# Machine Learning Models in Skilly Backend

This document provides a comprehensive explanation of the machine learning models implemented in the Skilly backend system. The system contains two different models for career role prediction, each with its own implementation and characteristics.

## Introduction

The Skilly backend implements two neural network models for career role prediction. These models analyze user input data to predict the most suitable career role based on various features. The models differ in their complexity, preprocessing steps, and architectural design.

## Model 1: Advanced Neural Network Model

### Overview

Model 1 is implemented in `src/script/model-1/model.py` and represents a sophisticated neural network architecture designed for high accuracy and robust performance. This model employs advanced techniques for handling class imbalance and complex data preprocessing.

### Data Preprocessing Pipeline

*1. Data Loading*

```
data = pd.read_csv('../career-mapping.csv')
```

- Loads the career mapping dataset from CSV
- The dataset contains features and target variables
- Each row represents a career profile with associated features

*2. Feature Extraction*

```
X = data.iloc[:, :-1].values  # Features
y = data.iloc[:, -1].values   # Target
```

- Extracts all columns except the last one as features (X)
- Uses the last column as the target variable (y)
- Converts data to numpy arrays for efficient processing

*3. Label Encoding*

```
label_encoder = LabelEncoder()
y = label_encoder.fit_transform(y)
```

- Converts categorical target labels to numerical values
- Creates a mapping between original labels and encoded values
- Essential for neural network processing which requires numerical inputs

*4. Feature Standardization*

```
scaler = StandardScaler()
X = scaler.fit_transform(X)
```

- Standardizes features to have zero mean and unit variance
- Formula: $z = (x - \mu) / \sigma$
  - x: original feature value
  - μ: mean of the feature
  - σ: standard deviation of the feature
- Improves model convergence and performance

**Model Architecture**

*1. Input Layer*

```
tf.keras.layers.Dense(128, activation='relu', input_shape=(X_train.shape[1],))
```

- Takes input features with shape matching the training data
- 128 neurons in the first layer
- ReLU activation function: $f(x) = \max(0, x)$
- Purpose: Initial feature transformation and dimensionality expansion

*2. Batch Normalization*

```
tf.keras.layers.BatchNormalization()
```

- Normalizes the activations of the previous layer
- Reduces internal covariate shift
- Improves training stability and speed
- Formula: $y = \gamma * (x - \mu) / \sqrt{(\sigma^2 + \varepsilon)} + \beta$
  - γ, β: learnable parameters
  - ε: small constant for numerical stability

*3. Dropout Layer*

```
tf.keras.layers.Dropout(0.3)
```

- Randomly drops 30% of neurons during training
- Prevents overfitting by reducing co-adaptation
- Acts as a regularization technique
- Only active during training, not during inference

*4. Hidden Layers*

```
tf.keras.layers.Dense(64, activation='relu')
tf.keras.layers.Dense(32, activation='relu')
```

- First hidden layer: 64 neurons
- Second hidden layer: 32 neurons
- Both use ReLU activation
- Progressive dimensionality reduction

*5. Output Layer*

```
tf.keras.layers.Dense(len(np.unique(y)), activation='softmax')
```

- Number of neurons equals number of unique classes
- Softmax activation for multi-class classification
- Outputs probability distribution over classes
- Formula: softmax(x_i) = exp(x_i) / Σ exp(x_j)

**Training Process**

*1. Data Splitting*

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y)
```

- 80% training data, 20% testing data
- Stratified split to maintain class distribution
- Random state for reproducibility

*2. Class Weight Calculation*

```
class_weights = class_weight.compute_class_weight('balanced', classes=np.unique(y), y=y_train)
```

- Handles class imbalance
- Weights inversely proportional to class frequencies
- Formula: weight = n_samples / (n_classes * n_samples_for_class)

*3. Model Compilation*

```
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

- Adam optimizer: Adaptive learning rate optimization
- Categorical cross-entropy loss for multi-class classification
- Accuracy metric for performance evaluation

*4. Training*

```
model.fit(
    X_train, y_train,
    epochs=50,
    batch_size=32,
    validation_data=(X_test, y_test),
    class_weight=class_weights
)
```

- 50 training epochs
- Batch size of 32 samples
- Validation on test set
- Class weights applied to handle imbalance

**Prediction Process**

*1. Input Processing*

```
input_data_scaled = scaler.transform(input_data)
```

- Scales input features using pre-fitted scaler
- Maintains consistency with training data preprocessing

*2. Model Loading*

```
loaded_model = tf.keras.models.load_model('./src/script/model-1/modified_model.h5')
```

- Loads trained model from saved file
- Preserves architecture and weights

*3. Prediction*

```
prediction = loaded_model.predict(input_data_scaled)
predicted_label = label_encoder.inverse_transform([np.argmax(prediction)])[0]
```

- Generates probability distribution over classes

- Selects class with highest probability
- Converts numerical prediction back to original label

## Model 2: Simplified Neural Network Model

### Overview

Model 2 is implemented in src/script/model-2/train-model.py and represents a more streamlined approach to career role prediction. This model focuses on efficiency and simplicity while maintaining reasonable accuracy.

### Data Preprocessing Pipeline

*1. Data Loading and Feature Selection*

```
df = pd.read_csv("../career-mapping.csv")
X = df.iloc[:, :27]  # First 27 columns as features
y = df.iloc[:, -1]   # Last column as target
```

- Loads dataset and selects specific features
- Uses first 27 columns as input features
- Last column as target variable

*2. Missing Value Handling*

```
X = X.fillna(0)
```

- Fills missing values with zeros
- Simple but effective approach for this use case

*3. Target Encoding*

```
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)
y_categorical = tf.keras.utils.to_categorical(y_encoded)
```

- Converts labels to numerical values
- Transforms to one-hot encoded format
- Required for categorical cross-entropy loss

### Model Architecture

*1. Input Layer*

```
tf.keras.layers.Dense(64, activation='relu', input_shape=(27,))
```

- 64 neurons
- ReLU activation

- Fixed input shape of 27 features

*2. Hidden Layer*

```
tf.keras.layers.Dense(64, activation='relu')
```

- 64 neurons
- ReLU activation
- Maintains dimensionality

*3. Output Layer*

```
tf.keras.layers.Dense(y_categorical.shape[1], activation='softmax')
```

- Number of neurons matches number of classes
- Softmax activation for probability distribution

**Training Process**

*1. Model Compilation*

```
model.compile(
    optimizer='adam',
    loss='categorical_crossentropy',
    metrics=['accuracy']
)
```

- Adam optimizer
- Categorical cross-entropy loss
- Accuracy metric

*2. Training*

```
model.fit(
    X_train, y_train,
    epochs=50,
    batch_size=32,
    validation_data=(X_test, y_test)
)
```

- 50 epochs
- Batch size 32
- Validation on test set

**Model Persistence**

*1. Model Saving*

```
model.save("career_model.h5")
```

- Saves model architecture and weights
- HDF5 format for efficient storage

*2. Preprocessing Artifacts*

```
joblib.dump(scaler, "scaler.pkl")
joblib.dump(label_encoder, "label_encoder.pkl")
```

- Saves scaler for feature standardization
- Saves label encoder for label conversion
- Enables consistent preprocessing during inference

## Integration with Backend

### Controller Implementation

```
async function runPythonScript(inputData) {
  return new Promise((resolve, reject) => {
    const process = spawn("python3", ["./src/script/model-1/test-model.py",
JSON.stringify(inputData)]);
    // ... process handling
  });
}
```

- Spawns Python process for model execution
- Handles input/output communication
- Manages process lifecycle
- Error handling and response formatting

### Data Flow

1. API receives request with user data
2. Controller formats data for model input
3. Python process executes model prediction
4. Results returned to API
5. Response formatted and sent to client

## Model Comparison

### Architecture

1. **Model 1**:
   - Complex architecture with multiple layers
   - Batch normalization for stability
   - Progressive dimensionality reduction

    - More parameters to learn
2. **Model 2**:
    - Simpler architecture
    - Fewer layers and parameters
    - Focus on computational efficiency
    - Faster training and inference

### Feature Handling

1. **Model 1**:
    - Uses all available features
    - Sophisticated preprocessing
    - Class weight balancing
    - More robust to data variations
2. **Model 2**:
    - Fixed 27 features
    - Basic preprocessing
    - Simpler implementation
    - Faster processing

### Performance Characteristics

1. **Model 1**:
    - Higher accuracy potential
    - More computational resources required
    - Better handling of complex patterns
    - Slower training and inference
2. **Model 2**:
    - Faster execution
    - Lower resource requirements
    - Suitable for real-time applications
    - More efficient deployment

## Technical Details

### Dependencies

- TensorFlow: Deep learning framework
- scikit-learn: Machine learning utilities
- pandas: Data manipulation

- numpy: Numerical computing
- joblib: Model persistence

**System Requirements**

1. Python 3.x
2. Sufficient RAM for model loading
3. GPU optional but recommended for Model 1
4. Disk space for model artifacts