Multiclass Logistic Regression with TensorFlow 2: Implementation and Applications

Exploring the softmax model for multiclass classification in TensorFlow 2, from its theoretical formulation to its practical implementation.

Hernández Pérez Erick Fernando

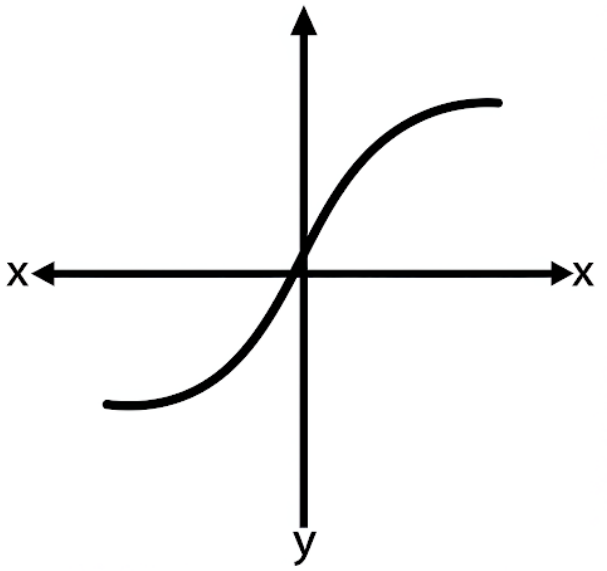

Figure 1

Logistic regression, as explored in Entry 8: Introduction to Logistic Regression: Fundamentals and Applications, is a statistical model that works as a linear separator for binary classification problems, distinguishing between a positive class and a negative class. However, in many real-world problems, data is not limited to just two categories, but can belong to multiple classes. To extend logistic regression to multiclass scenarios, it is necessary to explore alternatives that allow handling more than two categories effectively.

There are three main approaches to address multiclass classification with logistic regression:

- **One-vs-All (OvA) or One-vs-Rest (OvR)**: In this method, a logistic regression model is trained for each class, considering the target class as positive and grouping all other classes as negative.
  During prediction, each model assigns a probability, and the class with the highest probability is selected as the final prediction.
- **One-vs-One (OvO)**: In this approach, a logistic regression model is trained for each possible pair of classes, comparing them independently.
  To classify a new sample, the results of all models are evaluated and the class with the most "votes" is selected.
  This method is useful when the number of classes is small, since the number of classifiers grows quadratically (if there are k classes we need $\frac{k(k-1)}{2}$ pairs of classes) with the number of classes.
- **Softmax (Multinomial Logistic Regression)**: Instead of training multiple classifiers, a single softmax function is used that converts the output values into probabilities for each class.
  This approach allows logistic regression to assign probabilities to all classes simultaneously, ensuring that the sum of probabilities is 1.

It is the most common method in deep learning and neural networks, since it integrates efficiently into models optimized by gradient descent.

In the case of the first two examples, it is clear that the multiclass problem becomes "artificially" a binary problem and, therefore, what we saw in entry 8 would be enough to solve it; although of course this increases the number of models that need to be created.

If we want a model that can be capable, by itself, of solving the multiclass problem, then we need the softmax function.

The softmax function

"The softmax function is a soft form of the argmax function; instead of giving a single class index, it provides the probability of each class. Therefore, it allows us to compute meaningful class probabilities in multiclass settings" (Raschka et al., 2022)[5].

The softmax function, takes a vector $\mathbf{z} = (z_1, z_2, ..., z_K) \in \mathbb{R}^K$ and computes each component of vector $\sigma(\mathbf{z}) \in (0, 1)^K$ with:

$$\sigma(\mathbf{z})_i = \frac{e^{z_j}}{\sum_{j=1}^{K} e^{z_j}}$$

In simple terms, the softmax function applies the standard exponential operation to each element z_i of the input vector $\mathbf{z}$, which consists of K real numbers. Then, it normalizes these values by dividing each one by the sum of all the exponentials. This normalization ensures that the sum of the components in the output vector $\sigma(\mathbf{z})$ equals 1.

"It's called softmax because it's a 'soft' or 'smooth' approximation of the max function, which is not smooth or continuous" (Mattmann, 2020) [3].

Multinomial logistic regression

> In multinomial logistic regression we want to label each observation with a class k from a set of K classes, under the stipulation that only one of these classes is the correct one (sometimes called hard classification; an observation can not be in multiple classes). (Jurafsky & Martin, 2024)[1]

We loosely follow the representation proposed by Jurafsky & Martin, but we have modified some notation to better align with the other entries.

The output $\mathbf{y}$ for each input $\mathbf{x}$ will be a vector of length K. If class c is the correct class, we'll set $y_c = 1$ and set all the other elements of $\mathbf{y}$ to be 0, i.e., $y_c = 1$ and $y_j = 0 \ \forall j \neq c$. This kind of vector is called a one-hot vector. The job of the classifier is to produce an estimate vector $\hat{\mathbf{y}}$. For each class k, the value $\hat{y_k}$ will be the classifier's estimate of the probability $p(y_k = 1|x)$.

When we apply softmax for logistic regression, the input will (just as for the sigmoid) be $\mathbf{w}_k^T\mathbf{x} + b_k$. Now, we separate wight vectors and bias for each of the K classes. Using the softmax function, the probability of each of our output classes $\hat{y}_k$ can this be computed as:

$$p(y_k = 1|\mathbf{x}) = \frac{e^{\mathbf{w}_k^T\mathbf{x}+b_k}}{\sum_{j=1}^{K} e^{\mathbf{w}_j^T\mathbf{x}+b_j}} = \frac{\exp(\mathbf{w}_k^T\mathbf{x} + b_k)}{\sum_{j=1}^{K} \exp(\mathbf{w}_j^T\mathbf{x} + b_j)}$$

In this way, each probability is calculated separately. This perspective is illustrated by Raschka in his series of presentations *STAT 453: Introduction to Deep Learning and Generative Models.* [4]. (See figure 2).

But we can also represent the set of K weight vectors as a weight of the matrix $\mathbf{W}$ and a bias vector $\mathbf{b}$. Each row k of $\mathbf{W}$ corresponds to the vector of weights w_k. $\mathbf{W}$ thus has shape $[K \times f]$, for K the number of output classes and f the number of input features. The bias vector $\mathbf{b}$ has one value for each of the K output classes.

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}\mathbf{x} + \mathbf{b})$$

> One helpful interpretation of the weight matrix $\mathbf{W}$ is to see each row $\mathbf{w}_k$ as a prototype of class k. The weight vector $\mathbf{w}_k$ prototype that is learned represents the class as a kind of template. Since two vectors that are more similar to each other have a higher dot product with each other, the dot product acts as a similarity function. Logistic regression is thus learning an exemplar representation

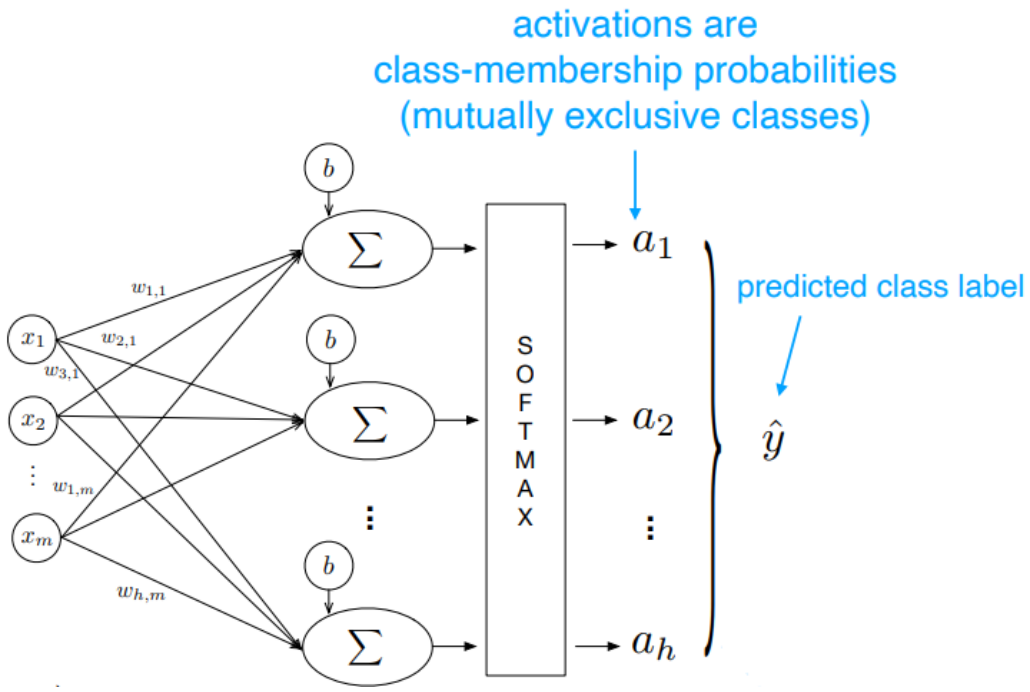

Figure 2

for each class, such that incoming vectors are assigned the class k they are most similar to from the K classes. (Jurafsky & Martin, 2024)[1]

The loss function

Let us first do a brief reminder of the Cross-Entropy loss function (For a more detailed approach see entry 8: Introduction to Logistic Regression: Fundamentals and Applications).

$$-\sum_{i}^{C=2} y_i \cdot \log(\hat{y}_i) = -y_1 \cdot \log(\hat{y}_1) - (1 - y_1) \cdot \log(1 - \hat{y}_1)$$

Now, as noted by Gómez (2018)[2] and Raschka (n.d.)[4], the loss function used with softmax consists of a softmax activation function combined with a Cross-Entropy loss function.

$$\mathcal{L} = \sum_{i=1}^{C} -y_i \cdot \log(\text{softmax}(z_i))$$

In other literature you can also find the addition for each of the samples in the training, having the following expression:

$$\mathcal{L} = \sum_{i=1}^{n} \sum_{j=1}^{C} -y_j^{(i)} \cdot \log(\text{softmax}(z_j)^{(i)})$$

In the specific (and usual) case of Multi-Class classification the labels are one-hot, so only the positive class C_p keeps its term in the loss. There is only one element of the Target vector t which is not zero $t_i = t_p$. So discarding the elements of the summation which are zero due to target labels. (Gómez, 2018)[2]

So, we can write:

$$-\log\left(\frac{e^{z_p}}{\sum_{j=1}^{C} e^{z_j}}\right)$$

Where z_p is the score for the positive class.

A simple implementation

Our data

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

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))

plt.scatter(x1_label0, x2_label0, c='r', marker='o', s=20)
plt.scatter(x1_label1, x2_label1, c='g', marker='x', s=20)
plt.scatter(x1_label2, x2_label2, c='b', marker='_', s=20)
plt.show()

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.matrix([[1., 0., 0.]] * len(x1_label0) + [[0., 1., 0.]] * len(x1_label1) +
↪   [[0., 0., 1.]] * len(x1_label2))

arr = np.arange(xs.shape[0])
np.random.shuffle(arr)
xs = xs[arr, :]
labels = labels[arr, :]

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))
test_labels = np.matrix([[1., 0., 0.]] * 10 + [[0., 1., 0.]] * 10 + [[0., 0., 1.]] * 10)
train_size, num_features = xs.shape
```

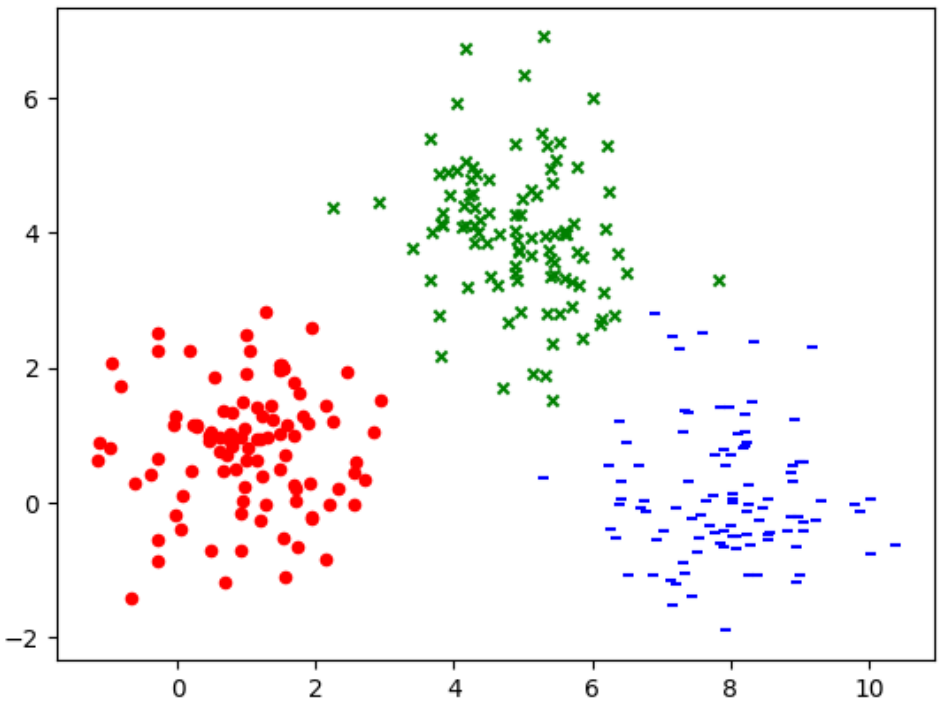

Figure 3: Our data

The model

```python
import numpy as np
import tensorflow as tf

# Hyperparameters
lr = 0.01
training_epochs = 1000
num_labels = 3
batch_size = 100

# Initialization of Weights and Biases
W = tf.Variable(tf.zeros([num_features, num_labels]), dtype=tf.float32)
b = tf.Variable(tf.zeros([num_labels]), dtype=tf.float32)

# Loss function (Categorical Crossentropy with logits)
loss_fn = tf.losses.CategoricalCrossentropy(from_logits=True)

# Optimizer (Adam)
optimizer = tf.optimizers.Adam(learning_rate=lr)

# Training function
def train_step(batch_xs, batch_labels):
    with tf.GradientTape() as tape:
        logits = tf.matmul(batch_xs, W) + b  # Forward pass
        loss = loss_fn(batch_labels, logits)  # Loss calculation

    # Compute and apply gradients
    gradients = tape.gradient(loss, [W, b])
    optimizer.apply_gradients(zip(gradients, [W, b]))
```

```
    return loss

# Model training
for epoch in range(training_epochs):
    # Shuffle data in each epoch
    arr = np.arange(train_size)
    np.random.shuffle(arr)
    xs = xs[arr]
    labels = labels[arr]

    avg_loss = 0
    total_batches = train_size // batch_size

    for step in range(total_batches):
        offset = step * batch_size
        batch_xs = xs[offset:offset + batch_size, :]
        batch_labels = labels[offset:offset + batch_size]

        loss = train_step(batch_xs.astype(np.float32), batch_labels.astype(np.float32))
        avg_loss += loss / total_batches

    if epoch % 100 == 0:
        print(f"Epoch {epoch}: Average loss = {avg_loss.numpy()}")

# Prediction on test data
logits_test = tf.matmul(tf.cast(test_xs, tf.float32), W) + b
predictions = tf.nn.softmax(logits_test)

# Print the first predictions
print("First 5 predictions:\n", predictions.numpy()[:5])
```

The presented code implements a Softmax regression using TensorFlow to solve a multiclass classification problem.

In the first section of the code, the hyperparameters of the model are defined, including the learning rate (`lr`), the number of training epochs (`training_epochs`), the number of classes (`num_labels`) and the batch size (`batch_size`). Next, the weights (`W`) and bias (`b`) of the model are initialized with values of zero. These parameters will be tuned during training using the optimization algorithm.

The loss function used is categorical cross-entropy, implemented with `tf.losses.CategoricalCrossentropy(from_logit` This function is suitable for multiclass classification problems, as it measures the discrepancy between the predicted probabilities and the actual labels in one-hot encoding format. For optimization, the stochastic gradient descent (SGD) algorithm is used using `tf.optimizers.SGD(learning_rate=lr)`, which allows updating the weights and minimizing the loss function.

Model training is performed in the `train_step()` function, where the model output is calculated as:

$$\text{logits} = \text{tf.matmul}(\text{batch\_xs}, \text{W}) + b. \tag{1}$$

The loss is then evaluated using cross entropy, and the gradients of the weights and bias with respect to the loss function are calculated. Finally, these gradients are applied to update the model parameters using the SGD optimizer.

During training, the data is randomly shuffled in each epoch to improve the generalization of the model. It is then split into batches and the weights are updated at each iteration. At the end of every 100 epochs, the average loss is printed to monitor the model performance.

Once training is complete, the prediction is made on a test data set. The logits are calculated as:

$$\text{logits\_test} = \text{tf.matmul}(\text{tf.cast}(\text{test\_xs}, \text{tf.float32}), \text{W}) + b, \tag{2}$$

and then the function `tf.nn.softmax(logits_test)` is applied, which converts the output values into normalized probabilities. Finally, the first five predictions of the model are printed, allowing the results obtained to

be visualized.

The Test

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

# Plot the points with colors according to the predicted class
plt.figure(figsize=(8, 6))

# Points of class 0
plt.scatter(test_xs[predicted_classes == 0, 0], test_xs[predicted_classes == 0, 1],
↪  c='r', marker='o', s=50, label="Class 0")
# Points of class 1
plt.scatter(test_xs[predicted_classes == 1, 0], test_xs[predicted_classes == 1, 1],
↪  c='g', marker='x', s=50, label="Class 1")
# Points of class 2
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 the plot
plt.show()
```

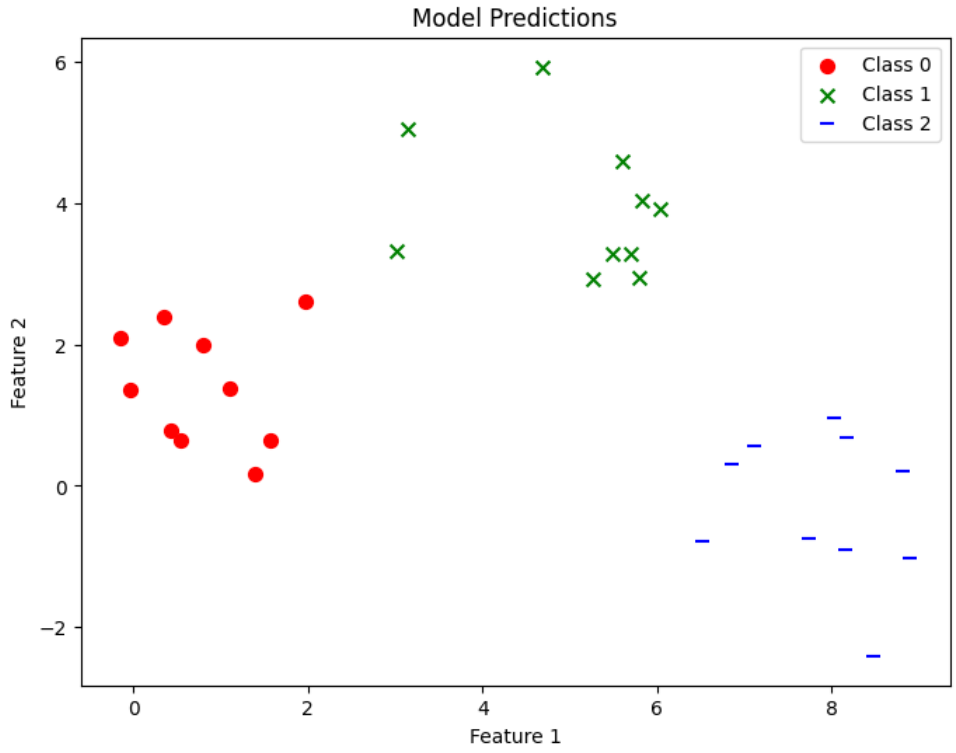

Figure 4: Our test

References

[1] Jurafsky, D. & Martin, J. (2024). *Logistic Regression*. Available at: https://web.stanford.edu/~jurafsky/slp3/5.pdf

[2] Gómez, R. (2018). *Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names*. Available at: https://gombru.github.io/2018/05/23/cross_entropy_loss/

[3] Mattmann, C. (2020). *Machine Learning with TensorFlow*. Manning.

[4] Raschka, S. (n.d.). *STAT 453: Introduction to Deep Learning and Generative Models*. Available at: https://sebastianraschka.com/pdf/lecture-notes/stat453ss21/L08_logistic__slides.pdf

[5] Raschka, S., Liu, Y. & Mirjalili, V. (2022). *Machine Learning with PyTorch and Scikit-Learn*. Packt.