Implementing Logistic Regression and Softmax Models with Keras in TensorFlow 2

A practical approach to binary and multiclass classification using neural networks in Python.

Hernández Pérez Erick Fernando

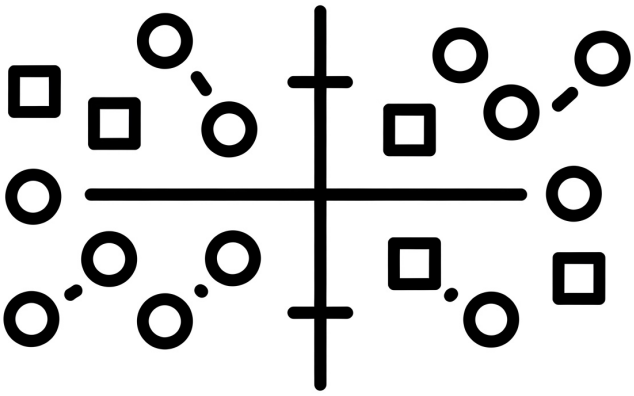

Figure 1

In the previous two posts—Introduction to Logistic Regression: Fundamentals and Applications (Post 8) and Multiclass Logistic Regression with TensorFlow 2: Implementation and Applications (Post 9)—we explored the implementation of logistic regression and softmax regression from a theoretical perspective. Our primary goal was to understand the fundamental steps required for these models to learn, using TensorFlow 2 to illustrate each process in detail.

However, as we deepen our understanding of these concepts, it becomes evident that a purely theoretical approach has its limitations. To bridge this gap, this post aims to take our exploration a step further by shifting towards a more practical implementation. Specifically, we will revisit two previously explored codes and adapt them to a new, more applied framework using Keras. While Keras introduces concepts commonly associated with neural networks, the transition will be smooth, as it builds upon the knowledge we have already developed.

In this post, we will present the updated code along with a brief analysis of the new functionalities that Keras brings to the table. Additionally, we will introduce a new implementation of logistic regression known as *2D logistic regression*, named for its intuitive graphical representation. This will allow us to visualize how the model separates different classes in a two-dimensional space, reinforcing our understanding of its decision boundaries.

By the end of this post, you will not only have a stronger grasp of logistic regression from both theoretical and practical perspectives but also gain insights into how modern deep learning frameworks like Keras can streamline implementation.

Logistic Regression 1D

In the case of **1D logistic regression**, we consider a single input feature x, making it the simplest form of logistic regression. The model follows the equation:

$$h(x) = \frac{1}{1 + e^{-(wx+b)}}$$

where w is the weight (slope) of the model, b is the bias (intercept), and $h(x)$ represents the predicted probability of belonging to the positive class. The function $h(x)$ is known as the *sigmoid function*, which maps any real number to a range between 0 and 1, ensuring that the output can be interpreted as a probability.

The decision boundary in 1D logistic regression is determined by solving $h(x) = 0.5$, which corresponds to the point where the model is equally uncertain about the classification. This means:

$$x = -\frac{b}{w}$$

which represents the threshold at which the model transitions between predicting one class or the other.

1D Logistic Regression

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Data generation
x_label0 = np.random.normal(-4, 2, 1000)
x_label1 = np.random.normal(4, 2, 1000)
xs = np.append(x_label0, x_label1).astype(np.float32)
labels = np.array([0.] * len(x_label0) + [1.] * len(x_label1), dtype=np.float32)

# Plot the data
plt.scatter(xs, labels, alpha=0.3)

# Create the model with Keras
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_shape=(1,), activation='sigmoid')
])

# Compile the model
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.01),
              loss=tf.keras.losses.BinaryCrossentropy(),
              metrics=['accuracy'])

# Train the model
model.fit(xs, labels, epochs=30, verbose=0, batch_size=32)

# Get the trained parameters (bias and weight)
w0, w1 = model.layers[0].get_weights()
print(f"Trained parameters: w0 = {w0[0][0]}, w1 = {w1[0]}")

# Plot the fitted sigmoid
all_xs = np.linspace(-10, 10, 100)
all_xs = all_xs.reshape(-1, 1)  # Ensure all_xs has shape (N, 1)
preds = model.predict(all_xs)   # Make a single prediction outside the plot
plt.plot(all_xs, preds, label="Sigmoid Model", color='red')
plt.scatter(xs, labels, alpha=0.3, label="Data")
plt.legend()
plt.show()
```

Creating the model with Keras

Once the data has been generated, the next step is to define the model using `tf.keras.Sequential()`, which allows us to create a neural network sequentially (layer by layer).

```
model = tf.keras.Sequential([
tf.keras.layers.Dense(1, input_shape=(1,), activation='sigmoid')
])
```

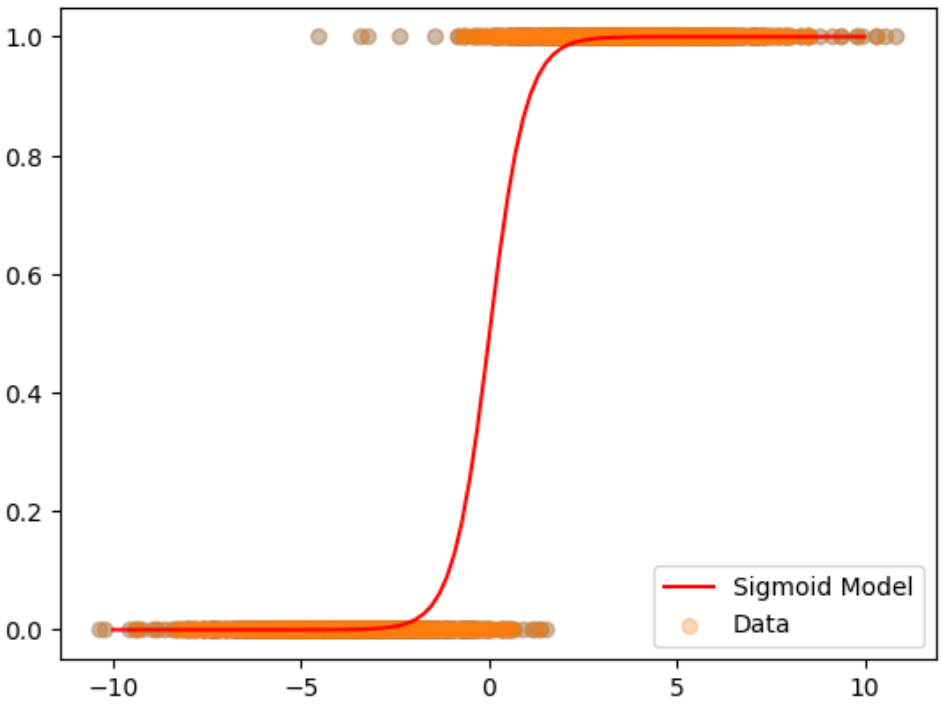

Figure 2: Our 1D Logistic regression

Here we define an extremely simple architecture with a single dense layer (`Dense`). Let's analyze each of the elements of this layer:

- **`tf.keras.layers.Dense(1)`**:
  - Define a fully connected (dense) layer.
  - The number 1 indicates that the layer has only one neuron, since we are solving a binary classification problem with only one output (probability of belonging to class 1).
- **`input_shape=(1,)`**:
  - Specifies that the input to the model is a single numeric value (`x`), since we are working with one-dimensional data.
  - This parameter is only needed in the first layer of the model so that Keras can build the architecture correctly.
- **`activation='sigmoid'`**:
  - Uses the sigmoid function as activation, which is defined as:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

    where $z = w \cdot x + b$ is the linear combination of the weights and the input.
  - This function transforms any real number into a value between 0 and 1, making it ideal for binary classification problems.

In summary, this model is equivalent to logistic regression, where the model learns a weight w and a bias b to predict the probability that a data item belongs to class 1.

Building the model

After defining the architecture of the model, it is necessary to compile it to specify how it will be trained.

```
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.01),
loss=tf.keras.losses.BinaryCrossentropy(),
metrics=['accuracy'])
```

Three fundamental elements are established here:

- **Optimizer (Adam)**:
  - Adam (Adaptive Moment Estimation) is used, an optimizer that combines the advantages of RMSprop and Momentum for efficient adjustment of weights.
  - A learning rate (`learning_rate=0.01`) is specified, indicating how quickly the weights are updated at each iteration.
- **Loss function (BinaryCrossentropy)**:
  - Since this is a binary classification problem, binary cross entropy is used as the error metric.
  - This function measures the difference between predicted probabilities and actual values, penalizing incorrect predictions in a logarithmic manner.
- **Evaluation metric (accuracy)**:
  - Precision (`accuracy`) is measured, which calculates the percentage of correct predictions.

Model training

Once compiled, the model is trained using the `.fit()` method.

```
model.fit(xs, labels, epochs=30, verbose=0, batch_size=32)
```

Important parameters:

- `xs, labels`:
  - The input data `xs` and the labels `labels` generated above are passed.
- `epochs=30`:
  - The model will iterate over the dataset 30 times, adjusting its weights at each iteration.
  - A larger number of epochs allows for better learning, but can also lead to overfitting.
- `verbose=0`:
  - Do not display training progress information in the console.
  - If you want to see the progress of the training, you can use `verbose=1`.
- `batch_size=32`:
  - The data is divided into batches of 32 examples for each weight update.
  - Using batches instead of processing all the data at once improves training efficiency and stability.

The goal of training is to find the optimal values of the parameters (w and b) that minimize the loss function (`BinaryCrossentropy`), allowing the model to fit the data correctly.

2D Logistic Regression

2D Logistic Regression

```python
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

# Hyperparameters
lr = 0.1
training_epochs = 50

# Data generation
x1_label1 = np.random.normal(3, 1, 1000)
x2_label1 = np.random.normal(2, 1, 1000)
x1_label2 = np.random.normal(7, 1, 1000)
x2_label2 = np.random.normal(6, 1, 1000)

x1s = np.append(x1_label1, x1_label2)
```

```
x2s = np.append(x2_label1, x2_label2)
ys = np.asarray([0.] * len(x1_label1) + [1.] * len(x1_label2))

# Reshape the data to have shape (N, 2)
X = np.column_stack((x1s, x2s))
y = ys.reshape(-1, 1)

# Define the model in Keras
model = tf.keras.Sequential([
    tf.keras.layers.Dense(1, input_shape=(2,), activation='sigmoid')
])

# Compile the model
model.compile(optimizer=tf.keras.optimizers.SGD(learning_rate=lr),
              loss=tf.keras.losses.BinaryCrossentropy())

# Train the model
model.fit(X, y, epochs=training_epochs, verbose=0, batch_size=32)

# Get the trained weights
w, b = model.layers[0].get_weights()
print(f"Trained parameters: w1 = {w[0][0]}, w2 = {w[1][0]}, b = {b[0]}")

# Create a mesh to plot the decision boundary
x1_range = np.linspace(-2, 10, 200)
x2_range = np.linspace(-2, 10, 200)
X1_mesh, X2_mesh = np.meshgrid(x1_range, x2_range)
Z = model.predict(np.c_[X1_mesh.ravel(), X2_mesh.ravel()])
Z = Z.reshape(X1_mesh.shape)

# Plot the decision boundary
plt.figure(figsize=(8, 6))
plt.contourf(X1_mesh, X2_mesh, Z, levels=[0, 0.5, 1], alpha=0.4, cmap="coolwarm")

# Plot the points
plt.scatter(x1_label1, x2_label1, c='red', marker='x', s=20, label='Class 0')
plt.scatter(x1_label2, x2_label2, c='green', marker='1', s=20, label='Class 1')

# Plot settings
plt.xlabel("X1")
plt.ylabel("X2")
plt.title("Decision Boundary of the Logistic Model")
plt.legend()
plt.grid(True, linestyle="--", alpha=0.6)
plt.show()
```

For this experiment, we generated two sets of data with a normal distribution. Each set represents a different class, with feature values X_1 and X_2 distributed around different centers. These values are combined into an input matrix X with dimensions $(2000, 2)$, while the target variable y is defined as a binary vector, where the first half of the data belongs to class 0 and the second half to class 1.

The model is built using **Keras**, with a single **Dense** layer of **one neuron**, since this is a binary classification problem. The **sigmoid activation function** is used in this layer to transform the output into a probability between 0 and 1.

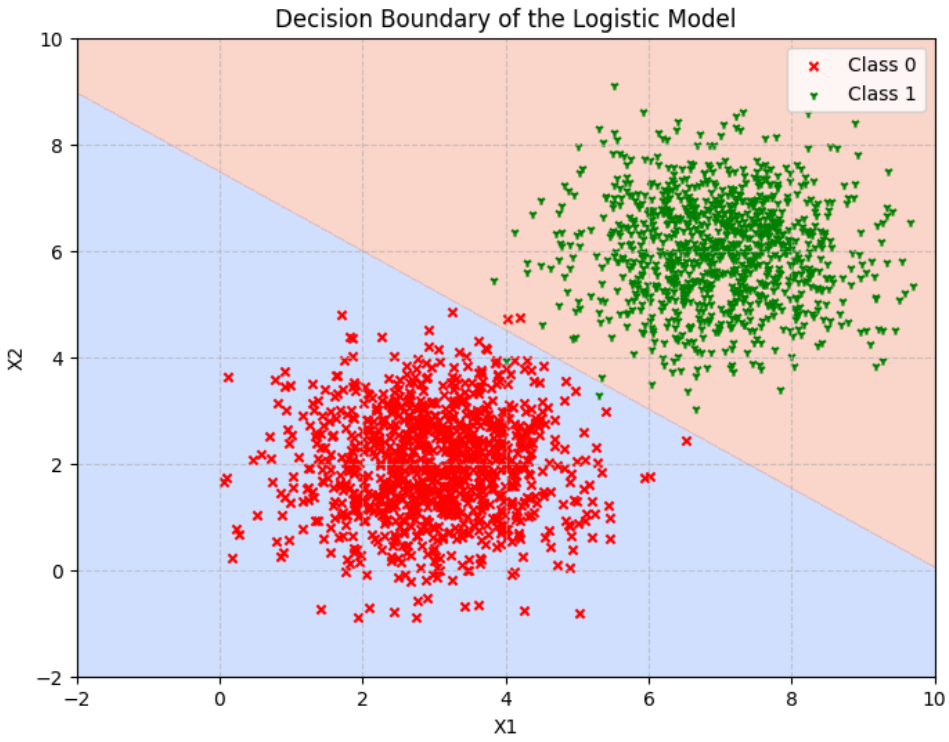

Figure 3: Our 2D Logistic regression

Softmax Regression

Softmax Regression

```python
import numpy as np
import matplotlib.pyplot as plt
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# Data generation
x1_label0 = np.random.normal(1, 1, (100, 1))
x2_label0 = np.random.normal(1, 1, (100, 1))
x1_label1 = np.random.normal(5, 1, (100, 1))
x2_label1 = np.random.normal(4, 1, (100, 1))
x1_label2 = np.random.normal(8, 1, (100, 1))
x2_label2 = np.random.normal(0, 1, (100, 1))

xs_label0 = np.hstack((x1_label0, x2_label0))
xs_label1 = np.hstack((x1_label1, x2_label1))
xs_label2 = np.hstack((x1_label2, x2_label2))
xs = np.vstack((xs_label0, xs_label1, xs_label2))

labels = np.array([[1., 0., 0.]] * 100 + [[0., 1., 0.]] * 100 + [[0., 0., 1.]] * 100)

# Shuffle data
arr = np.arange(xs.shape[0])
np.random.shuffle(arr)
xs, labels = xs[arr].astype(np.float32), labels[arr].astype(np.float32)
```

```python
# Test data
test_x1_label0 = np.random.normal(1, 1, (10, 1))
test_x2_label0 = np.random.normal(1, 1, (10, 1))
test_x1_label1 = np.random.normal(5, 1, (10, 1))
test_x2_label1 = np.random.normal(4, 1, (10, 1))
test_x1_label2 = np.random.normal(8, 1, (10, 1))
test_x2_label2 = np.random.normal(0, 1, (10, 1))

test_xs_label0 = np.hstack((test_x1_label0, test_x2_label0))
test_xs_label1 = np.hstack((test_x1_label1, test_x2_label1))
test_xs_label2 = np.hstack((test_x1_label2, test_x2_label2))

test_xs = np.vstack((test_xs_label0, test_xs_label1, test_xs_label2)).astype(np.float32)
test_labels = np.array([[1., 0., 0.]] * 10 + [[0., 1., 0.]] * 10 + [[0., 0., 1.]] *
↪  10).astype(np.float32)

# Parameters
lr = 0.01
training_epochs = 50
batch_size = 100
input_dim = xs.shape[1]
num_classes = labels.shape[1]

# Keras model
model = keras.Sequential([
    layers.Dense(num_classes, input_shape=(input_dim,), activation='softmax')
])

# Model compilation
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=lr),
    loss=keras.losses.CategoricalCrossentropy(),
    metrics=['accuracy']
)

# Training
model.fit(xs, labels, epochs=training_epochs, batch_size=batch_size, verbose=1)

# Evaluation
loss, accuracy = model.evaluate(test_xs, test_labels, verbose=0)
print(f"\nTest Accuracy: {accuracy:.4f}")

# Predictions
predictions = model.predict(test_xs)
print("\nFirst 5 predictions:\n", predictions[:5])

# Get the class with the highest probability
predicted_classes = np.argmax(predictions, axis=1)

# Visualization of predictions
plt.figure(figsize=(8, 6))

plt.scatter(test_xs[predicted_classes == 0, 0], test_xs[predicted_classes == 0, 1],
↪  c='r', marker='o', s=50, label="Class 0")
```

```
plt.scatter(test_xs[predicted_classes == 1, 0], test_xs[predicted_classes == 1, 1],
↪   c='g', marker='x', s=50, label="Class 1")
plt.scatter(test_xs[predicted_classes == 2, 0], test_xs[predicted_classes == 2, 1],
↪   c='b', marker='_', s=50, label="Class 2")

# Labels and legend
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.title('Model Predictions')
plt.legend()

# Show plot
plt.show()
```

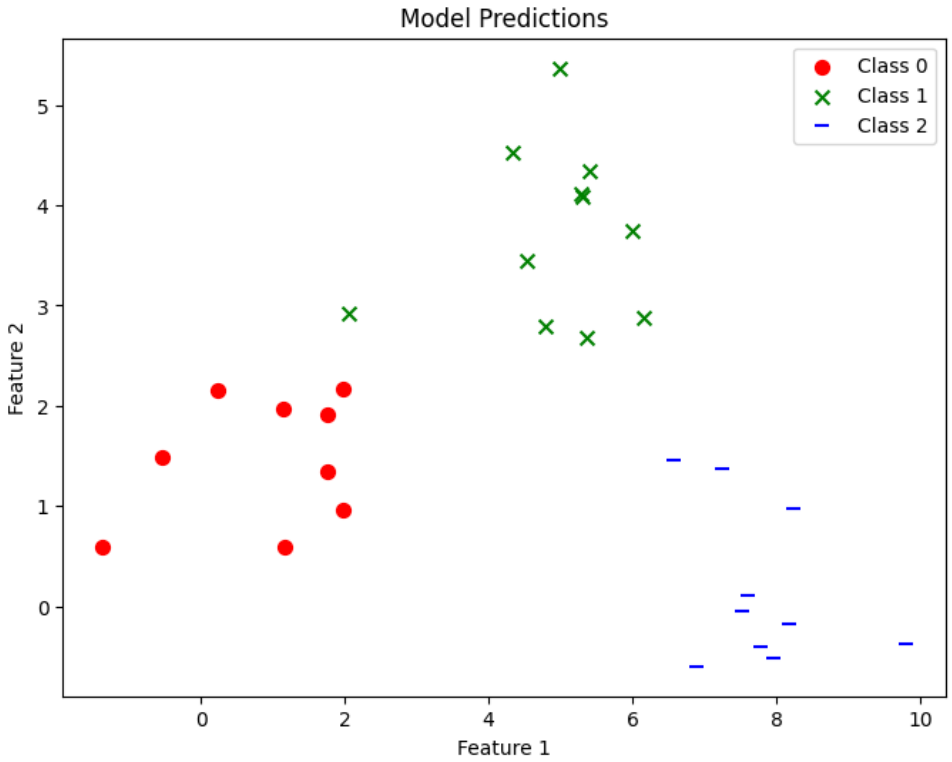

Figure 4: Our Softmax classification

The data is generated from three distinct clusters with a normal distribution. Each cluster represents a class and is composed of two features X_1 and X_2, with different means and variances. A total of 300 examples are created, with labels encoded in **one-hot** format:

- Class 0: X_1 and X_2 with mean at (1,1)
- Class 1: X_1 and X_2 with mean at (5,4)
- Class 2: X_1 and X_2 with mean at (8,0)

The data is combined into a matrix X of dimensions $(300, 2)$, and the labels into a matrix Y of dimensions $(300, 3)$.

To ensure proper training, the data is randomly shuffled before being used.

The neural network model is defined using the **Keras** sequential API. A single **dense layer** with three neurons is used, corresponding to the three output classes. The **softmax activation function** is used to convert the outputs into normalized probabilities.

The model is compiled with the following parameters:

- Optimizer: **Adam**, with learning rate of 0.01.
- Loss function: **Categorical Crossentropy**, suitable for multiclass classification problems with **one-hot** labels.
- Metric: **Accuracy**.

Training is performed for 50 **epochs**, with a batch size of **100 examples**.

After training, the model is evaluated with an independent test set, composed of 30 examples generated with the same distribution. The accuracy of the model is measured and predictions are generated on the test data.

Each prediction produces a vector of three probabilities, corresponding to each class. The final class assigned to each point is the one with the highest probability.

To analyze the performance of the model, the predictions made on the test set are graphed. Each point is represented with a different color and marker depending on the assigned class:

- **Class 0**: red color, circular marker.
- **Class 1**: green color, "x" shaped marker.
- **Class 2**: blue color, dash marker.

The graph allows you to visualize how the model classifies the new data and if there are possible classification errors.

References

[1] TensorFlow. (n.d.). *Keras guide*. TensorFlow. Retrieved March 4, 2025, from https://www.tensorflow.org/guide/keras?hl=es-419