**1. Recurrent neural networks for text classification**

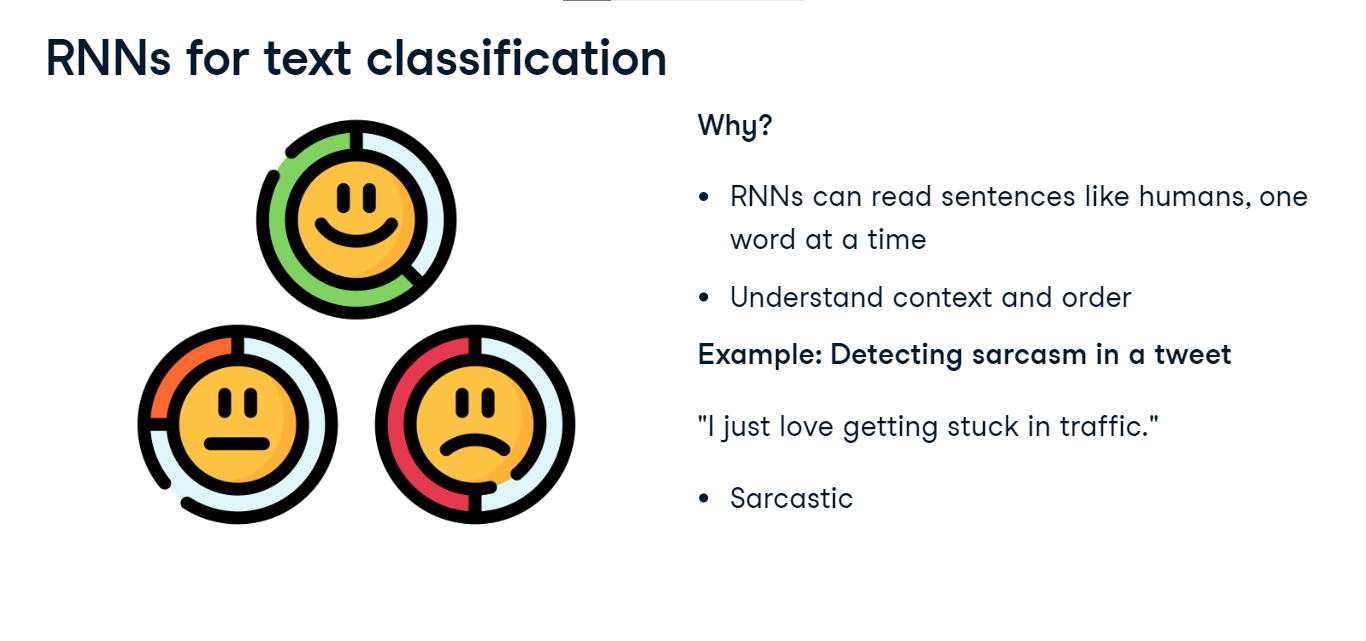
We've explored CNNs for text; now it's time to explore Recurrent Neural Networks, or RNNs, for text.

**2. RNNs for text**

Recurrent Neural Networks, or RNNs, are great at handling sequences of varying lengths. They maintain an internal short-term memory, enabling them to learn patterns across time. Unlike CNNs that spot patterns in chunks of text, RNNs remember past words to understand the whole sentence's meaning. Today, we will explore how to employ RNNs for text classification.

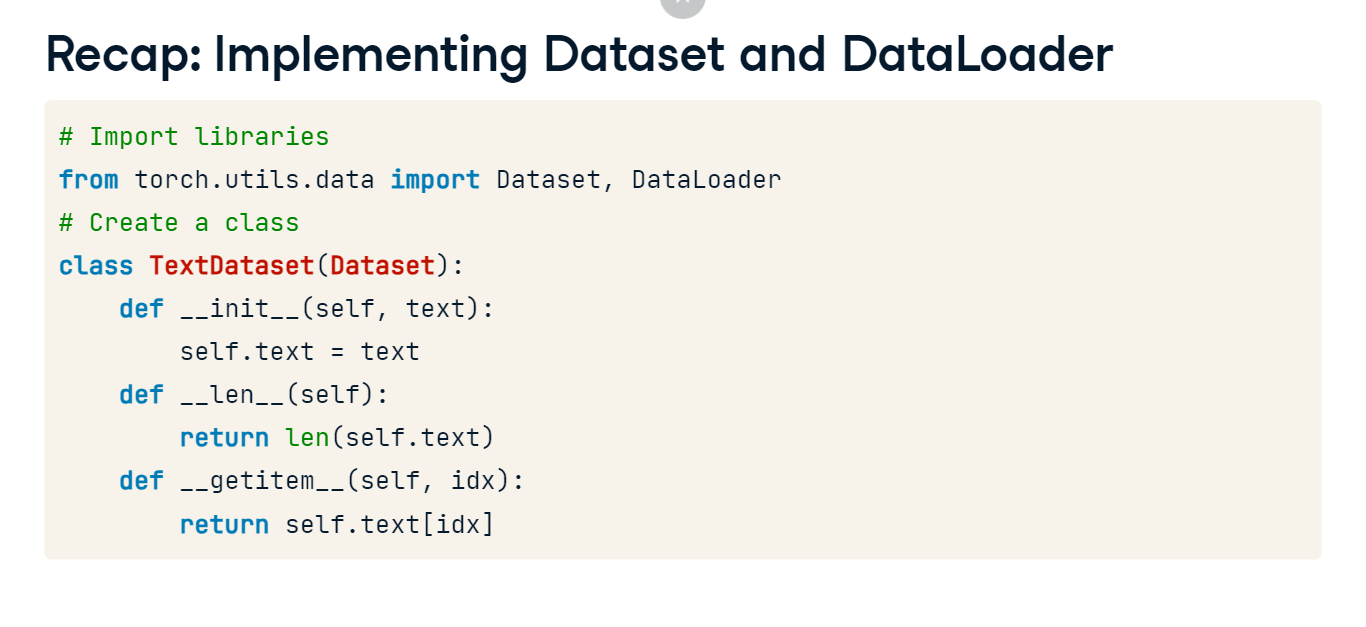
**3. RNNs for text classification**

RNNs are suitable for text classification because they process sequential data like humans read, one word at a time, allowing them to capture the context and order of words. Consider the tweet, "I just love getting stuck in traffic"; RNNs can accurately classify the tweet as sarcastic.



**4. Recap: Implementing Dataset and DataLoader**

Let's remind ourselves how to apply Dataset and DataLoader for text data in PyTorch. We create a custom class TextDataset, serving as our data container. The init method initializes the dataset with the input text data. The len method returns the total number of samples in the dataset, and the getitem method allows us to access a specific sample at a given index. This class, extending PyTorch's Dataset, allows us to organize and access our text data efficiently.



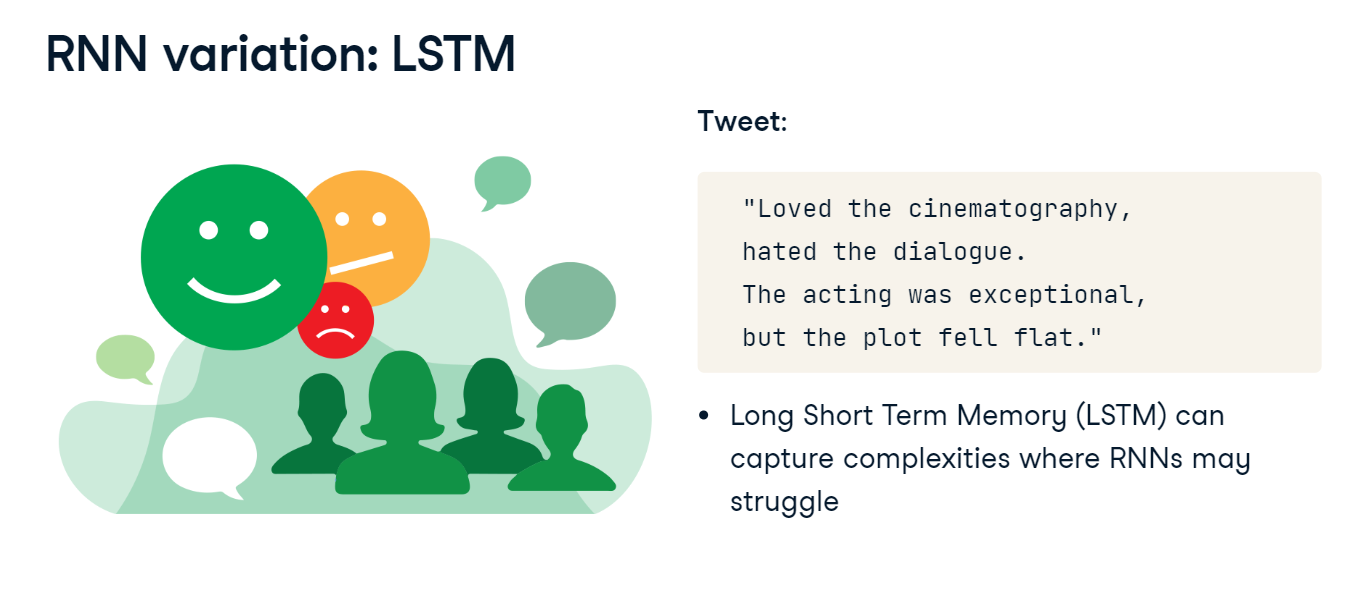
**5. RNN implementation**

Now let's take a look at an example of sentiment analysis for movie review from a tweet. We want to train an RNN model to classify movie reviews as either positive or negative. We can use our entire text processing pipeline here to feed to the model. This includes encoding or embedding. We preprocess the tweet and convert it to a tensor, which is not shown here for brevity. Then, we pass the preprocessed tensor through the model to make a sentiment prediction. In this case, the model predicts that the sentiment is "Positive."

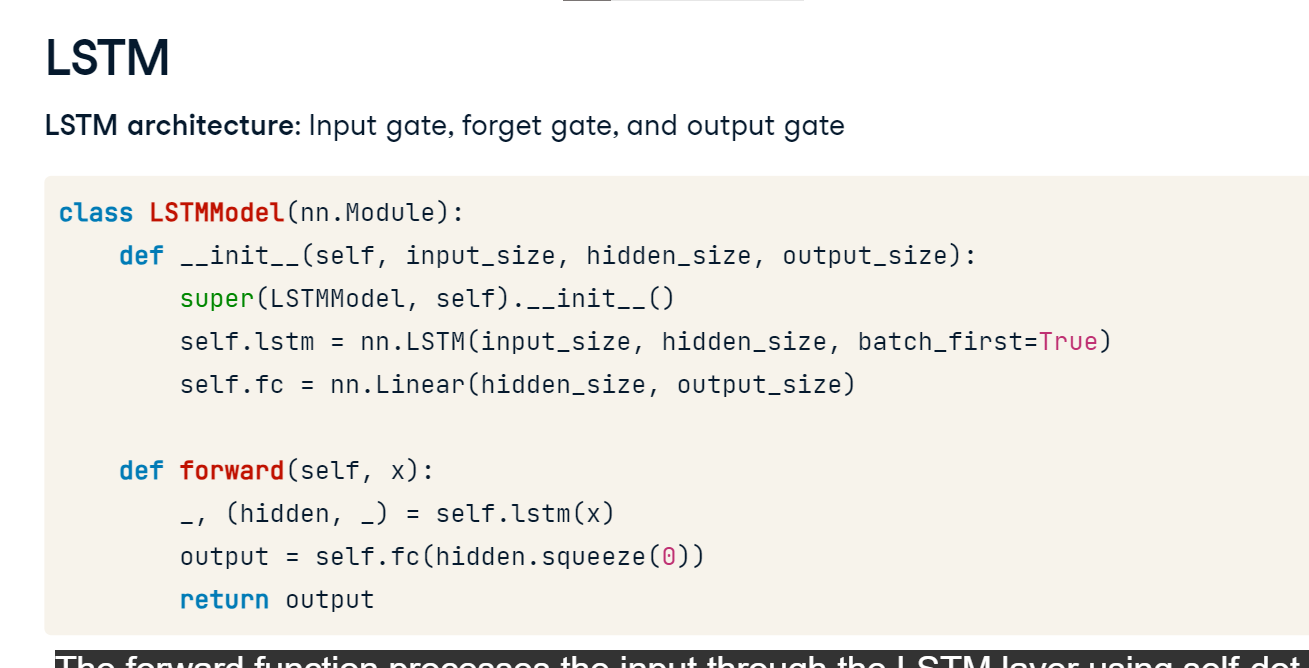


**6. RNN variation: LSTM**

But what if the tweet is not so straightforward to understand the sentiment. Take the tweet, "Loved the cinematography, hated the dialogue. The acting was exceptional, but the plot fell flat". These complex sentences contain subtle nuances and conflicting sentiments. While RNNs may struggle to capture the negative sentiment, Long Short Term Memory models or LSTMs excel at capturing such complexities. They can effectively understand the underlying emotions, making them a powerful tool for sentiment analysis.

**7. LSTM**

LSTMs have input, forget, and output gates that enable them to store and forget information as needed. This architecture is ideal for complex classification tasks. The code defines an LSTM model using nn-dot-LSTM, with an initialization function that sets the input size, hidden size, and batch-first parameter. The forward function processes the input through the LSTM layer using self-dot-lstm, and the rest is similar to RNN.

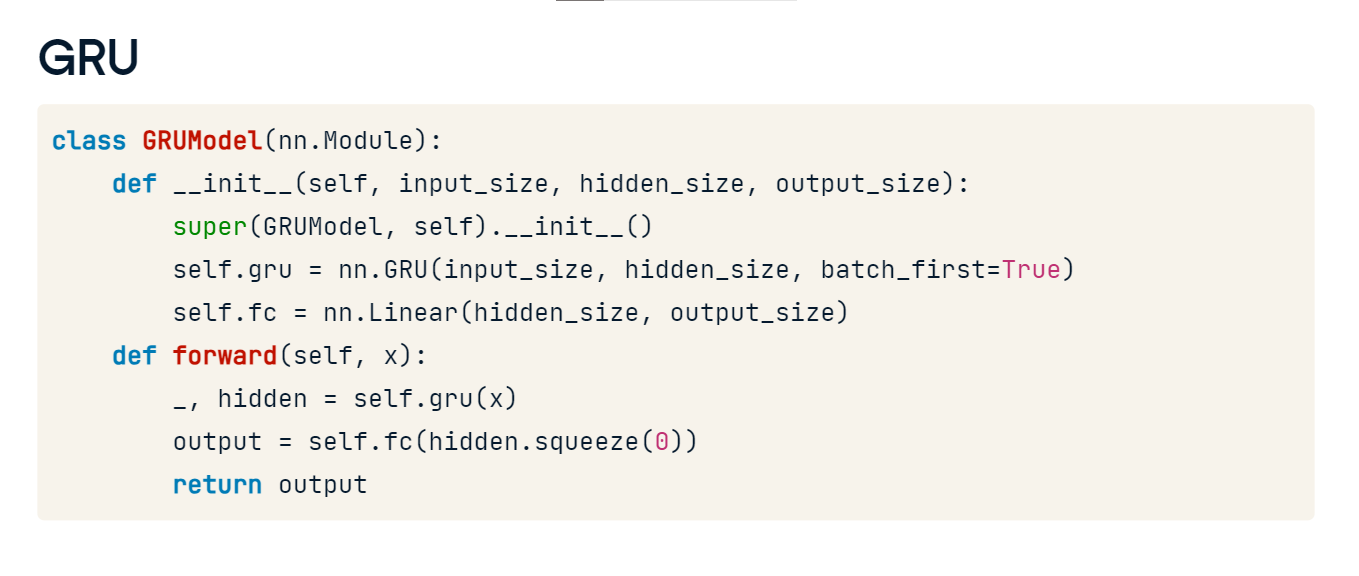


**8. RNN variation: GRU**

But, what if we wanted to detect spam emails without needing the full context. Given an email subject like "Congratulations! You've won a free trip to Hawaii!", a Gated Recurrent Unit or GRU, can quickly recognize spammy patterns without needing the full context. This makes them suitable for tasks like spam detection, sentiment analysis, text summarization, and more.

**9. GRU**

GRUs are a streamlined version of LSTMs that trade some complexity for faster training. The code defines a GRU model using nn-dot-GRU, with an initialization function that specifies the input size, hidden size, and batch-first parameter. The forward function remains the same, with the change of self-dot-lstm becoming self-dot-gru.



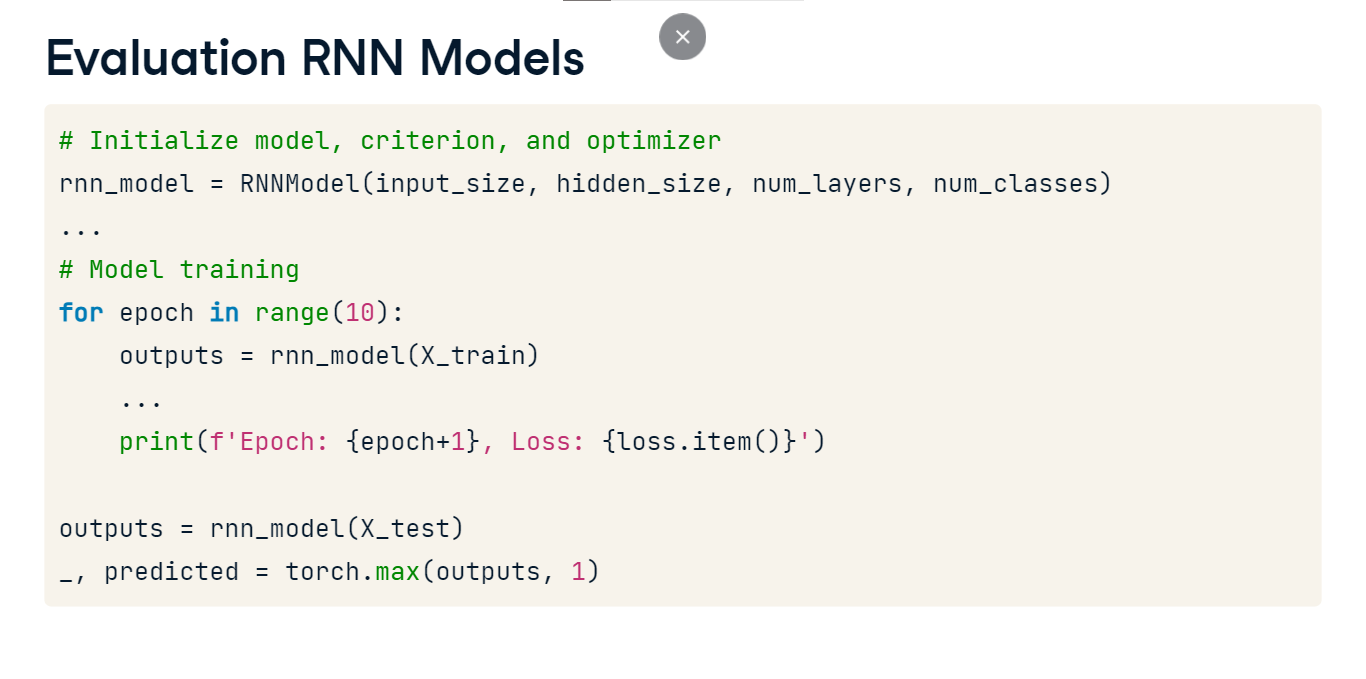
# Evaluation metrics for text classification

**Why evaluation metrics matter**

Picture this: Our model, designed to assess the sentiment of book reviews, suggests that a best-seller has mostly negative reviews. Should we accept its judgment? We can use evaluation metrics to answer this.

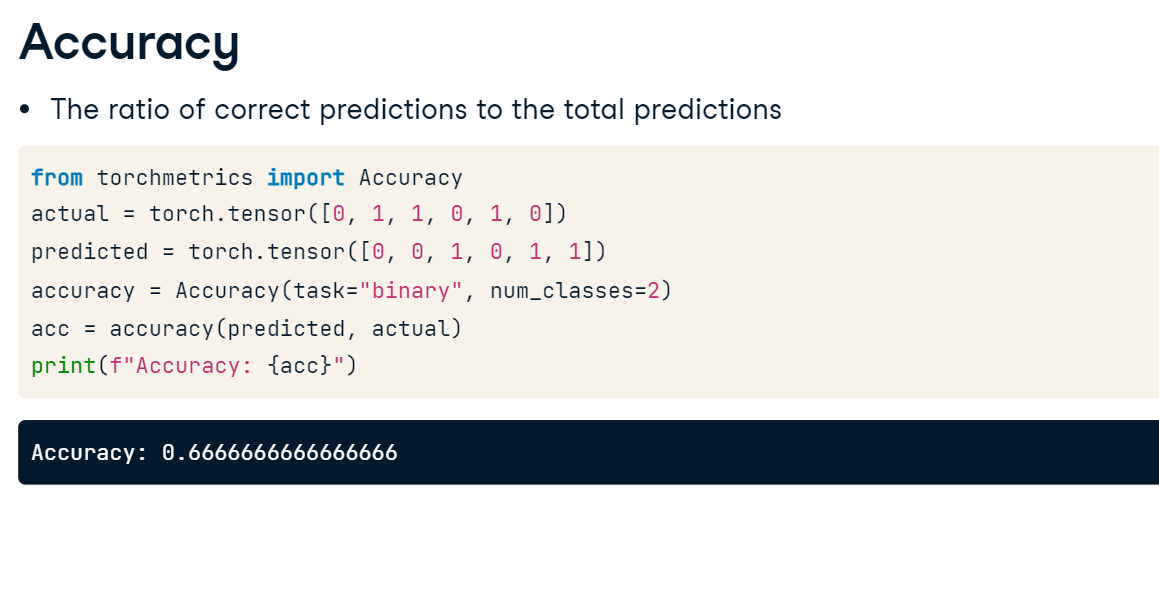
**Evaluation RNN Models**

Before evaluating, we must generate predictions from the model. First, we pass the test dataset through the model to obtain the output predictions for each class. Next, we store the predictions in the predicted variable using the torch-dot-max function that returns the indexes of the maximum values along the specified dimension, indicated by the argument one. We'll use the predicted variable for evaluation metrics.



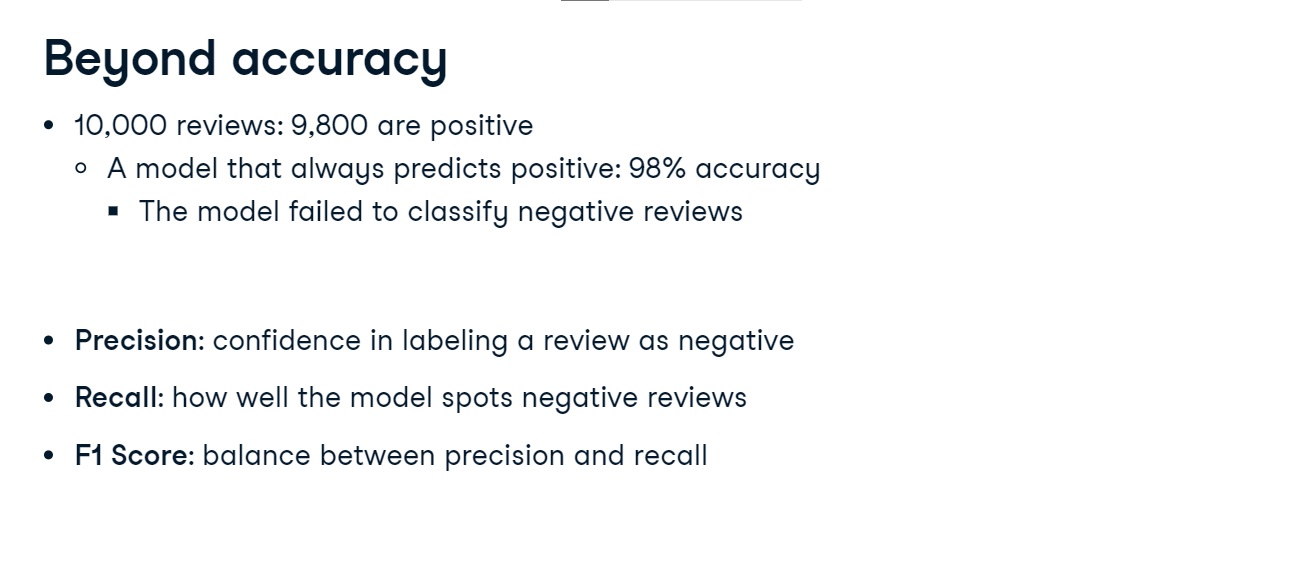
**Accuracy**

The most straightforward metric is accuracy, the ratio of correct predictions to the total predictions. Using torchmetrics, the tensors actual represent our actual labels, and predicted the model predictions. We want to determine if an instance belongs to class zero or class one, a binary classification. The accuracy class is initialized with a binary task and num\_classes set to two for our two categories. The task can also be multiclass if there are more than two categories to classify. Passing labels to the accuracy instance gives the model's accuracy score. A score of zero-point-66 indicates the model predicted just over 66 percent of the samples correctly. A good score can vary based on the complexity of the problem. Scores range from zero to one, with higher scores representing greater accuracy. For example, zero-point-75 may be reasonable for sentiment analysis but poor elsewhere. As we learn more about metrics, we'll see that accuracy alone doesn't capture everything.



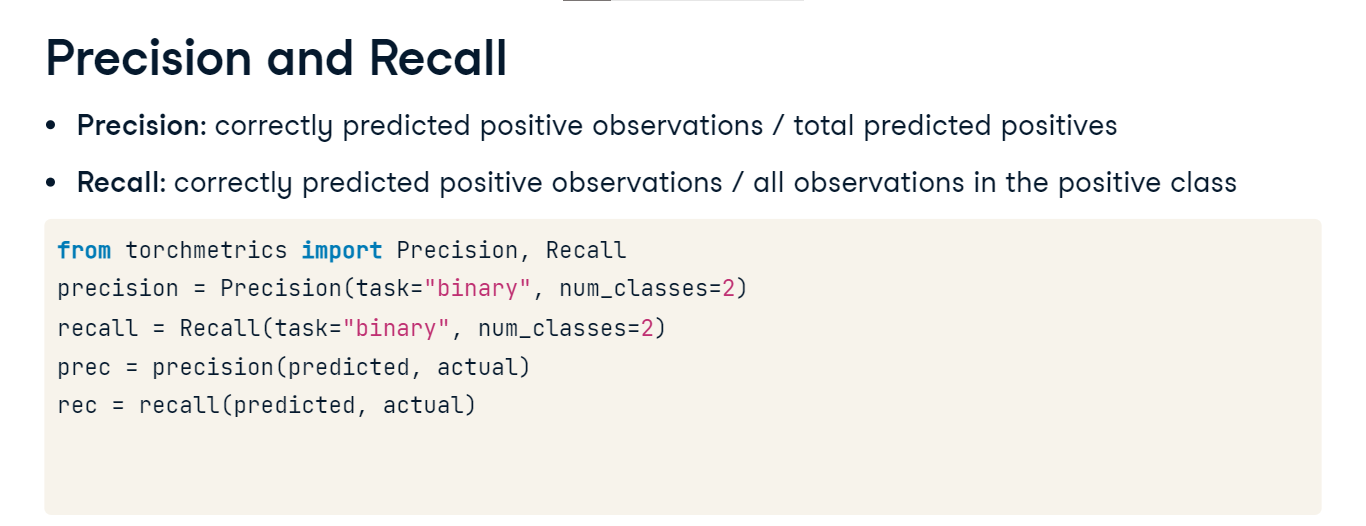
**Beyond accuracy**

Imagine a dataset of 10,000 book reviews where 9,800 readers adore the book and 200 found faults. Let's assume our model predicts all instances as positive, making it 98% accurate! But look closer. Such a model can't classify a single negative sentiment. Enter precision, which questions the model's confidence in labeling a review as negative. Recall checks how well the model spots actual negative reviews. The F1 Score harmonizes these two, ensuring neither is neglected. If we were to trust accuracy alone, we'd miss significant feedback. Let's explore each in more detail.

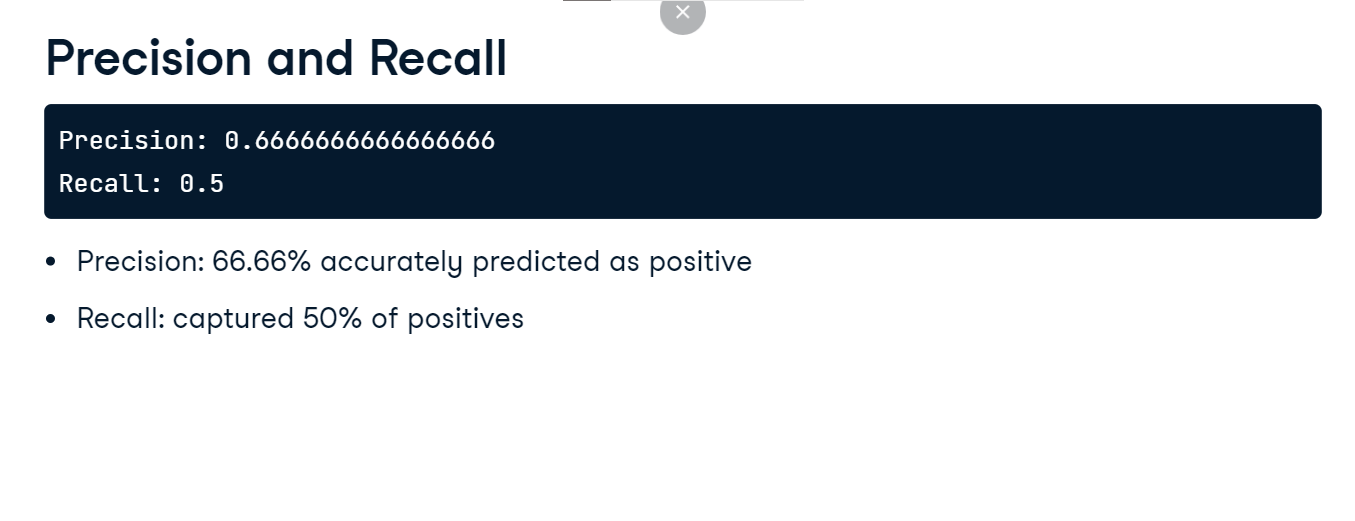


**Precision and Recall**

Precision is the ratio of correctly predicted positive observations to the total predicted positives. Recall is the ratio of correctly predicted positive observations to all observations in the actual positive class. To calculate these, we import the Precision and Recall classes from torchmetrics, use the same parameters as before, and print the results.

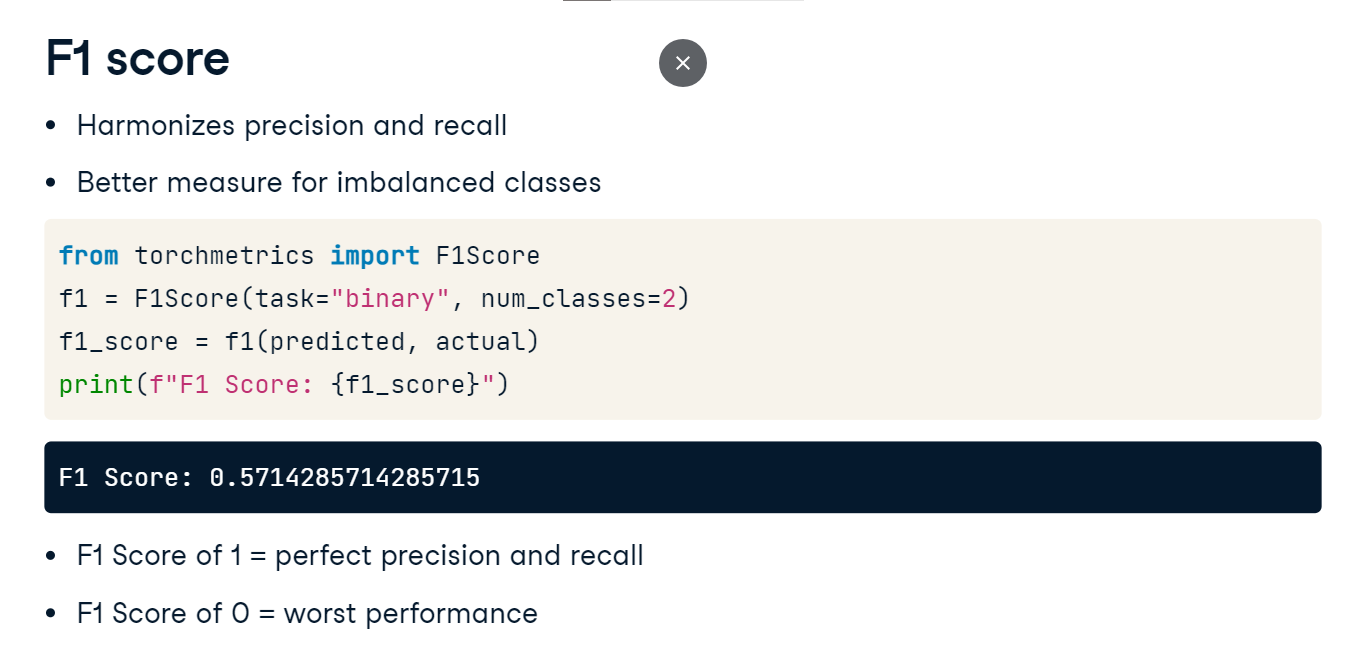


A precision of zero-point-six-six suggests that out of all positive predictions, just over 66 percent were accurate. Meanwhile, a recall of zero-point-five signifies the model captured 50 percent of all genuine positives. Like accuracy, the scores range from zero to one. The complexity of the problem needs to be considered when defining a score as good or bad.



**F1 score**

The F1 Score harmonizes precision and recall and is especially useful when dealing with imbalanced classes. To calculate it, we import the F1 Score class from torchmetrics and instantiate it with the same parameters. An F1 Score of one indicates perfect precision and recall, while a score of zero indicates the worst possible performance. Here F1 Score of zero-point-57 suggests a reasonably balanced trade-off between precision and recall, but this trade-off will depend on the task.



**Considerations**

In some instances, such as with multi-class classification, we may find that all scores are identical. Generally, this indicates a model is performing well across all classes. But remember to always consider the problem when interpreting results!