section describes all the steps taken, referencing the code that implements each step. Our methodology is split into five main parts: (1) Data Preprocessing, (2) Models Integrated, (3) Loss Function, (4) Optimization, and (5) Evaluation & Comparison.

3.1 Data Preprocessing

- **Techniques Used:**
  - Handling missing values via median (numeric) or Unknown (categorical).
  - One-hot encoding for categorical variables.
  - Normalizing numerical features.
  - Retaining a binary target column (Graduate = 1, Dropout = 0).
- **Reference in Code:** The main functions are preprocess_data and preprocess_and_get_input_size, which handle loading the CSV files, cleaning, and encoding.

3.2 Models Integrated

We employ two main models in our solution: a **Linear (Logistic Regression)** model and a **Neural Network** model. Both are defined in the code via the define_models function.

3.2.1 Linear (Logistic Regression) Model

The linear model is a straightforward approach to binary classification. In our project, it takes the form of:

$$\hat{y} = \sigma(\mathbf{w}^T\mathbf{x} + b),$$

where:

- $\mathbf{x}$ is the input feature vector (student's numerical/categorical data).
- $\mathbf{w}$ and b are the parameters learned during training.
- $\sigma(\cdot)$ is the sigmoid function, $\sigma(z) = \frac{1}{1+e^{-z}}$.

This model outputs a probability between 0 and 1, indicating the likelihood that a student is classified as Graduate (versus Dropout).

3.2.2 Neural Network Model

The neural network architecture in our project is designed to capture more complex patterns from the data. The specific architecture referenced in `define_models` includes:

- An **input layer** with `input_size` neurons (matching the number of features).
- One **hidden layer** (e.g., 32 neurons) with a ReLU activation function:

  $$\mathbf{h} = ReLU(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1),$$

  where $ReLU(z) = \max(0, z)$.
- An **output layer** with either
  - 1 neuron and a sigmoid (for binary output), or
  - 2 neurons (for `Dropout` vs. `Graduate`) and a softmax activation internally handled by the loss function.

This flexibility allows us to classify each student as a likely `Graduate` or `Dropout`. By stacking layers, the model can learn nonlinear relationships that may be missed by the simpler logistic regression approach. **Implementation Note:** Both models are defined using `PyTorch nn.Sequential` blocks. Refer to the Python code in `define_models` for the exact layer definitions and parameter initialization.

3.3 Loss Function

- **Logistic Regression:** Binary Cross-Entropy Loss (BCE).

  $$BCE(y, \hat{y}) = -\frac{1}{N}\sum_{i=1}^{N}\Big(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\Big)$$

- **Neural Network:** Cross-Entropy Loss for multi-class output layer.

Reference code: `define_loss_functions`.

3.4 Optimization Method

To train our models, we use built-in optimizers from the `torch.optim` package. In particular:

- `torch.optim.SGD`:
  - Implements *Stochastic Gradient Descent*, where parameters are updated based on the gradient of the loss function computed on mini-batches.
  - Each update is performed as:

    $$\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L},$$

    where θ denotes the model parameters, α is the learning rate, and $\nabla_\theta \mathcal{L}$ is the gradient of the loss $\mathcal{L}$.
  - Over multiple epochs (passes through the dataset), the parameters converge to values that (locally) minimize the loss.
- `torch.optim.Adam`:
  - An *adaptive* optimizer that extends SGD by maintaining moving averages of both gradients and their squares.
  - Adjusts the learning rate for each parameter individually based on estimates of first and second moments of the gradients. This mechanism often speeds up convergence and handles noisy/lower-scale gradients more effectively than SGD.

In our code:

- `define_optimizers` initializes `SGD` for the logistic model and `Adam` for the neural network.
- We also optionally use a **learning rate scheduler** (e.g., `StepLR`) to reduce the learning rate after a set number of epochs, helping fine-tune convergence over time.

3.5 Evaluation of Models / Comparison

To assess our models, we rely on multiple stages: (1) splitting our data into training and validation sets, (2) training and validating each model epoch by epoch, and (3) performing hyperparameter tuning where necessary.

3.5.1 Data Splitting for Evaluation

We use the function `split_and_prepare_data` to divide the preprocessed dataset into training and validation subsets. Internally, this function:

1. Separates the `Target` column (and also drops the `id`).
2. Shuffles and splits the indices according to a user-defined `split_ratio` (e.g., 80–20).
3. Converts these splits into PyTorch tensors, handling the difference between binary and multi-class targets (`float32` vs. `long`).
4. Wraps the subsets into `DataLoader` objects, which feed mini-batches to the model during training.

This ensures an efficient batching process and a clean separation of training vs. validation data.

3.5.2 Training and Validation Flow

The `train_and_validate` function manages the core training loop:

- **Forward Pass:** Computes predictions from the current model parameters.
- **Loss Computation:** Applies either BCE or Cross-Entropy loss depending on the model.
- **Backward Pass:** Calculates gradients and updates parameters via the chosen optimizer.
- **Validation Phase:** Periodically evaluates performance on the validation set by freezing model updates (i.e., `model.eval()`), calculating loss, and measuring accuracy.

We track training losses, validation losses, and validation accuracies at each epoch to visualize convergence and diagnose over/underfitting.

3.5.3 Metrics

Our primary metrics for comparison are:

- **Loss (training and validation):** Indicates how well the model's predictions fit the true labels.
- **Accuracy (validation):** Tracks the proportion of correct predictions to gauge overall performance.

3.5.4 Hyperparameter Tuning

We perform hyperparameter tuning via `hyperparameter_tuning`, which:

1. Takes the following arguments:
   - `train_data`: The training dataset.
   - `target_column`: The name of the target column in the dataset.
   - `model_type`: The type of model to be used ("logistic" or "neural_net").
   - `param_grid`: A dictionary of hyperparameters to test.
   - `epochs`: (Optional) The number of epochs for each configuration, defaulting to 5.
2. For each combination, builds the corresponding model (logistic or neural network) and trains it for a few epochs.
3. Records the validation loss for each run, then identifies and returns the combination that yields the lowest loss.

This systematic search helps us find a better learning rate or batch size for both the logistic regression and the neural network.

3.5.5 Comparison and Observations

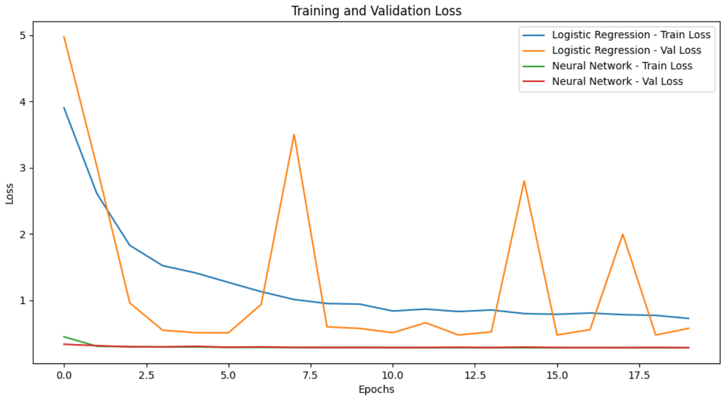


Figure 1: Training and Validation Loss for Logistic Regression vs. Neural Network

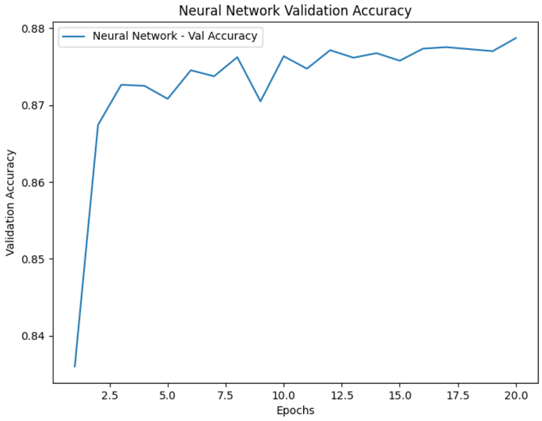


Figure 2: Neural Network Validation Accuracy Over 20 Epochs

Figure 1 shows how the training and validation loss evolve for both models across 20 epochs:

- **Logistic Regression:** The training loss (blue) decreases steadily below 1.0, but the validation loss (orange) fluctuates more drastically, sometimes spiking above 3.0. This inconsistency suggests potential underfitting or sensitivity to certain batches, as the validation data may differ in distribution or complexity.

- **Neural Network:** The training loss (green) remains very low from the start, and the validation loss (red) is almost flat. This indicates the network has quickly learned a stable representation of the data without large swings, suggesting better generalization.

Figure 2 focuses on the neural network's validation accuracy. Notable observations:

- The network starts at a lower accuracy (around 0.83) but climbs to approximately 0.88 by the end of 20 epochs.

- Minor fluctuations around 0.86–0.88 suggest the model is still improving but may be nearing a performance plateau.

Overall Comparison:

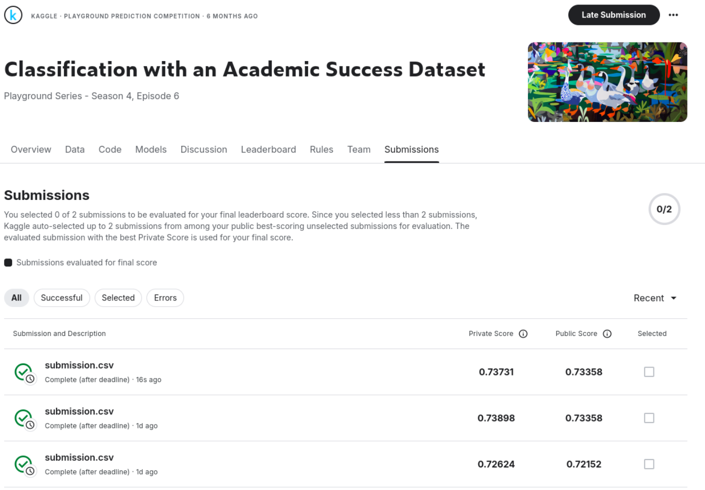
- The *logistic regression* model suffers from higher and more volatile validation loss, indicating it may be less capable of capturing the complexities of the data.

- The *neural network* achieves more stable loss curves and a higher validation accuracy, suggesting that it better generalizes to unseen data.

Based on these results, the neural network appears to be the stronger model. In the subsequent section, we use this model (with its best hyperparameters) to generate final predictions for Kaggle submission.

4 Kaggle Submission

- Generated a `submission.csv` file using `generate_submission` function.
- Achieved a certain score on Kaggle's private leaderboard.



KAGGLE · PLAYGROUND PREDICTION COMPETITION · 6 MONTHS AGO

Late Submission ···

Classification with an Academic Success Dataset

Playground Series - Season 4, Episode 6

Overview Data Code Models Discussion Leaderboard Rules Team Submissions

Submissions

You selected 0 of 2 submissions to be evaluated for your final leaderboard score. Since you selected less than 2 submissions, Kaggle auto-selected up to 2 submissions from among your public best-scoring unselected submissions for evaluation. The evaluated submission with the best Private Score is used for your final score.

0/2

■ Submissions evaluated for final score

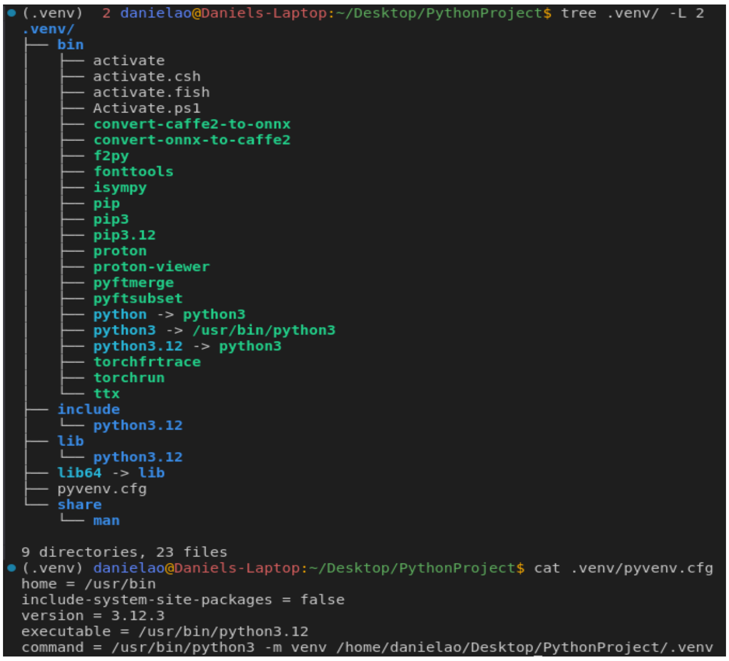
All Successful Selected Errors Recent ▾

Submission and Description	Private Score ⓘ	Public Score ⓘ	Selected
submission.csv Complete (after deadline) · 16s ago	0.73731	0.73358	□
submission.csv Complete (after deadline) · 1d ago	0.73898	0.73358	□
submission.csv Complete (after deadline) · 1d ago	0.72624	0.72152	□

Figure 3: Screenshot of successful submission on Kaggle

5 Launching the Code Successfully

- **Private Environment venv:** Python, libraries installed in venv (PyTorch, Pandas, pyTorch).



```
● (.venv)  2 danielao@Daniels-Laptop:~/Desktop/PythonProject$ tree .venv/ -L 2
.venv/
├── bin
│   ├── activate
│   ├── activate.csh
│   ├── activate.fish
│   ├── Activate.ps1
│   ├── convert-caffe2-to-onnx
│   ├── convert-onnx-to-caffe2
│   ├── f2py
│   ├── fonttools
│   ├── isympy
│   ├── pip
│   ├── pip3
│   ├── pip3.12
│   ├── proton
│   ├── proton-viewer
│   ├── pyftmerge
│   ├── pyftsubset
│   ├── python -> python3
│   ├── python3 -> /usr/bin/python3
│   ├── python3.12 -> python3
│   ├── torchfrtrace
│   ├── torchrun
│   └── ttx
├── include
│   └── python3.12
├── lib
│   └── python3.12
├── lib64 -> lib
├── pyvenv.cfg
└── share
    └── man

9 directories, 23 files
● (.venv) danielao@Daniels-Laptop:~/Desktop/PythonProject$ cat .venv/pyvenv.cfg
home = /usr/bin
include-system-site-packages = false
version = 3.12.3
executable = /usr/bin/python3.12
command = /usr/bin/python3 -m venv /home/danielao/Desktop/PythonProject/.venv
```

Figure 4: Caption

A Appendix: Code Function Signatures:

Here I will present the key function signatures and brief descriptions. See the accompanying Python script for full implementations.

A.1 `load_data`

Signature:

```python
def load_data(file_path):
```

Description:
Loads the dataset from a CSV file into a Pandas DataFrame.

A.2 `preprocess_data`

```python
def preprocess_data(data):
```

Handles data cleaning, encoding, normalization, and ensures a binary target column.

A.3 `preprocess_and_get_input_size`

```python
def preprocess_and_get_input_size(train_path, test_path):
```

Loads and preprocesses both training and test sets; returns processed DataFrames and the input size.

A.4 `define_models`

```python
def define_models(input_size):
```

Returns the logistic regression and neural network models.

A.5 `define_loss_functions`

```python
def define_loss_functions():
```

Specifies loss functions: BCE for logistic regression, cross-entropy for neural network.

A.6 `define_optimizers`

```python
def define_optimizers(logistic_model, neural_net, learning_rate=0.0005):
```

Sets up SGD for logistic model and Adam for the neural network.

A.7 `split_and_prepare_data`

```python
def split_and_prepare_data(data, target_column,
                           batch_size=32, split_ratio=0.2):
```

Splits data into training/validation and returns `DataLoader` objects.

A.8 train_and_validate

```
def train_and_validate(model, train_loader, val_loader,
                       loss_fn, optimizer, scheduler=None,
                       epochs=20):
```

Trains and evaluates the model, returns training/validation loss and validation accuracy.

A.9 hyperparameter_tuning

```
def hyperparameter_tuning(train_data, target_column,
                          model_type, param_grid, epochs=5):
```

Performs grid search over given hyperparameters and returns the best config.

A.10 generate_submission

```
def generate_submission(model, test_data,
                        file_name="submission.csv"):
```

Generates predictions for the test dataset and outputs a CSV for Kaggle.

```
if __name__ == "__main__":
    # 1) Preprocess Data
    train_data, test_data, input_size = preprocess_and_get_input_size(
        TRAIN_PATH, TEST_PATH
    )


    # 2) Define Models
    logistic_model, neural_net = define_models(input_size)


    # 3) Define Loss Functions
    logistic_loss, neural_net_loss = define_loss_functions()


    # 4) Define Optimizers and LR Scheduler
    logistic_optimizer, neural_net_optimizer = define_optimizers(
        logistic_model, neural_net, learning_rate=0.001
    )
    scheduler = torch.optim.lr_scheduler.StepLR(
        neural_net_optimizer, step_size=5, gamma=0.5
    )


    # 5) Split Data
    train_loader, val_loader = split_and_prepare_data(
        train_data, target_column="Target", batch_size=128
    )


    # 6) Train and Validate Logistic Regression
    logistic_train_losses, logistic_val_losses, logistic_val_accuracies = train_and_validate(
        logistic_model, train_loader, val_loader, logistic_loss, logistic_optimizer, epochs=20
    )


    # 7) Train and Validate Neural Network
    neural_net_train_losses, neural_net_val_losses, neural_net_val_accuracies = train_and_validate(
        neural_net, train_loader, val_loader, neural_net_loss,
        neural_net_optimizer, scheduler=scheduler, epochs=20
    )
```

```python
# 8) Plot Convergence and Accuracy
plt.figure(figsize=(12, 6))
plt.plot(logistic_train_losses, label="Logistic - Train Loss")
plt.plot(logistic_val_losses, label="Logistic - Val Loss")
plt.plot(neural_net_train_losses, label="Neural Net - Train Loss")
plt.plot(neural_net_val_losses, label="Neural Net - Val Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Training and Validation Loss")
plt.legend()
plt.show()



# Accuracy Figure
epochs = 20
plt.figure(figsize=(8, 6))
plt.plot(range(1, epochs+1), neural_net_val_accuracies,
         label="Neural Net - Val Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
plt.title("Neural Network Validation Accuracy")
plt.legend()
plt.show()

# 9) Generate Submission
generate_submission(neural_net, test_data, file_name="submission.csv")
```