Chapter 5. Explainability for Text Data

Language models play a central role in modern-day deep learning use cases and the field of natural language processing (NLP) has advanced rapidly, especially over the last few years. NLP is focused on understanding how human language works and is at the heart of applications such as machine translation, information retrieval, sentiment analysis, text summarization, and question answering. The models built for these applications rely on text data to understand how human language works, and many of the deep learning architectures commonly used today, like LSTMs (long short-term memory), attention, and transformer networks, were developed specifically to handle the nuances and difficulties that arise when working with text.

Perhaps the most significant of these advances is the transformer architecture, introduced in the paper "Attention Is All You Need."[1] Transformers rely on the attention mechanism and are particularly well-equipped for handling sequential text data. This is partly because of their computational efficiencies and because they are better able to maintain context since text is processed as a whole rather than sequentially. Soon after transformers hit the scene, BERT, which stands for Bidirectional Encoder Representations from Transformers, was introduced and it beat all the GLUE[2] (General Language Understanding Evaluation) benchmarks for NLU (natural language understanding) tasks ranging from sentiment classification, textual entailment, text similarity, and grammatical correctness. Since BERT, other record-breaking, transformer-based models have emerged, leading to better and better (and increasingly larger and larger) language models such as OpenAI's GPT2 and GPT3, Google's T5, DeepMind's Gopher, and more recently PaLM, a 540 billion parameter dense decoder transformer model that is capable of mathematical reasoning, code writing, and even explaining jokes.[3]

With these impressive developments, there is an increased desire to better explain how these models work. Explainable NLP has become a strong focus in the current research community aimed at better understanding how these large language models work and what they are learning. Of course, there is an important distinction to be made between standard,

day-to-day text models used by the typical ML practitioner and these state-of-the-art (SOTA) models like T5, GPT3, or PaLM. Models like T5, GPT3, and PaLM are the AI equivalent of a Formula One racing car,[4] and indeed, many of these models are outside the realm of the typical practitioner. These models train for weeks on end and require compute resources that the average practitioner or business does not have access to.

However, their underlying technology has become commonplace and, with easy-to-use implementations available in the Hugging Face library, many of these advanced architectures aren't so out of reach. In fact, it's likely model architectures like BERT, XLNet, and GPT2 will find their way into your day-to-day toolkit, if they haven't already. Just as we have seen when training image models, there is immense value in leveraging a pre-trained version of the large language models either directly or for fine-tuning on a more task-specific use case.

The focus of this chapter is to discuss commonly used explainability techniques and understand how they are applied when working with text data. The goal here is not to explain how complex architectures like T5 or transformers work in general; instead, we'll focus more on text models more closely aligned with the general ML practitioner. In this chapter, we'll introduce some new techniques but also revisit some of the techniques that we saw in previous chapters, such as LIME and Integrated Gradients. We'll show how these familiar methods can be adapted to work for text data due to the unique constraints of building language models. We'll also look into some of the tools that have been developed by the community to help explain text models.

Overview of Building Models with Text

In a way, natural language models are no different than models built on tabular or image data. As when building any machine learning model, features are extracted from the data and preprocessed for training. Oftentimes a deep neural network or some other advanced model architecture is used. The model's output prediction is compared against the true label and the weights are updated via a variant of gradient descent through backpropagation. When it comes to natural language models, however, the raw data is text, so special tricks and techniques are needed to handle the nuances of a textual dataset. In fact, because the techniques for working with text are so specialized, NLP is an active research area in its own right and there are a number of courses and specializations designed to specifically address how to work with natural language models.

In this section, we'll discuss at a high level the main ideas and key aspects to have in mind when building text models. This will also allow us to set up common terminology and introduce concepts that we'll need later. As we proceed through the chapter, we'll introduce other more advanced treatments, but it's important to keep these initial steps in mind because they can affect the explainability of the model downstream.

Tokenization

The first step to building a text-based model is to preprocess the text data into a format that can be used by the model. Machine learning models are numeric machines so, one way or another, text needs to be converted into a numeric quantity. Perhaps the most basic or naive way to convert a word into a vector is through one-hot encoding. One-hot encoding is a preprocessing step in which categorical variables are converted into a sparse vector consisting of a single one as a placeholder for a given categorical value, with all other indices set to zero (e.g., `[0,0,1,0,0,0]`). In the text setting, the words are the categorical variables. This poses a problem because our vocabulary could easily be on the order of tens or hundreds of thousands of words. This means that one-hot encoding maps a word like *cockatoo* to a vector of shape 100,000 with all zeros and a single one representing the word *cockatoo*. Figure 5-1 shows an example of one-hot encoding to a sentence of words.

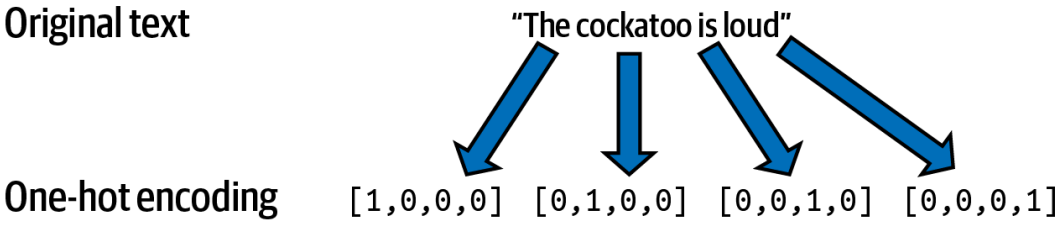

Figure 5-1. Transforming the sentence "The cockatoo is loud" into a sequence of one-hot encoded vectors. Here, the one-hot encoding is for a vocabulary of just those four words; in reality, the vocabulary could easily be on the order of tens or hundreds of thousands of words.

For some simple models and use cases, taking a vocabulary to be the 10,000 most frequently used words in a corpus and an "out-of-vocabulary" bucket for everything else works well enough. But more sophisticated models, in particular transformer models, require a more comprehensive vocabulary. There is a bit of a trade-off. The larger the vocabulary, the more expressive the model can be in learning the relationship between a wide variety of words. However, a larger vocabulary also leads to more sparse one-hot encoded vectors.

So, what should our vocabulary be? There are more than 500,000 words in the English language. The goal is to find the most meaningful, and

preferably the smallest, representation that will make sense for our model. Should we include punctuation? Should we include rarely used words like *crapulous* or *ostensibly*? Should we include words from Old English like *ye Olde Worlde*? Or more recently, how do we deal with emoticons and emojis, like "`¯\_(ツ)_/¯`" and "🤦‍♀️"? Should we split up word pairs like *Costa Rica* into two separate words *Costa* and *Rica*, or does it deserve its own token? How do we treat two words with a common root, like *chirp* and *chirping*? What about misspellings that are bound to arise, like *recieve* instead of *receive*? Or words that are spelled correctly but used in context incorrectly, like *affect* versus *effect* or *surreptitious* versus *serendipitous*. What if our model is multilingual? How do we incorporate the vocabulary of all the languages we want our model to accommodate? Should our vocabulary now include all languages and their idiosyncrasies? Surely there is a sweet spot.

This is where tokenization comes in. Given a character sequence (e.g., a sentence) and a defined document unit (e.g., a movie review), *tokenization* is the task of splitting up the character sequence into essential pieces, called *tokens*, and discarding irrelevant characters. Tokenization is at the forefront of any NLP pipeline when preprocessing text for machine learning. It is the translator from human-readable text into a numeric categorical representation that the machine learning model can use. If you've had experience working with natural language models, it's likely you are already familiar with tokenization. If not, or if you've forgotten the details, we'll quickly discuss the basics here. As we'll see in this chapter, the artifacts of tokenization often appear in the final explanations we may receive and even affect how some techniques are implemented. Let's begin by discussing what tokens are and how they're used in natural language processing.

Tokens are integer representations of a word or word piece and can be used to split a sentence into words or subwords. A subword is obtained by breaking down a word into smaller meaningful words. For example, the word *cockatoos* or *let's* is broken down into the subwords *cockatoo* and *s* or *let* and *'s*, respectively. You train a tokenizer just as you might train a large neural network. Of course, instead of learning parameter weights, you're passing over the entire dataset to determine which tokens to use and how to split words or sentences in your dataset. There are a number of popular tokenization algorithms, and determining an efficient and effective tokenization is an active area of current research.

To illustrate the general approach to tokenization, we'll discuss one of the most common algorithms: byte pair encoding (BPE), which builds a tokenizer vocabulary from an alphabet of bytes. Other commonly used tokenization algorithms are WordPiece, Unigram, and SentencePiece.

To train a BPE tokenizer, you start by tokenizing all characters in the entire training corpus. Then, passing over the dataset, you identify common pairs of tokens and merge the pairs into a single token. For example, after seeing *m*, *o*, *v*, *i*, and *e* enough times, the tokenizer will merge these five tokens into a single token *movie*. You iteratively repeat this process until the number of tokens in the vocabulary reaches the size you want. As with most things in machine learning, there is a trade-off when choosing the size of the vocabulary. If the vocabulary size is too small, then the vocabulary is too coarse, and different words may end up being tokenized as the same. However, increasing the vocabulary size too much increases the computational cost unnecessarily. A BPE tokenizer is nice because you don't have to pre-prescribe words in your vocabulary, and you don't need special tokens for unknown words. Word tokens are learned by passing through the data, and BPE ensures that the most common words are represented as a single token, while rare words are broken down into subwords.

Transfer learning is a technique that allows you to take advantage of the learned embeddings of a model that was trained on a large dataset. It provides an advantage over training from scratch. Just as transfer learning leverages pretrained weights of a model, pretrained tokenizers are a useful resource for the average practitioner. The quality of your model is heavily dependent on the quality of your tokenizer vocabulary, so it's to your benefit to take advantage of a tokenizer that was pretrained on a corpus likely much larger than yours. Luckily, Hugging Face has an entire library of pretrained tokenizers that can easily be loaded and incorporated into your ML workflows. For example, the BERT tokenizer is a class inside the Hugging Face transformer library and can be loaded with just a few lines of code:

```
from transformers import BertTokenizer
tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
```

BERT was trained using the WordPiece tokenization algorithm. With the tokenizer loaded, we can then easily tokenize a sample text using the `tokenize` method and convert the tokens to integer IDs, as shown in the

following code block and illustrated in Figure 5-2. These IDs can then be used to create embeddings for training the model, which we'll discuss in the next section:

```
sample_text = "If you like the original, you'll love this movie."
tokens = tokenizer.tokenize(sample_text)
ids = tokenizer.convert_tokens_to_ids(tokens)
```

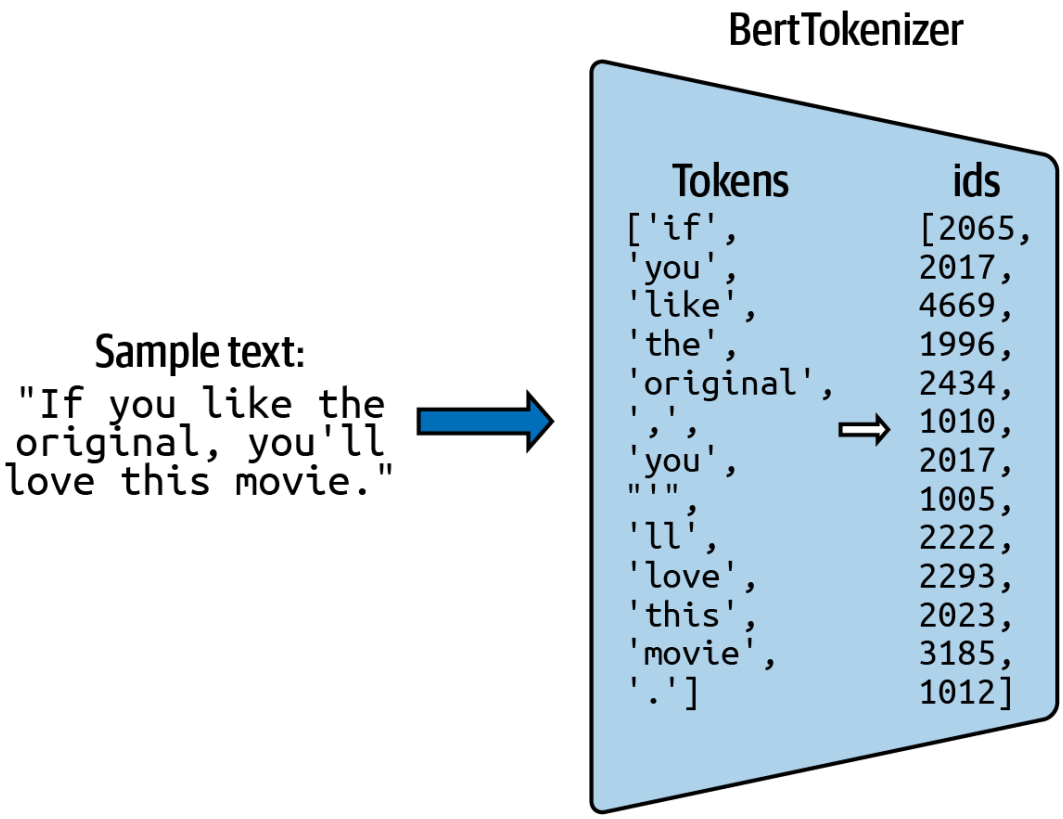

Figure 5-2. The BERT tokenizer is our translator from human-readable text into a numerical encoding that our machine learning model can use. Words and word pieces are first tokenized, then converted to integer IDs before being passed to the model.

Word Embeddings and Pretrained Embeddings

After tokenization, the words of our text dataset can be represented by an integer ID, and sentences can be represented as sequences of integers. This is only half the battle, however. The next step is to convert these tokens into a meaningful representation. Although the tokens are numeric, they don't have meaningful magnitude; the tokens are merely numeric placeholders. Converting each token into a vector creates a meaningful numeric representation of the word that can be processed by the model.

Essentially, the IDs are categorical features. Each ID represents a single token in our vocabulary. As with any categorical feature, these IDs must be encoded before being passed to the model. If the features don't have any relationship with one another, then one-hot encoding is a common

approach. However, these categorical IDs represent words in our corpus (or, more specifically, tokens in our vocabulary), and there are close relations between some words. So, a naive one-hot encoding won't work. Another issue with one-hot encoding is the high cardinality of our vocabulary. Our tokenizer vocabulary could consist of tens or hundreds of thousands of tokens. One-hot encoding the token IDs would lead to very sparse tensors and really inhibit the learning process.

Instead, you typically use an embedding layer to encode token IDs. A *word token embedding* converts the categorical ID representing a given word token into a dense vector. This essentially turns the sequence of token IDs into a sequence of dense vectors. First, the sentence is broken down into word tokens, then tokens are converted into categorical IDs, and finally the IDs are mapped to an embedding space, as shown in Figure 5-3. The embeddings are a mathematical vector representation that captures different aspects of the data so that similar words are mapped to points that are close to each other. For example, in Figure 5-3, each word is mapped to a vector in 8-dimensional Euclidean space. The sequence of IDs is just a placeholder for each word token. This sequence of dense vectors in the embedding space is ultimately what is passed to the model during training.

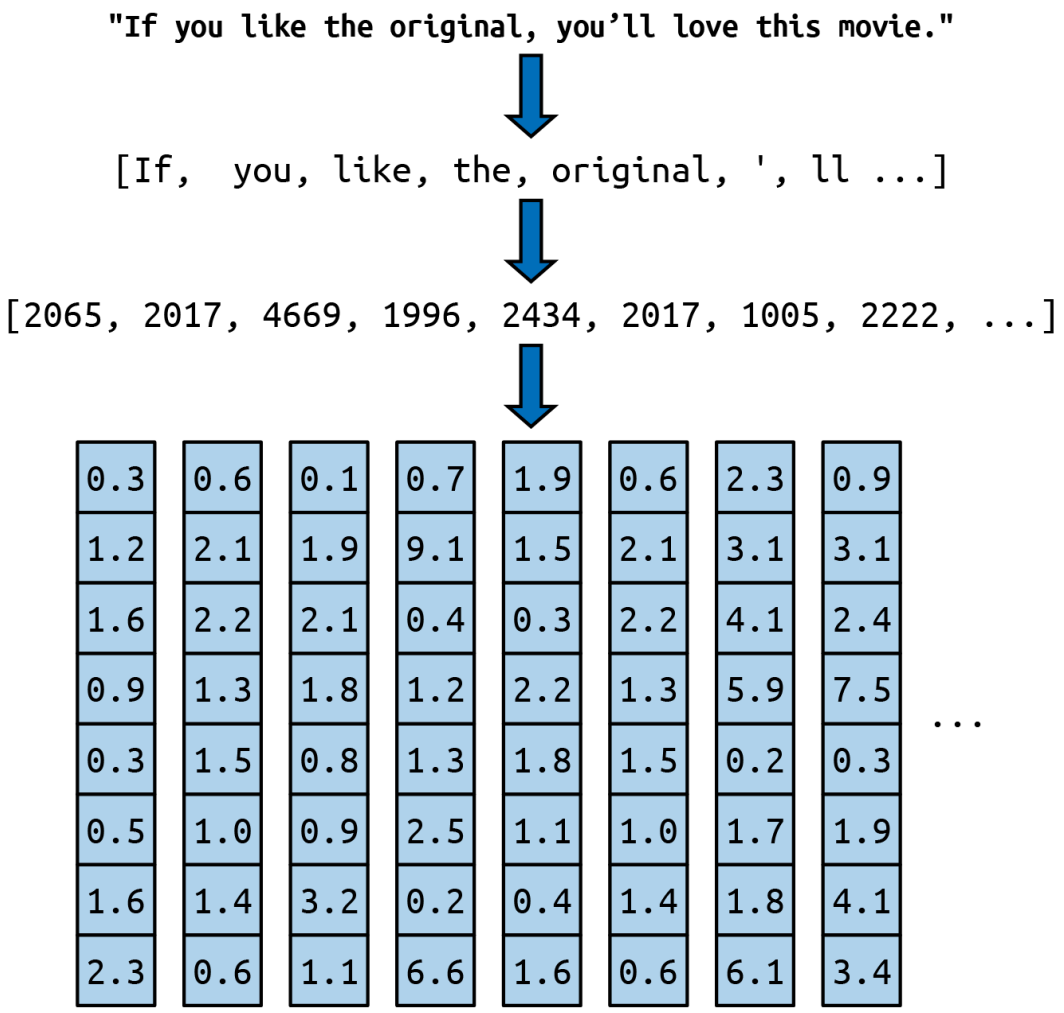

Figure 5-3. The movie review text is tokenized, then converted into token IDs, then passed through an embedding layer. The resulting sequence of dense vectors is then passed to the model for training.

To add an embedding layer in Keras, for example, you specify the embedding dimension for the output vector space, as shown in the following code. This makes the embedding layer a component of the network, and the weights for mapping from the categorial token IDs to the word embedding is then learned during the training process (see the notebook in this book's GitHub repository for the full code):

```
model = tf.keras.models.Sequential()
model.add(
    tf.keras.layers.Embedding(input_dim=vocabulary_size,
                              output_dim=embedding_dim,
                              input_length=MAX_SEQUENCE_LENGTH)
)
```

As usual with deep learning, it's a good idea to leverage a pretrained embedding whenever possible. TensorFlow makes this easy with TensorFlow Hub, where there are many pretrained text embeddings that can be used. We can load Google's neural network language model (NNLM) architec-

ture that has been pretrained on the English Google News 7B corpus with just a couple lines of code:

```
import tensorflow_hub as hub
embedding = "https://tfhub.dev/google/nnlm-en-dim50/2"
hub_layer = hub.KerasLayer(embedding, input_shape=[],
                           dtype=tf.string, trainable=True)
```

Given a string of text, this `hub_layer` outputs a 50-dimensional dense vector that can be used for training a text classification model. In fact, we use this embedding to train a sentiment analysis classifier on the IMDB movie review dataset. See this book's GitHub repository for the full code. We'll use the trained model to illustrate some of the explainability methods we discuss in this chapter.

Now that we've reviewed some of the basics of working with text data, let's now turn our attention to explainability methods. Since explainability methods rely on model features and feature importances, it's helpful to keep in mind these tokenization and embedding preprocessing steps when interpreting the results of an explainability technique. Also, we'll see how some techniques we've already discussed, like LIME and Integrated Gradients, are applied in the context of text data.

LIME

What you need to know about LIME:

- LIME stands for *local interpretable model-agnostic explanations*.
- When working with given text, LIME perturbs a given example text by randomly removing words.
- The cosine distance is used to compute proximity measure between the input instance and the sampled perturbations.
- LIME performs better with around 1,000 perturbations, especially for longer inputs. Beyond that, there isn't much improvement.

Pros	Cons
• LIME is model agnostic, so it can be applied to any machine learning model in any framework. • This is an easy-to-implement Python package with easy-to-interpret visualizations. • LIME is available in the Language Interpretability Tool (LIT).	• The proximity measure uses a kernel smoothing that can be very sensitive to the kernel width. Slight modifications to the kernel width can lead to quite different explanations. • LIME provides an explanation based on a linear approximation of the local behavior of a model. It does not work as well for highly nonlinear models. • LIME doesn't take into account the sequential nature of the text data, and repeated words in a sentence are treated as a single feature. • The publicly available Python package implementation `lime` is only for text classification.

We already saw how to use LIME for image data in Chapter 4. One useful aspect of this technique is that it can also be applied to text (or tabular) data. However, it's important to understand how the LIME algorithm changes when working with text as opposed to images. LIME is a post hoc, local perturbation technique, so it provides interpretability for a trained model's prediction on a specific example by measuring how the model's predictions change with small perturbations to the feature inputs. By measuring how the predictions change under these local perturbations, LIME reflects the contribution of each feature of a data sample. More (or less) important feature values will have more (or less) influence on changing the model's predictions.

For images or tabular data, this process is fairly straightforward. Afterall, an image is represented by RGB pixel values, so we can perturb the feature of an image by turning pixel values on or off. In fact, the input image is segmented into interpretable regions or superpixels, and these components are changed to be gray. Similarly in tabular data, for continuous

feature values, we can sample values nearby to any given feature value. But the features for a text model are words. How does one perturb or add noise to a word? In fact, this could be achieved in a few ways.

How LIME Works with Text

As we discussed in "Overview of Building Models with Text" the word pieces of a sentence are represented by a token, which is then mapped to some embedding space. This embedding space essentially represents each word as a k-dimensional vector that can then be perturbed just as a continuous feature value as before by sampling other vectors nearby in the embedding space.

However, this formulation is problematic. LIME uses an exponential smoothing kernel to define a neighborhood of a data instance. The smoothing kernel defines the size of a neighborhood by measuring the proximity of points in relation to the given data instance. The smaller the kernel width, the closer points need to be in order to be considered within the neighborhood, whereas a larger kernel width will include instances farther away in the neighborhood.

However, when working with text, just because a vector is "close" to another in the vector space, this doesn't mean that the perturbed vector also actually represents another word "close" to the input. In fact, most likely it does not! Also, it's known that LIME is very sensitive to the kernel width. It is a subtle but very important hyperparameter when implementing LIME, since modifying the kernel width to be larger or smaller can result in completely different explanations.

Instead, we must go a bit further "upstream" and modify the words directly in a way more akin to the kind of example perturbations that are made when implementing LIME on images. That is, individual words are turned "on" or "off" to create a new, nearby example for inference.

Let's take as an illustration this snippet from an example in the IMDB dataset of 50K movie reviews: "If you like the original, you'll like this movie." This is the beginning of a positive review from the dataset. When passing this example text to the `LimeTextExplainer`, it is modified by replacing words with empty spaces, creating a distribution of nearby examples to pass to the model for prediction (see Figure 5-4).

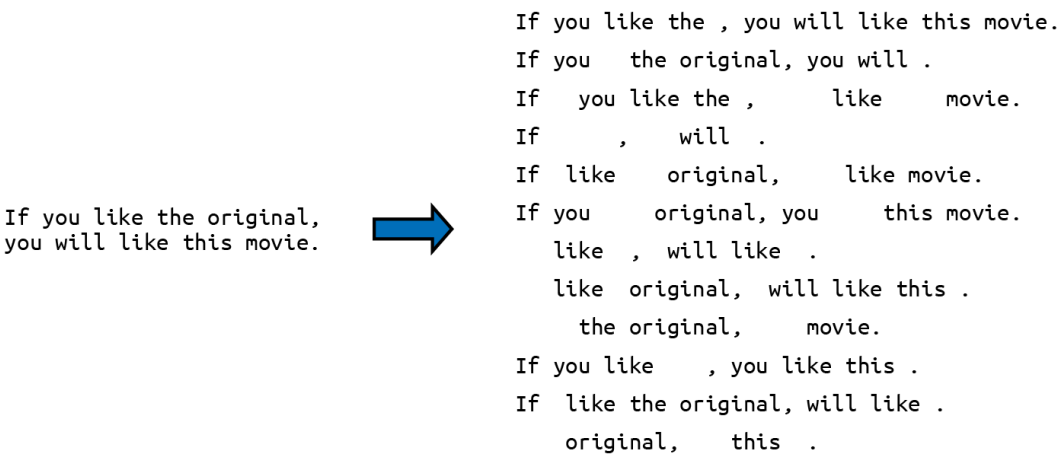

Figure 5-4. Given an input text for explanation, LIME creates new text to test the model predictions by randomly replacing words with spaces.

The next thing to consider when applying the LIME algorithm is how to measure the proximity of the perturbed inputs with the original example instance in question. This proximity is used to weight the generated samples to measure the influence of the features on the model's predictions. The public implementation of LIME uses as default an exponential smoothing kernel that takes the distance of two data instances and returns a proximity measure defined by:

$$\textbf{proximity}_{\kappa}(x_i,x_j) = \sqrt{e^{-d(x_i,x_j)^2/\kappa^2}}$$

where κ is the kernel width and d represents a distance metric; e.g., $d(x_i,x_j)$ could be the Euclidean distance between the vectors x_i and x_j.

CHOOSING THE KERNEL WIDTH IN LIME

One of the most sensitive hyperparameters in applying LIME is the choice of the proximity measure. This is true when applying LIME with text or with images or tabular data as well. This proximity is used to weight the generated samples, so the more nonlinear your model function is, the smaller you want the kernel width to be.

The default for text classification takes the distance to be the cosine distance and sets the kernel width at 25. However, the Python package for LIME[5] allows for users to modify the distance function d and the kernel width κ. You'll probably want to experiment with different values, though, and compare LIME explanations across different hyperparameters.

In the field of text mining there are various ways to measure the distance between two sentences; for example, the Euclidean distance, the cosine distance, Jaccard similarity, or a measure related to the proportion of words missing from the original. The default distance in LIME is the cosine distance that computes the cosine of the angle between the two sen-

tences expressed as vectors. The cosine distance of two sentences will range from 0 to 1; the larger the cosine similarity, the more words in common the two sentences have and the more similar they are. It is possible to change the distance metric when implementing LIME, and just as kernel width impacts the LIME explanations, so can the distance metric.

LIME then trains an interpretable model such as a logistic regression on these examples weighted by the proximity measure of the sampled instances. The weights of this linear model indicate the feature importance for each word. One nice thing about LIME is that it can be used as an interpretability technique for any model. The implementation treats the model as an opaque box.

Let's see how this works in practice for a text classification model trained on the IMDB dataset. Each example in this dataset is a movie review, not preprocessed in any way, and the corresponding label. The label is an integer value where 0 is a negative review, and 1 is a positive review. We'll train a TensorFlow model and use the `LimeTextExplainer` to explain a prediction by calling the `explain_instance` method as shown in the following code (see the notebook in this book's GitHub repository for the full code):

```
from lime.lime_text import LimeTextExplainer
explainer = LimeTextExplainer(class_names=['negative', 'positive'])
sample_text = "If you like the original, you will love this movie."
exp = explainer.explain_instance(text_instance=sample_text,
                                 classifier_fn=predict_prob,
                                 num_features=10)
```

The argument `num_features` indicates how many features (i.e., words) we'll use for the explanation and the `classifier_fn` is LIME's entry point to our model function. The classifier function must take a list of strings and output a two-dimensional array with prediction probabilities for each class. Since our TensorFlow model was built as a binary classification, we'll wrap our model in a function so that it has the appropriate signature:

```
def predict_prob(texts):
    preds = imdb_model.predict(texts)
    return np.concatenate((preds, 1-preds), axis=1)
```

Since 0 indicates a negative review and 1 indicates a positive review, for a given prediction `pred` for an example instance we return a tuple `(preds, 1-preds)`. The following code snippet calls `predict_prob` on the sample text "`If you like the original, you will love this movie.`", followed by the model's prediction:

```
predict_prob(["If you like the original, you will love this movie."])
> array([[0.04230487, 0.9576951 ]], dtype=float32)
```

Our model is 95% confident that this sample text is a positive review. We can then examine the output of the `explainer` as a list by calling `exp.as_list(label=1)` to see which words most contributed to the model's prediction with the target class 1:

```
[('you', 0.184102135),
 ('love', 0.177897292),
 ('will', 0.135323892),
 ('movie', -0.043178052),
 ('original', 0.0382655165),
 ('this', -0.0227692068),
 ('the', 0.01647684757),
 ('If', 0.0088829997),
 ('like', 0.0060883934)]
```

Here the numbers represent the attribution score for each of the features, in this case, words. The absolute value of each score informally represents their relative importance to the model prediction or performance. The positive or negative sign indicates their positive or negative influence on the target class.

For example, the word *love* has an attribution score of 0.178 and the model predicts the target class of 1, that is, a positive movie review. This means the word *love* has a relatively large positive influence on the model predicting this statement as a positive review. On the other hand, the word *movie* has an attribution score of –0.04, which indicates this word has a negative, though relatively small, influence on the model predicting this statement as a positive review.

When calling the method `as_list(...)`, we include the argument `label=1` to tell the explainer that we want attribution scores that con-

tributed to the positive label. This is because our model predicted the positive class for this input. If instead we wanted to list attribution scores for the negative label class, we would have used `exp.as_list(label=0)`.

In the way LIME is implemented for text data, each word in a sentence is treated as an individual feature. LIME itself does not take into account the sequential nature of the words in a sentence. This means that repeated words have a single feature importance. For example, in the sentence "`If you like the original, you will love this movie.`", the word *you* appears twice in the sentence. However, their importance is measured in aggregate just once. Be aware as this can cause misleading results.

Here we see that *you* has the strongest positive influence on the model, followed by *love*. The attribution for *you* is perhaps a bit misleading, but the positive attribution for *love* makes sense. Overall, the top three features with positive influence are *you*, *will*, *love*, which, when viewed together, makes sense.

There are also visualizations that can be used to represent the feature attributions, and we can display these in the notebook as well. For example, the method `as_pyplot_figure(label=1)` visualizes the attributions for each feature of the local example as green and red bars on a bar plot. Green indicates a positive influence for the model prediction, and red indicates negative influence on the model prediction. The result for this example is shown in Figure 5-5.

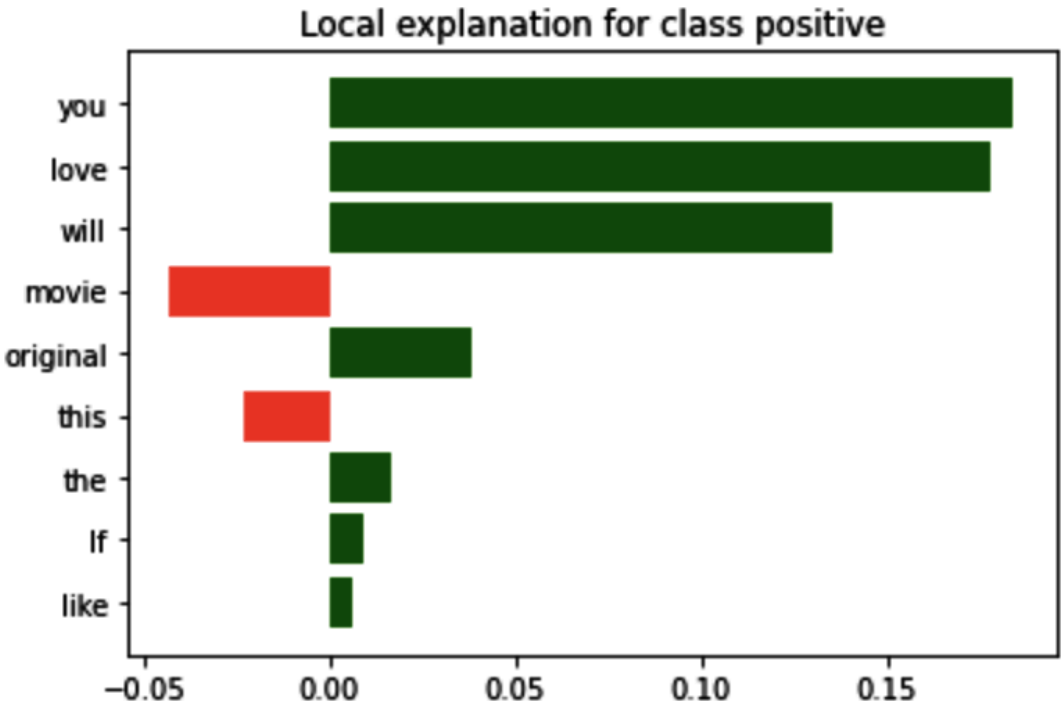

Figure 5-5. Local explanations can be visualized as a pyplot figure with green bars indicating positive influence for the model prediction (in this case the positive class) and red bars indicating negative influence for the model prediction.

We can also visualize these results within the notebook using `exp.show_in_notebook()` to see the influence of each feature value, as in Figure 5-6, both as a bar chart and as a heatmap overlaid on the original text.

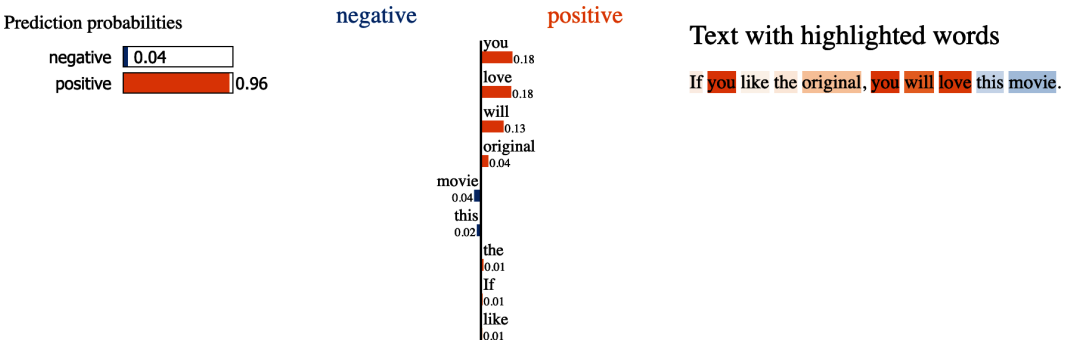

Figure 5-6. For this text example, the model predicts a positive sentiment. The LIME visualizations show that the words in the text that most contributed to this prediction are you, will, and love. (Print readers can see the color image at https://oreil.ly/xai-fig-5-6.)

Gradient x Input

Here's what you need to know about Gradient x Input:

- It's one of the simplest attribution methods, relying on gradients of the machine learning model. Gradient x Input can be referred to as "Grad cross Input," "Grad times input," or "Grad dot Input."
- Grad x Input is a saliency method and produces saliency scores proportional to the dot product of the gradient and the input.
- Saliency scores for word tokens can be positive or negative, indicating the influence that token had on the model prediction.
- Grad x Input was initially proposed as a technique to improve the sharpness of attribution maps generated by sensitivity analysis.
- Some studies suggest Grad x Input works better for BERT than it does for LSTM, based on its ability to find shortcuts in text classification tasks.[6]

Pros	Cons
• Easy and fast to implement for machine learning libraries like TensorFlow, PyTorch, and JAX; it can be applied to any differentiable model. • Gradient x Input has been shown to be the best performing explainability technique for transformers. • It is available in the Language Interpretability Tool.	• The gradients of a deep neural network can be, and typically are, noisy and sensitive to functions inputs. • Gradient x Input should be used in conjunction with other gradient-based techniques and especially sensitivity methods, like Grad L2-norm (also discussed in this section). • It requires differentiability of the ML model.

Gradients x Input (often referred to simply as Grad x Input) is a type of gradient-based attribution method. It's a favorite among practitioners because it's fast, easy to implement, and, particularly for transformers, known to perform well across various text classification datasets.[7]

More specifically, Grad x Input is a saliency method of explainability (later in this section, we discuss the difference between saliency and sensitivity methods). When working with text models, the salience scores for a certain word are proportional to the dot product of the word embeddings and the gradient of the model function with respect to that input. This gives a directional score for each input feature (i.e., word token). A positive score can be interpreted as that token having a positive influence on the prediction, while a negative score indicates that the prediction would be stronger if that token was not present.

Intuition from Linear Models

Suppose we have a simple linear model to estimate a baby's weight y in pounds based on two input variables: the mother's age x_1 and the duration of the pregnancy in days x_2. Using a linear regression model, we would learn a function:

$$y = w_0 + w_1 x_1 + w_2 x_2 + \epsilon$$

where the w_i's are the model weights and bias and ϵ is the residual error. If we learn through the data model parameters $w_1 = 0.1$ and $w_2 = 0.02$, then we have a clear explanation of the model's predictions using these model weights (to keep things simple, we'll take a zero bias $w_0 = 0$). The attribution for each input feature is simply the partial derivative of the target variable y with respect to the input feature x_i:

$$\textbf{feature attribution for mother's age} = \frac{\partial y}{\partial x_1} = w_1 = 0.1$$

$$\textbf{feature attribution for gestation days} = \frac{\partial y}{\partial x_2} = w_2 = 0.02$$

Since the feature attribution for the mother's age is substantially larger than that of gestation days, you might think it's obvious that the mother's age plays a more important role when predicting baby weight using our model.

But let's take an explicit example. Suppose we had a 30-year-old mother who had carried the baby for 40 weeks. In this case, our model would predict a baby weight of 8.6 pounds. Due to the linearity of the model, we can see that the final prediction for the baby weight is explained as the sum of the effect from the two input variables: 3 pounds from the mother's age and 5.6 pounds from the gestation period. That is to say, for this local explanation, we get the following feature attributions:

$$\textbf{feature attribution for mother's age} = x_1 \cdot \frac{\partial y}{\partial x_1} = (30)\,(0.1) = 3$$

$$\textbf{feature attribution for gestation days} = x_2 \cdot \frac{\partial y}{\partial x_2} = (280)\,(0.02) = 5.6$$

The feature attributions have flipped. Now the more important feature is the gestation period for the pregnancy. We have two possible explanations for the model's prediction and both are based on the gradient of the model function. One involves the gradient alone, and the other involves multiplication of the gradient at the input value.

Both methods described in the example are valid. They illustrate the differences between sensitivity and saliency methods. The first approach involving just the gradient is a sensitivity method. The second method, involving the gradient and the input value, is a saliency method and illustrates exactly the intuition behind the Grad x Input explainability technique we'll discuss in this section.

From Linear to Nonlinear and Text Models

Using the intuition from linear models, we can better appreciate how Grad x Input is defined. The attribution for a feature input is computed by taking the signed gradient of the model prediction function at a given input and multiplying component-wise with the (embedded) input itself.

To make this more precise with an example, suppose we have a text classification model to predict the sentiment of a given review from the IMDB dataset of movie reviews. Let's take the review "`If you like the original, you'll love this movie.`" and suppose our model predicts (correctly) a positive sentiment for this input instance. After tokenizing and embedding the words in the review, we have a sequence of vector embeddings we'll represent by $\boldsymbol{x}_{1:n}$ where $n = 10$ since there are 10 words in the review. If we let $\boldsymbol{f}$ represent the model function, the attribution for the $\boldsymbol{i}$-th word in the review is given by:

$$\nabla_{x_i} f\left(x_{1:n}\right) \cdot x_i$$

The dot product of these two vectors will give a scalar value. This scalar can be positive or negative, depending on the influence the input word embedding $\boldsymbol{x_i}$ has on the model prediction function $\boldsymbol{f}$.

Grad L2-norm

Grad L2-norm is closely related to Grad x Input. The idea is that we want a single scalar value to provide a relevance score for each feature (word) input. Taking the dot product between the gradient and the input produces such a scalar value. However, there are, of course, other ways to obtain a scalar value. Another commonly used approach is to take the L2-norm. Just as with Grad x Input, we compute the gradient of the neural network with respect to the inputs (note the subscript $\boldsymbol{x}$ in the gradient).

Just as we use auto differentiation for backpropagation during training to update the parameter weights, we can also compute the gradient with respect to the inputs. Thus the Gradient L2-norm indicates how much a change in a word's embedding affects the output of the model. The smaller the gradient, the less sensitive the model function is to small changes in the input. The larger the gradient, the steeper the model function and the more sensitive it is. Figure 5-7 illustrates this idea in one dimension.

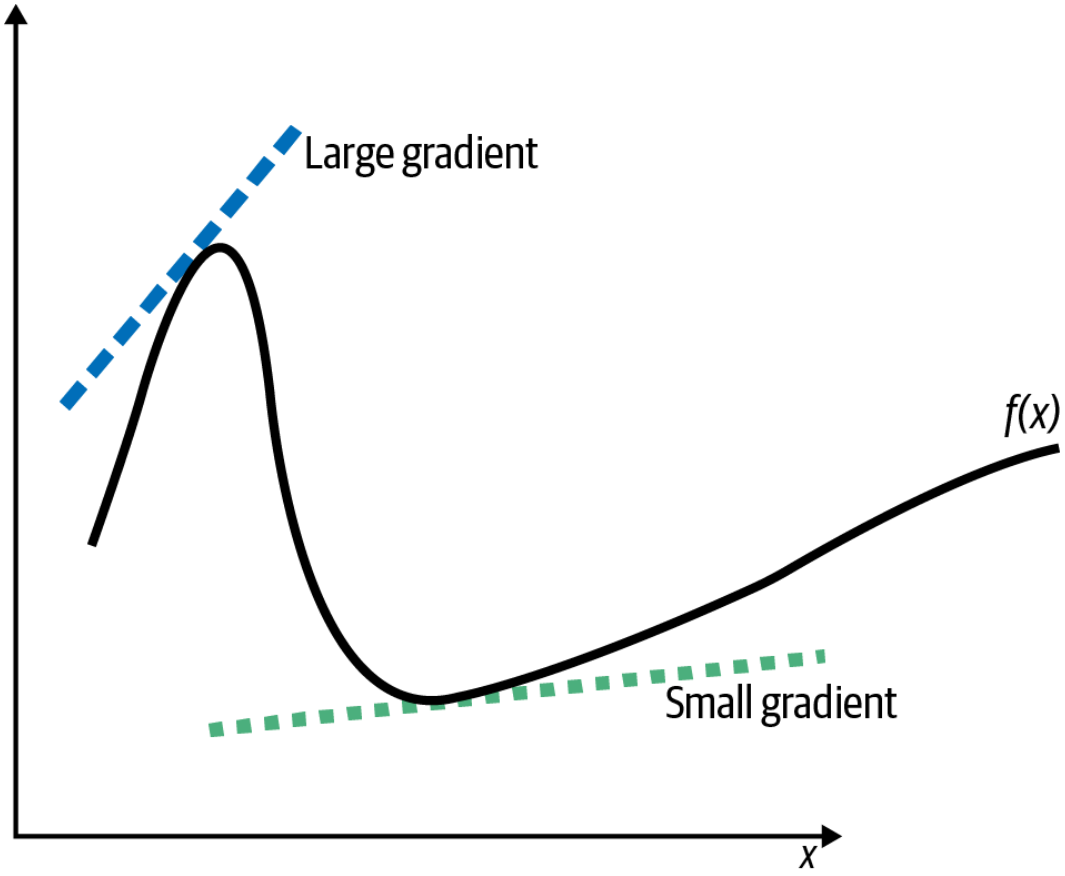

Figure 5-7. In this one-dimensional example, we see that the larger the gradient, the steeper the function. The smaller the gradient, the flatter the function, indicating the function is less sensitive at that input value.

Using the same notion as in the previous section, we can express the Gradient L2-norm as:

$$\nabla_{x_i} f(x_{1:n}) \cdot x_i$$

that is, the L2-norm of the gradient of the model function with respect to the inputs. Recall, the x_i here are the word embeddings. So, the Gradient L2-norm indicates how much a change in a word's embedding affects the output of the model. Note also, in comparison with the Grad x Input, the L2-norm is essentially the dot product of the gradient with itself and so lends itself to a similar interpretation. One caveat of Grad L2-norm is that because it is a norm measure, the saliency value is unsigned and always positive. However, it is still a common technique and often used when working with text.

Comparing sensitivity and saliency methods

There are two broad categories of explainable attribution methods: sensitivity methods and salience methods. A *sensitivity* method describes how the output of the network changes when one or more input features are perturbed. For differentiable models, this can be represented by the derivative or gradient of the prediction function with respect to the input features. This gives a first order Taylor expansion of a nonlinear function

so, just as Taylor series are accurate within a small neighborhood of an input, a sensitivity measure for a nonlinear function is only accurate within a very small perturbation. Since Grad L2-norm depends only on the gradient of the model function, it is a sensitivity method.

A *saliency* method, on the other hand, describes the marginal effect of a feature on the model prediction. Grad x Input is a saliency method because the attribution is computed by taking the (signed) partial derivatives of the output with respect to the input and multiplying them feature-wise by the input itself. The saliency score that is produced describes the contributions of the different input variables to the final model output. Thus, saliency methods take into account the input as well as the model gradient when computing the feature attributions.

As the example involving baby weights illustrates, the two methods are equally valid but answer different questions and show that the two attribution methods cannot be directly compared.

Layer Integrated Gradients

Here's what you need to know about Layer Integrated Gradients:

- Layer Integrated Gradients (LIG) is a variation of Integrated Gradients, but it focuses on a single layer of the network instead of input features; LIG provides attributions with respect to layers inputs or outputs.
- Similar to Integrated Gradients, LIG is a gradient-based attribution method.
- The implementation is available using the Captum library.

See also the pros and cons of Integrated Gradients in Chapter 4.

Pros	Cons
• LIG is useful for text to isolate the embedding layer for computing Integrated Gradients. • It can be applied to any differentiable model for any data type, images, text, tabular, etc. • There is an easy and intuitive implementation that even novice practitioners can apply.	• LIG requires differentiability of the model and access to the gradients, so it does not apply well to tree-based models. • The results can be sensitive to hyperparameters or choice of baseline.

As stated above, LIG is a variation on the classic IG method. The need for variation is in how text models are built. As we discussed in the beginning of this chapter, when working with text data, the text must be preprocessed to be fed to a machine learning model. The first step of preprocessing is tokenization. These tokens are then converted to categorical IDs and passed to an embedding layer. These embeddings are numeric vector representations of each word token.

The issue that arises is the step from tokenization to categorical IDs. Integrated Gradients is a gradient-based attribution method, but this step is not differentiable. Of course, there is a natural solution. In order to explain text features via Integrated Gradients, we need to compute the gradients with respect to the embeddings, not the token indices. One solution is to use Layer Integrated Gradients.

A Variation on Integrated Gradients

In Chapter 4, we discussed how the method of Integrated Gradients works for image-based models. At a high level, the method of Integrated Gradients determines feature attributions for a model's prediction by measuring how the model gradient with respect to the feature inputs changes along an integral path in the input feature space. More precisely, this integral path is a straight line from a given baseline to the input example, and the gradients are cumulated (integrated) over the straight-line path. This produces feature attributions for the input features, which in

the case of images, can be represented as a saliency map indicating which pixel most contributed to the model's prediction.

The main difference in LIG is that where IG assigns an importance score to the model's input features, Layer Integrated Gradients assigns an importance score to a given layer's inputs (or outputs); i.e., it allows us to apply Integrated Gradients on the interior of the model. Layer Integrated Gradients is a layer attribution where Integrated Gradients is a feature attribution method, as depicted in Figure 5-8.

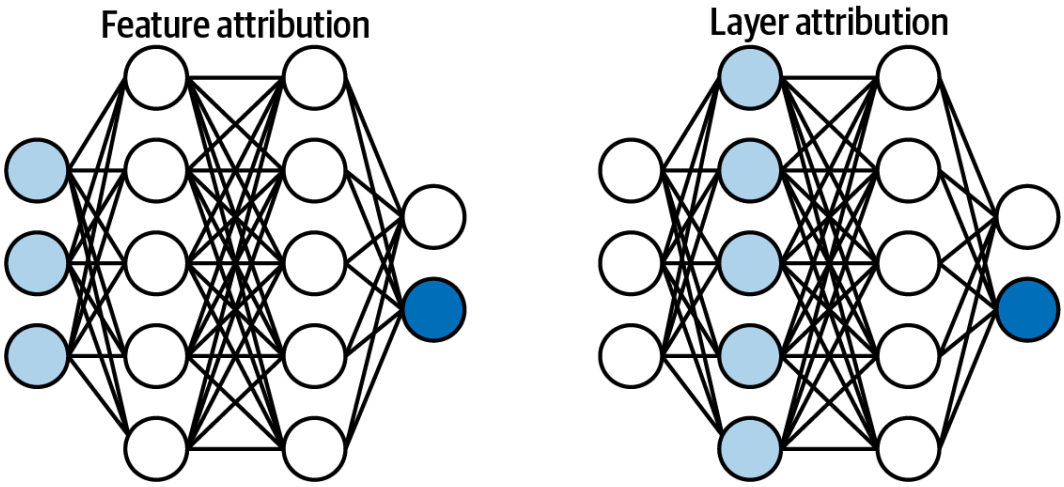

Figure 5-8. Layer Integrated Gradients is a variation of classic Integrated Gradients. They differ in where they measure attributions (the light shaded circles) of the model. For Integrated Gradients, attribution of the model output (the dark shaded circles) is assigned to the features, whereas for Layer Integrated Gradients, the attribution is assigned to internal layers of the model.

The other consideration when implementing Integrated Gradients is the choice of baseline. We saw in Chapter 4 that when using Integrated Gradients for images, the baseline plays an important role. The same is true for text. When working with text, a common baseline is just a sequence of the zero embedding vectors or, at the word level, a sequence of zero tokens. The zero embedding vector is a vector of all zeros with the same dimension of the embedding space. This is the baseline recommended in the original paper on Integrated Gradients.[8] During training, unimportant words tend to have small norms, and since the baseline is meant to represent an information-less feature, the all-zero input vector is a natural choice, even though it doesn't actually represent a valid input for the model in the same way an all-black image would for an image model.

The zero token represents any token or value you would use when padding a sequence. This makes the zero token a good candidate to use as a baseline. For example, when working with the BERT tokenizer, the pad token is simply a zero `PAD`. Therefore, the baseline sequence for the text "`If you like the original, you'll love this movie.`" would be

`[CLS, PAD, PAD, PAD, PAD, PAD, PAD, PAD, PAD, PAD, PAD, SEP]` or, when converted to integer tokens, `[101, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 102]`.

There is a nice implementation of Layer Integrated Gradients (among many other layer attribution methods) available in the PyTorch Captum library. One very useful library built on top of Captum is the Transformer Interpret tool designed to work specifically with the Hugging Face `transformers` package.

For example, suppose we want to use a BERT sentiment classifier to classify movie reviews from the IMDB database. BERT is a transformer-based natural language model released in 2018 by Google AI Language. The Transformers Interpret tool has an easy-to-use interface to provide a saliency map built on top of Captum's Layer Integrated Gradients. As when building any text model, we start with tokenization. To do this, we'll use a pretrained tokenizer that was used for training the BERT model. We can also use a pretrained BERT model. The following code block shows how to load the pretrained BERT tokenizer and pretrained BERT model for sequence classification. The `predict` function shows how to call this model to make classification predictions given an input text sequence:

```
from transformers import BertTokenizer

# load tokenizer
tokenizer = BertTokenizer.from_pretrained('./model')
# load model
model = BertForSequenceClassification.from_pretrained('./model')
def predict(inputs):
   return model(inputs)[0]
```

To generate explanations of this model using Layer Integrated Gradients, we'll use the `LayerIntegratedGradients` class in Captum. This is done by specifying the layer where you want to compute the attributions, as shown in the following code block. Technically, we can use any layer in our model. Here, we'll use the BERT embeddings layer given by `model.bert.embeddings`:

```
from captum.attr import LayerIntegratedGradients
lig = LayerIntegratedGradients(custom_forward, model.bert.embeddings)
```

When instantiating the `LayerIntegratedGradients` class, we provide two arguments: a forward pass of the model and the layer for which we want to compute attributions. Here we set the forward pass of our model to be a custom function defined by:

```
def custom_forward(inputs):
    preds = model(inputs)[0]
    return torch.softmax(preds, dim = 1)[0][1].unsqueeze(-1)
```

This way, `custom_forward` takes as an argument the input IDs of a tokenized text (e.g., a movie review) and returns the model prediction as a probability of positive sentiment. We can then compute the attributions for the tokenized input text (see the Layer Integrated Gradients notebook in this book's GitHub repository for the full code):

```
lig_attributions, delta = lig.attribute(inputs=input_ids,
                                        baselines=ref_input_ids,
                                        n_steps=700,
                                        internal_batch_size=3,
                                        return_convergence_delta=True)
```

Given the input text "`If you like the original, you'll love this movie.`," our model classifies that statement as having positive sentiment with probability 0.998. We can also visualize the saliency map for the word importance for the model's prediction using the `Transformers Interpret` library. As shown in Figure 5-9, we see that the words that most contribute to the positive sentiment prediction were *love*, *this*, and *movie*.

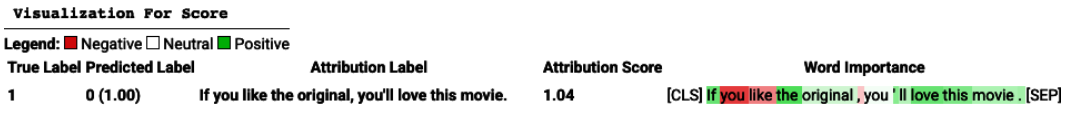

Figure 5-9. The visualization provided by `Transformers Interpret` and Captum indicates the word importance for the model's prediction. We see that phrases like love this movie have positive influence, but also (confusingly) if and the. (Print readers can see the color image at https://oreil.ly/xai-fig-5-9.)

However, the visualization also indicates a positive influence from the words *if* and *the*, which is somewhat less convincing. Also, the model appears to give a negative importance to *you* and *like*. Although this could indicate that there are more improvements to be made on our sentiment classification model, it's also a good reminder that explainability results

should be taken in context. In Chapter 7, we'll discuss some of the considerations you should keep in mind as a human interacting with explainability.

CONDUCTANCE

Conductance is another notion of attribution that is often considered in conjunction with feature attribution and layer attributions. The idea of conductance was introduced to extend the idea of attribution to better understand the importance of hidden units. Conductance is meant to be a natural refinement of Integrated Gradients. Instead of measuring the importance of a feature, conductance measures the importance of a single hidden unit to a model's prediction (see Figure 5-10).

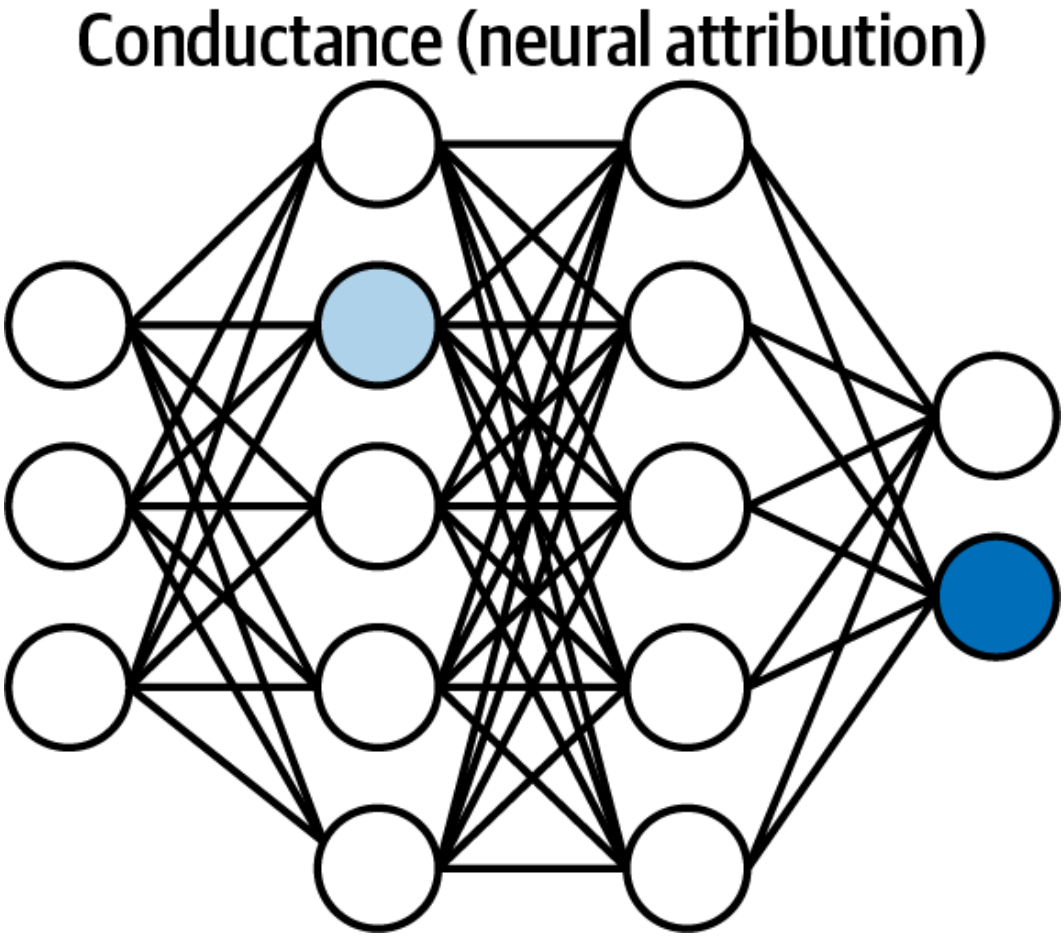

Figure 5-10. The conductance at a specific neuron (the light shaded circle) in a hidden layer is computed in a similar way to how Integrated Gradients compute attribution for a feature input.

Layer-Wise Relevance Propagation (LRP)

Here's what you need to know about Layer-Wise Relevance Propagation:

- LRP is a saliency method of explainability that measures the attributions of a given layer by backpropagating the contributions of all neurons in a layer to the neurons of the previous layer.
- At each layer, a special backward pass backpropagates the top-level relevance given by the model prediction to the inputs to that layer. The relevance is redistributed all the way back to the input layer.
- For models with only ReLU activations and max pooling nonlinearities, LRP is equivalent to the Gradient x Input method.

Pros	Cons
• The implementation is very modular. That is, each type of layer has its own propagation rules (there are different rules for redistribution for feed-forward layers and LSTMs), and different propagation rules can be applied for different layers in the network. • This can be applied to various data types (images, text, audio, video, etc.) and neural architectures (DNNs, CNNs, and LSTMs). • LRP has been shown to work better than gradient-based methods on text classification tasks.	• LRP requires implementing a custom backward pass, and the relevance is redistributed differently for feed-forward layers versus LSTMs. • Attribution values tend to concentrate on only a few features, which can inhibit performance for less common words. • Depending on your use case, you may want to combine different propagation rules for the best results. This can require some extra experimentation and hyperparameter tuning.

LRP is one of the most commonly used explainability techniques because it can be applied to various data types (not just text) and works well across various model architectures, such as DNNs, CNNs, and LSTMs. The purpose of LRP is to provide an explanation of any neural network's output in the domain of its input.

LRP is similar in spirit to DeepLIFT (Deep Learning Important FeaTures). DeepLIFT was introduced in 2016; it is a method for decomposing the output prediction of a neural network by backpropagating the contributions of all neurons to every feature of the input. The activation of each neuron is compared to its "reference activation," and the attribution scores for each neuron are determined by the difference between the activation and the reference. This allows for positive and negative contributions for any neuron.

How LRP Works

As the name suggests, LRP is considered a propagation explainability technique. This means that the feature attributions are achieved by recursively propagating some measure of the model's relevance for a prediction back through the network and to the instance's input features. At each level of the network, the propagation procedure must satisfy a conversation property so that whatever has been received by a neuron from a higher layer has to be fully redistributed to the neurons of the previous layer. This way the total relevance from upper-level layers to lower-level layers remains constant (see Figure 5-11).

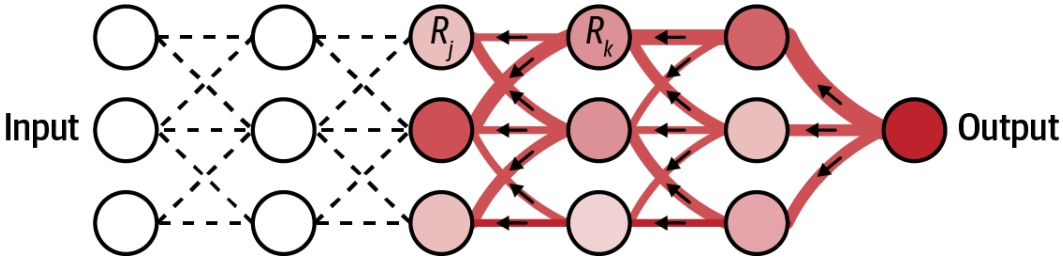

Figure 5-11. The top-level relevance is redistributed to the lower layers of the model so that whatever has been received by a neuron from a higher layer has to be fully redistributed to the neurons of the previous layer.

In Figure 5-11, let R_j and R_k denote the relevance of the neurons j and k located at layer l and $l+1$, respectively. The relevance score for neuron j is determined by propagating the relevance from every neuron in level k onto the neurons in level j according to the rule:

$$R_j^{(l)} = \sum_k \frac{z_{jk}}{\sum_j z_{jk}} R_k^{(l+1)}$$

The quantity z_{jk} is a general contribution measure that measures how much neuron j contributes to neuron k. When implementing LRP, you can determine how to compute this quantity. Defining z_{jk} can take some experimentation, though there are commonly used implementations and formulas that you can use as a starting point, as we'll soon see. The important thing to note is that when summing both sides of the equation

with respect to j, a conservation property is satisfied; namely, the sum of the relevance for all neurons in a layer is constant.

Let's look at a familiar example from text classification. Suppose we had the input text from the IMDB dataset of movie reviews, "`If you like the original, you'll love this movie.`" Each word is mapped to an embedding layer to create a sequence of n vectors $x_{1:n}$. This is then passed to the model to determine the top-level relevance $f(x_{1:n})$. The aim of LRP is to propagate this relevance score, the model's output, back through the individual neurons and layers of the model with the rule that at each layer the total relevance is maintained, as in Figure 5-12.

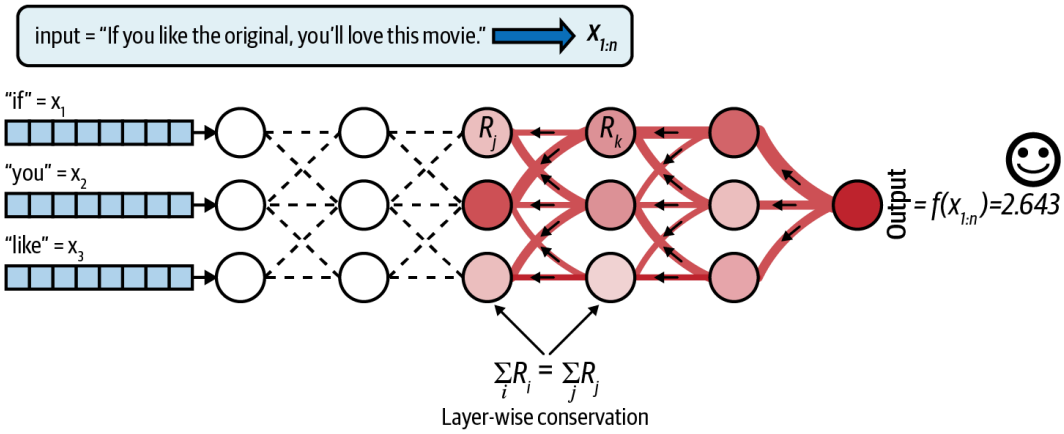

Figure 5-12. Each layer preserves the total relevance of the model's prediction on the input instance. This top-level relevance is propagated back through the network and ultimately to the model inputs to assign feature attributions to each feature (i.e., word) input.

The way that relevance of a neuron is distributed to its contributing neurons from the previous layer (i.e., defining the contribution measure z_{jk}) can be defined according to different schemes. It's even possible to have different propagation rules for different layers. For deep neural networks with ReLU activation functions, the recommended contribution measure is defined by $z_{jk} = x_j w_{jk}$, where x_j are the input values and w_{jk} are the parameter weights. In this setting, the basic rule for propagation then becomes:

$$R_j^{(l)} = \sum_k \frac{x_j w_{jk}}{\sum_j x_j w_{jk}} R_k^{(l+1)}$$

This is often referred to as the LRP-0 rule and is arguably the simplest rule you can use for Layer-Wise Relevance Propagation (the reason for the zero in the name will become evident in the next paragraph).

One way to improve upon LRP-0 is by incorporating a constant ϵ in the denominator. This is done for two main reasons. First, adding a small constant to the denominator helps in capturing the relevance when the contributions of an input neuron are small. With larger values of ϵ, only

the most important explanation factors still make enough of a difference to be recognized. Second, from a computational perspective, the ϵ helps make the fraction more stable. The LRP-ϵ rule is given by:

$$R_j^{(l)} = \sum_k \frac{x_j w_{jk}}{\epsilon + \sum_j x_j w_{jk}} R_k^{(l+1)}$$

Thus, when $\epsilon = 0$, we have the original LRP-0 rule from before. The LRP-ϵ propagation rule can help in creating explanations that are less noisy and sparser in terms of input features.

One of the advantages of LRP is that the propagation rules can be implemented efficiently and modularly. Different LRP rules can be applied at different levels of the network. For example, at the upper layers it's better to use the basic LRP rule (with $\epsilon = 0$) to have a propagation rule close to the function and its gradient. While in the middle layers, it helps more to use LRP-ϵ for some positive ϵ value to help filter out spurious variations within the network and highlight only the most salient features. There are other propagation rules that are mentioned in the original paper,[9] if you are interested in exploring others, but LRP-0 and LRP-ϵ will get you pretty far.

The relationship between LRP and Grad x Input

It's interesting to note that when all the activation functions of a deep neural network are ReLU functions, the Layer-Wise Relevance Propagation method is equivalent to Grad x Input.[10]

ReLU stands for Rectified Linear Unit and is a piecewise linear function that outputs the input when the input is positive and returns zero otherwise (see the graph in Figure 5-13).

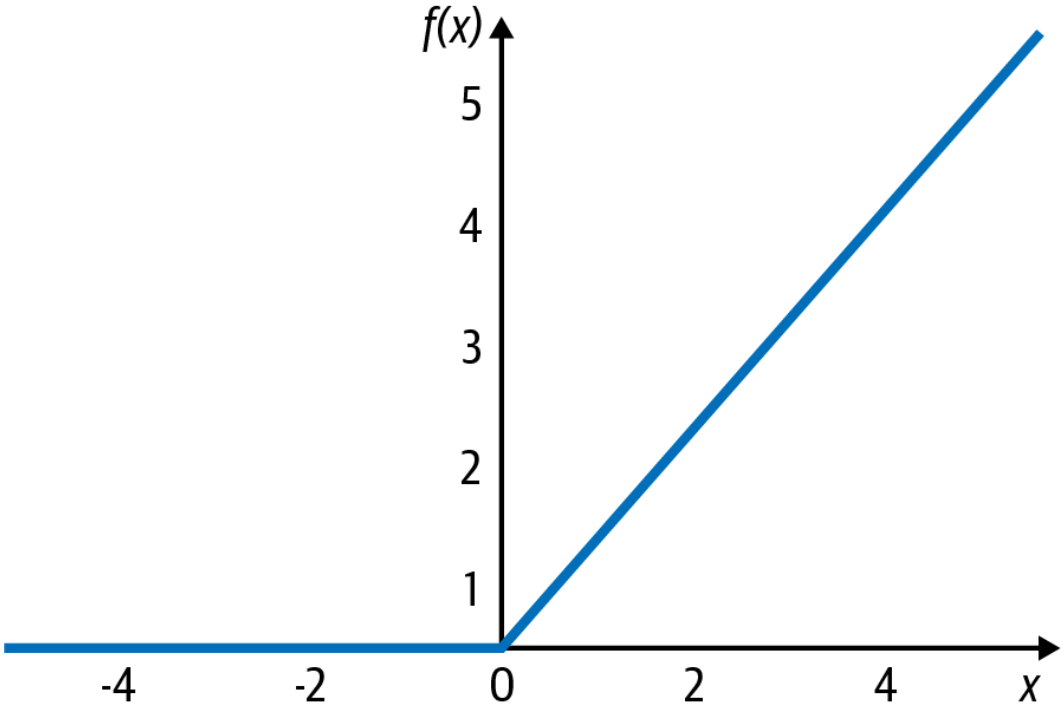

Figure 5-13. When ReLU is inactive, Grad x Input is zero. When ReLU is active, Grad x Input is equal to the output.

As shown in the graph, for negative values, the ReLU activation is equal to zero and the gradient is zero. So, when ReLU is inactive, the Grad x Input is zero and LRP assigns zero signal to that unit. On the other hand, when ReLU is positive and the unit is active, the gradient is equal to one and Grad x Input is equal to the output, and LRP assigns full signal to the unit.

Deriving Explanations from Attention

Since the transformer architecture was introduced in 2017, *attention* mechanisms have played an increasingly important role in natural language models and can achieve state-of-the-art results for almost all NLP tasks. Attention mechanisms were introduced to address the shortcoming that arose when training more traditional encoder-decoder sequence models, particularly when decoding long sequences.

As the name suggests, a traditional encoder-decoder model consists of two components, the encoder and the decoder. The encoder steps through each time step of the input sequence and encodes the input into a fixed-length context vector. The decoder then takes that context vector and decodes it into a meaningful output sequence, one time step at a time. Encoder-decoders are a useful architecture for sequence-to-sequence models, but they struggle when decoding long sequences. Attention addresses this issue by keeping track of the hidden state for each time step of the input sequence, rather than encoding the entire input sequence into a single context vector. With attention, the context vector *plus* all of the input sequence hidden states are passed to the decoder at each time

step during the decode step. This way, attention allows the decoder model to focus on different parts of the input sequence at each stage of decoding. This helps to preserve the input sequence context, particularly when decoding long input sequences.

Attention now dominates the field of text processing and has even become popular in computer vision tasks as well. Given their wide range of applications and impressive results, there have also been attempts to crack open the attention mechanism of transformers to derive some form of explanation into the model's predictions.

The heart of the transformer is the self-attention block. Self-attention is similar to the attention mechanism we just described for encoder-decoder models, but it also applies attention to elements of the input sequence, allowing the encoder to look at other words in the input sequence as it encodes a word at a certain time step. So, attention is used by the decoder when decoding the context received by the encoder, while self-attention is used by the encoder to create the context the encoder produces.

Self-attention layers assign a pair-wise attention value to every two words (or tokens) in an input sequence. Since attention is ultimately a matrix of learned weights, it is possible to visualize a transformer model by examining the learned relevancy score for input features. A larger weight value indicates a stronger attribution for that feature. As an example, consider the task of machine translation. Given our sample input sentence "If you like the original, you'll love this movie.," our model returns the Spanish translation "Si te gusta la original, encantara esta pelicula." The alignment matrix between the input and target sentence, shown in Figure 5-14, illustrates which part of the input sequence is more or less important at each decoding step. Each row of the matrix indicates the learned weights associated with the annotations for that hidden state. The (i, j)-th element of the matrix indicates the weight of the annotation of the *j*-th input word for the *i*-th output word. Note that the alignment of words between English and Spanish is almost perfectly aligned word for word with strong weights along the diagonal of each matrix.

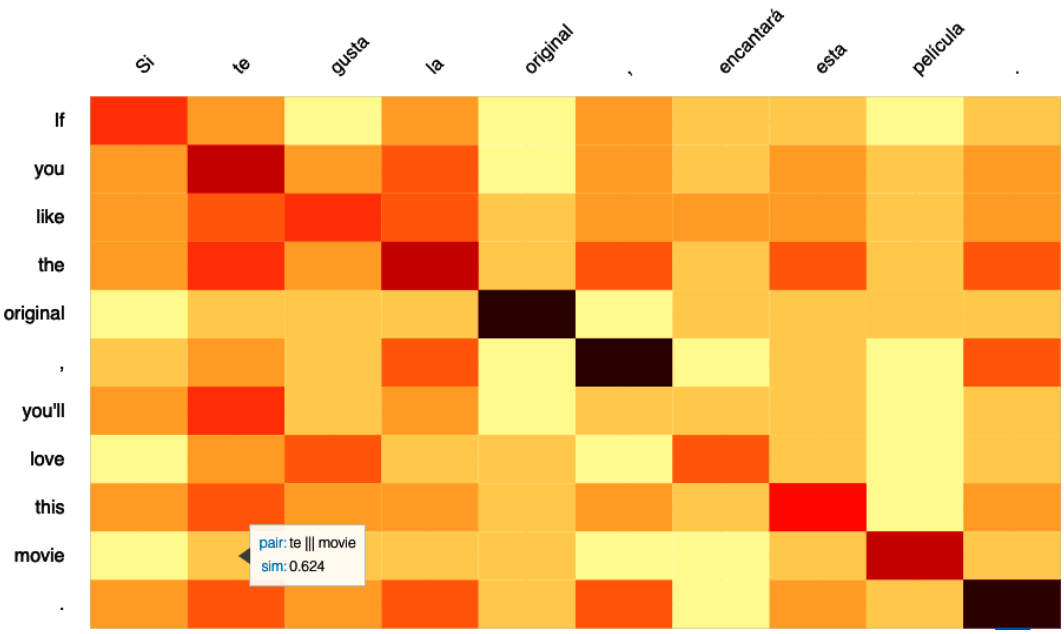

Figure 5-14. For machine translation, we can visualize how the model attends to different parts of the input and target sentence.

While we can visualize an alignment matrix for the single attention layer of an encoder-decoder model (as in Figure 5-14), this is much more difficult for a typical transformer. Transformers build on the idea of self-attention in an encoder-decoder, and a typical transformer will have multiple self-attention layers (and multiple attention heads) that are combined one after the other with nonlinearities in between, much like the fully connected layers of a deep neural network.

Be careful in trying to derive too much explanatory power from visualizing alignment matrices from attention. A self-attention head in a transformer model involves computation of queries, keys, and values, and when attention is applied over hidden states, there is inherently information about other time steps automatically mixed in. Any attempt to reduce self-attention to only values learned in the attention matrix doesn't capture the entire picture. In fact, some research shows that the attention weights that are learned through training can be uncorrelated with the results of gradient-based methods for determining feature importances.[11]

So, although there has been some work in visualizing attention weights to infer explainability or interpret which layers the model was focusing on for certain predictions, these visualizations should be taken with a grain of salt, and it's better to rely on saliency methods if you want quantitative importance scores.

One approach to deriving explanations for self-attention models is to borrow an idea of Layer-Wise Relevance Propagation (LRP) to compute scores for each attention head in each layer of the transformer. These

scores can then be integrated throughout the attention graph to provide a final feature attribution score.[12]

To see how this technique works in practice, let's create a text classification model that relies on transformer architecture and is trained using the IMDB movie review dataset. We'll use the Hugging Face `transformer` library to load a pretrained BERT model for sequence classification. Then, using the pretrained tokenizer, we can evaluate the model on IMDB movie reviews to predict a positive or negative sentiment, shown in the following code (the full code is available in the LRP notebook from this book's GitHub repository):

```
model = BertForSequenceClassification.from_pretrained("bert-base-uncased-SST-
tokenizer = AutoTokenizer.from_pretrained("textattack/bert-base-uncased-SST-2
# initialize the explanations generator
explanations = Generator(model)

classifications = ["NEGATIVE", "POSITIVE"]
```

Here, `Generator` is a class that is instantiated using the pretrained BERT model.[13] This class contains the methods we'll use to generate explanations using LRP. Let's take the sample review text "`If you like the original, you'll love this movie.`" We can tokenize and pass to our explainer using layer-wise propagation:

```
sample_txt = ["If you like the original, you'll love this movie."]
encoding = tokenizer(sample_txt, return_tensors='pt')

# true class is positive - 1
true_class = 1

# generate an explanation for the input
expl = explanations.generate_LRP(input_ids=input_ids,
                                 attention_mask=attention_mask,
                                 start_layer=0)[0]
```

Using Captum's library for visualization, we can explore the explanations that were generated (see Figure 5-15).

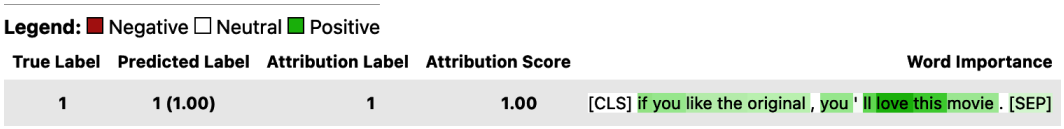

Figure 5-15. For this input example, the model predicts positive sentiment (the true label), and the word importance score using LRP creates a saliency mask over the words that contributed most to that prediction. (Print readers can see the color image at https://oreil.ly/xai-5-15.)

Which Method to Use?

So far, you've seen a number of explainability techniques that can be used when working with text data. You're probably wondering which is the best method and which one you should use for your next (or current) ML project. Of course, there is no clear answer to that question. If anything, we've tried more to illustrate the *what* and *how* of each technique and not claim to suggest that any one method beats out all the rest. They each have their pros and cons and, moreover, the technique that is best for you and your use case will depend heavily on your data, your model, and how you plan to use the explanations (see Chapter 7 where we discuss in detail human interaction aspects of explainability).

With that in mind, our advice would be to try various techniques and see which explanations ring the truest for you given what you know about your data, your model, and your use case. Of course, implementing even one of the techniques we discuss in this chapter could be a lot of development time and effort. We'll end this chapter by discussing the Language Interpretability Tool (LIT) developed by Google's People + AI Research (PAIR) team. LIT is an incredibly useful tool for any practitioner hoping to explain their NLP model and provides an easy-to-use interface that you can use to help you decide which explainability method to use.

Language Interpretability Tool

The Language Interpretability Tool (LIT)[14] is a visual, interactive model-understanding platform for NLP models developed by the PAIR group at Google. LIT is similar in spirit to the What-If Tool that allows ML practitioners the ability to analyze and debug their models and datasets without the need to write a lot of code. However, LIT is focused on the specific challenges that arise when working with NLP models and has an intuitive interface that allows you to explore and interact with your dataset and model predictions. The main workspace contains modules for exploration and analysis, including UMAP (uniform manifold approximation and projection) and t-SNE (t-distributed stochastic neighbor embedding) embeddings of your dataset, data tables, and editors to explore individual exam-

ples and a slicer editor that allows you to create and examine slices of interest in your dataset. The group-based workspaces can be set up to focus on performance, predictions, explanations, or counterfactuals; this is the workspace that we will primarily be focusing on.

Like LIT, the What-If Tool (WIT) was also developed by the PAIR group at Google. It is designed to provide an easy-to-use interface for exploring and understanding the predictions of classification and regression ML models. The plugin allows you to apply inference on a large subset of examples and visualize the result in a number of different ways.

You can explore how your model prediction would change if a feature value was different or examine how different features affect each model's prediction in relation to each other. You can examine global statistics of your dataset to uncover hidden biases or correlations among features. You can even explore slices of your dataset and evaluate model performance metrics on subgroups to check for model fairness.

The LIT platform has an extensive list of capabilities to assist practitioners in explaining their model's predictions, covering a wide range of NLP tasks from text classification to sequence generation. Many of the XAI techniques we discussed in this chapter are also available in LIT along with nice visualizations. For example, there are built-in modules for visualizing attention and saliency maps and features for aggregation metrics for your text samples. There is also support for counterfactual generation for model examples and side-by-side comparison for two different models to easily see how they differ or agree. There is a wide range of features, and we highly encourage you to go explore the functionality that is available. In terms of analyzing different explainability techniques, one of the nicest properties is that LIT allows you to easily compare the results of different explainability methods without having to develop and experiment with each one yourself.

There are a number of demos available that illustrate some of the capabilities of LIT. For example, let's look at the demo for sentiment analysis. This demo uses the Stanford Sentiment Treebank consisting of movie reviews, similar to the IMDB dataset we've seen already in this chapter. Let's provide a test sentence: "`If you like the original, you'll love this movie.`" Within the LIT UI, under the Predictions tab, we see that the model predicts a positive sentiment for this example with probability 0.979. There is a lot of functionality within the LIT platform that can be explored, but let's focus on the explanations for this input instance. LIT

currently supports a collection of explainability methods, including Grad x Input, Integrated Gradients, LIME, and Grad L2-norm. Clicking on the Explanations tab, we can see a visualization of the saliency maps for each of these methods (see Figure 5-16), allowing for quick and easy comparison of the different techniques. This way you can easily see which techniques most resonate with what you know about your data, your model, and your use case. A large discrepancy between different techniques might be cause for concern or suggest further analysis, whereas a similar consensus among techniques sends a clearer signal on the current state of the model.

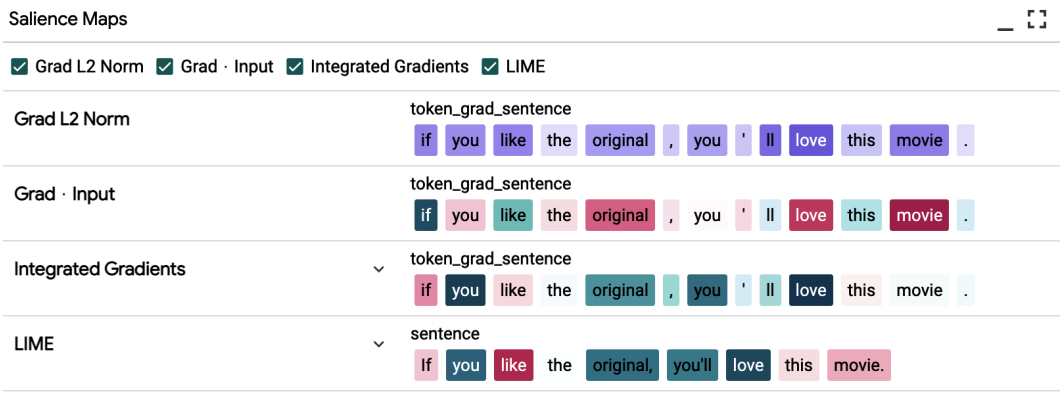

Figure 5-16. The Language Interpretability Tool allows for easy comparison of several commonly used explainability techniques, including many that we discussed in this chapter. The color intensity indicates salience with blue and purple, indicating positive attribution, and red indicating negative. (Print readers can see the color image at https://oreil.ly/xai-5-16.)

Summary

In this chapter, we focused on explainability methods for natural language models. We started with a brief overview of the nuances of working with text data and the role played by tokenizers and embedding layers. It's important to be aware of these preprocessing steps as their artifacts make their way downstream to explainability tasks later either directly or indirectly. We then turned to techniques, starting with LIME. LIME is a common explainability method and can be used for images, tabular data, and text; we saw in detail how LIME is implemented when working with text.

We then turned our attention to two related techniques: Grad x Input and Grad L2-norm. These two techniques have been shown to work well with transformer models, and they give a good comparison of sensitivity and saliency techniques in XAI. Next, we discussed Layer Integrated Gradients, which is really just an extension of the classic Integrated Gradients technique to provide layer attribution within a mode. This approach is particularly useful when working with text since examining a

model on the input token ID level is problematic because integer IDs are not differentiable. Instead, it's necessary to evaluate the Integrated Gradients at the embedding layer of the model.

Lastly, we looked at Layer-Wise Relevance Propagation, which provides a way to distribute the relevance score of a model's prediction to the internal neurons of the network. By propagating the relevance through the network, we can ultimately determine which input features contain more relevance for a given prediction. We then explored the Language Interpretability Tool as a platform for exploring features of NLP models and, in particular, explainability methods. LIT has a suite of capabilities and an interactive platform that can be run in a Jupyter notebook. It's a great way to apply the explainability techniques we discussed in this chapter to your own NLP model.

In the next chapter, we'll look at some other advanced techniques and recent trends in explainability that you might also like to add to your explainability toolkit.

1 Ashish Vaswani et al., "Attention Is All You Need," *Advances in Neural Information Processing Systems* 30 (2017).

2 For more information on GLUE, see *https://gluebenchmark.com*.

3 See "Pathways Language Model (PaLM): Scaling to 540 Billion Parameters for Breakthrough Performance" on the Google AI blog.

4 See this recent tweet from Hugging Face cofounder and CEO Clement Delangue.

5 See *https://github.com/marcotcr/lime* for more information on LIME.

6 Bastings et al., "Will You Find These Shortcuts? A Protocol for Evaluating the Faithfulness of Input Salience Methods for Text Classification," arXiv, 2021, *https://arxiv.org/abs/2111.07367*.

7 Pepa Atanasova et al., "A Diagnostic Study of Explainability Techniques for Text Classification," arXiv, 2020, *https://arxiv.org/pdf/2009.13295.pdf*.

8 Mukund Sundararajan et al., "Axiomatic Attribution for Deep Networks," International Conference on Machine Learning, PMLR, 2017.

9 Alexander Binder et al., "Layer-Wise Relevance Propagation for Deep Neural Network Architectures," Information Science and Applications (ICISA), Springer (Singapore, 2016): 913–22.

10 Avanti Shrikumar et al., “Not Just a Black Box: Learning Important Features Through Propagating Activation Differences,” arXiv, 2017, *https://arxiv.org/abs/1605.01713*.

11 Sarthak Jain and Byron C. Wallace, “Attention Is Not Explanation,” *Proceedings of NAACL-HLT 2019*, *https://aclanthology.org/N19-1357.pdf*.

12 Hila Chefer et al., “Transformer Interpretability Beyond Attention Visualization,” Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2021.

13 See *https://github.com/hila-chefer/Transformer-Explainability* for more information.

14 More information is available at *https://pair-code.github.io/lit*.