other hand, isolated `Stream` nodes can have any language. Since most node embedding algorithms encode isolated nodes identically, you would introduce noise to your classification model by including isolated nodes. Therefore, you would want to exclude all isolated nodes from the training and test datasets.

On the other hand, if you are dealing with isolated nodes and the relationships are not missing, meaning that no new relationships will be formed in the future, you can include isolated nodes in your workflow. For example, imagine you were to predict a person's net worth based on their network role and position. Suppose a person has no relationships and, therefore, no network influence. In that case, encoding isolated nodes could provide a vital signal to the machine learning model that predicts net worth. Always remember that most node embedding models encode isolated nodes identically. So if isolated nodes all belong to a single class, then considering them would make sense. However, if isolated nodes belong to various classes, then it would make sense to remove them from the model to remove noise.

9.3 The node2vec algorithm

Now that the graph is constructed, it is your job to encode nodes in the embedding space to be able to train the language prediction model based on the network position of the nodes. As mentioned, you will use the node2vec algorithm (Grover & Leskovec, 2016) for this task. The node2vec algorithm is transductive and can be fine-tuned to capture either homophily- or role-based embeddings.

9.3.1 The word2vec algorithm

The node2vec algorithm is heavily inspired by the word2vec (Mikolov et al., 2013) skip-gram model. Therefore, to properly understand node2vec, you must first understand how the word2vec algorithm works. Word2vec is a shallow, two-layer neural network that is trained to reconstruct linguistic contexts of words. The objective of the word2vec model is to produce word representation (vectors) given a text corpus. Word representations are positioned in the embedding space such that words that share common contexts in the text corpus are located close to one another in the embedding space. There are two main models used within the context of word2vec:

- Continuous bag-of-words (CBOW)
- Skip-gram model

Node2vec is inspired by the skip-gram model, so you will skip the CBOW implementation explanation. The skip-gram model predicts the context for a given word. The context is defined as the words adjacent to the input term.

Figure 9.7 visualizes how training pairs of words are collected in a skip-gram model. Remember, the objective of the skip-gram model is to predict context words or words that frequently co-appear with a target word. The algorithm creates training pairs for every word in the text corpus by combining the particular word with its adjacent words. For example, in the third row of figure 9.7, you can observe that the word

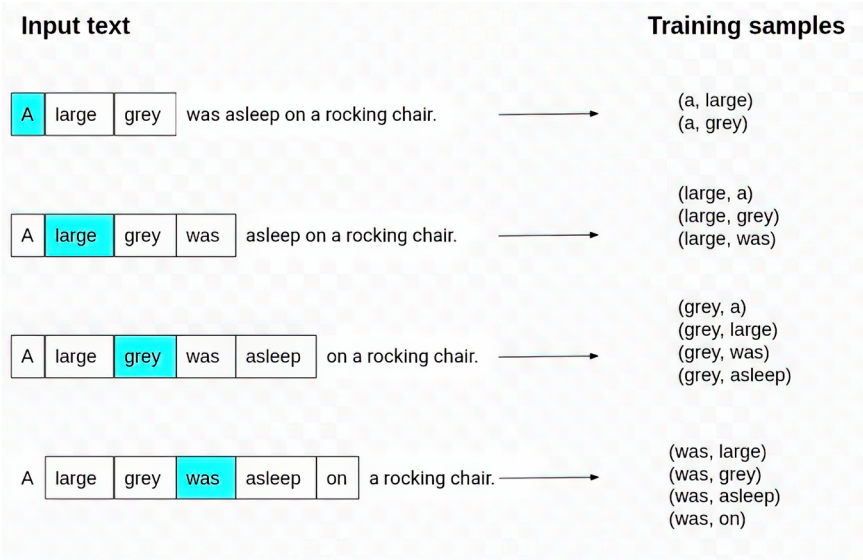

Figure 9.7 The process of predicting the language for new streams

grey is highlighted and defined as the target word. The algorithm collects training samples by observing its adjacent or neighboring words, representing the context in which the word appears. In this example, two words to the left and the right of the highlighted word are considered when constructing the training pair samples. The maximum distance between words in the context window with the target word in the center is defined as the *window size*. The training pairs are then fed into a shallow, two-layer neural network.

Figure 9.8 visualizes the word2vec neural network architecture. Don't worry if you have never seen or worked with neural networks. You simply need to know that during the training of this neural network, the input is a *one-hot-encoded* vector representing the input word, and the output is also a one-hot-encoded vector representing the context word.

Most machine learning models cannot work directly with categorical values. Therefore, one-hot encoding is commonly applied to convert categorical values into numerical ones. For example, you can see that all the distinct categories in figure 9.9 transformed into columns through the one-hot-encoding process. There are only three distinct categories in figure 9.9, so there are three columns in the one-hot-encoding output. Then, you can see that the category `Blue` is encoded as 1 under the `Blue` column and 0 under all other columns. Essentially, the numerical representation of the category `Blue` is [1,0,0]. Likewise, the numerical representation of `Yellow` is [0,0,1]. As you can observe, the one-hot-encoded vectors will have a single 1 under the column of the category they belong to, while the other elements of the vectors are 0. Consequently, the one-hot-encoding technique assures that the Euclidean distance

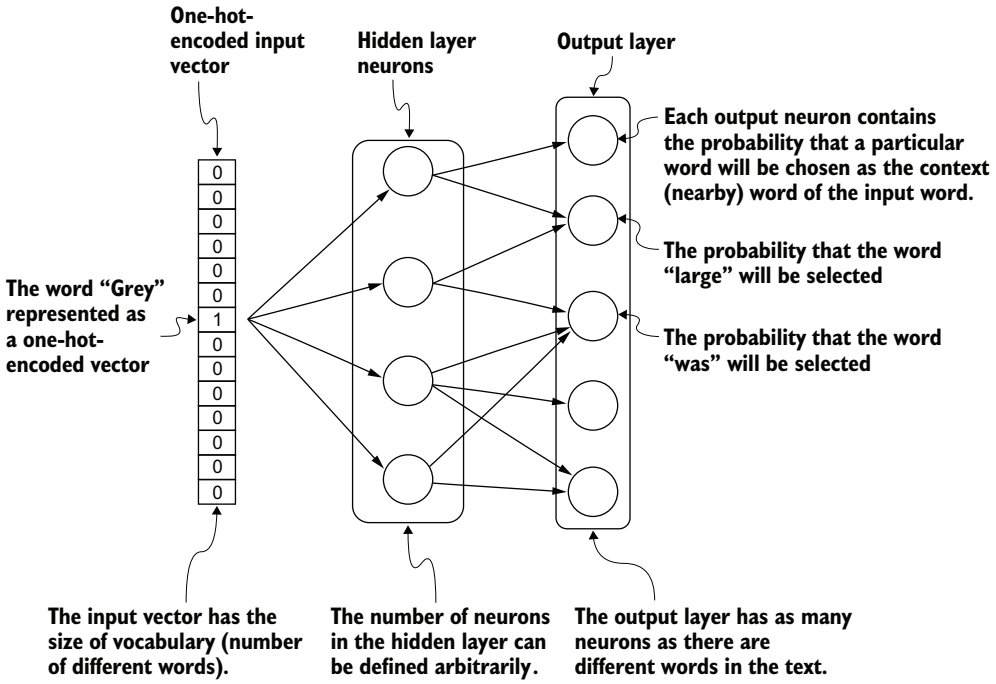

Figure 9.8 Word2vec shallow neural network architecture

between all classes is identical. While this is a straightforward technique, it is quite popular, as it allows the simple transformation of categorical values into numerical ones, which can then be fed into machine learning models.

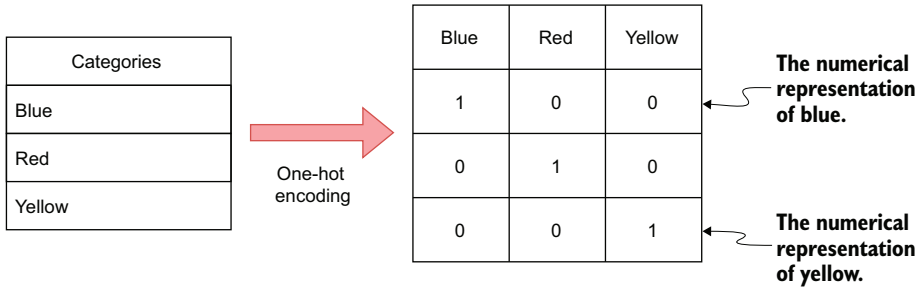

Figure 9.9 One-hot-encoding technique transforming categorical values into numerical values

After the training step of the skip-gram model is finished, the neurons in the output layer represent the probability a word will be associated with the target word. Word2vec uses a trick where we aren't interested in the output vector of the neural network but

have the goal of learning the weights of the hidden layer. The weights of the hidden layer are actually the word embedding we are trying to learn. The number of neurons in the hidden layer will determine the *embedding dimension* or the size of the vector representing each word in the vocabulary. Note that the neural network does not consider the offset of the context word, so it does not differentiate between directly adjacent context words to the input and those more distant in the context window or even if the context word precedes or follows the target term. Consequently, the window size parameter has a significant influence on the results of the word embedding. For example, one study (Levy, 2014) found that larger context window size tends to capture more topic or domain information. In contrast, smaller windows tend to capture more information about the word itself (e.g., what other words are functionally similar).

9.3.2 Random walks

So what does word2vec have to do with node embeddings? The node2vec algorithm uses the skip-gram model under the hood; however, since you are not working with a text corpus in a graph, how do you define the training data? The answer is quite clever. Node2vec uses *random walks* to generate a corpus of "sentences" from a given network. Metaphorically, a random walk can be thought of as a drunk person traversing the graph. Of course, you can never be sure of an intoxicated person's next step, but one thing is certain. A drunk person traversing the graph can only hop onto a neighboring node.

The node2vec algorithm uses random walks to produce the sentences, which can be used as input to the word2vec model. In figure 9.10, the random walk starts at node A and traverses to node H via nodes C, B, and F. The random walk length is decided arbitrarily and can be changed with the *walk length* parameter. Each node in the random walk is treated as a word in the sentence, where the size of the sentence is defined with the walk length parameter. Random walks start from all the nodes in the graph to make sure to capture all the nodes in the sentences. These sentences are then passed to the word2vec skip-gram model as training examples. That is the gist of the node2vec algorithm.

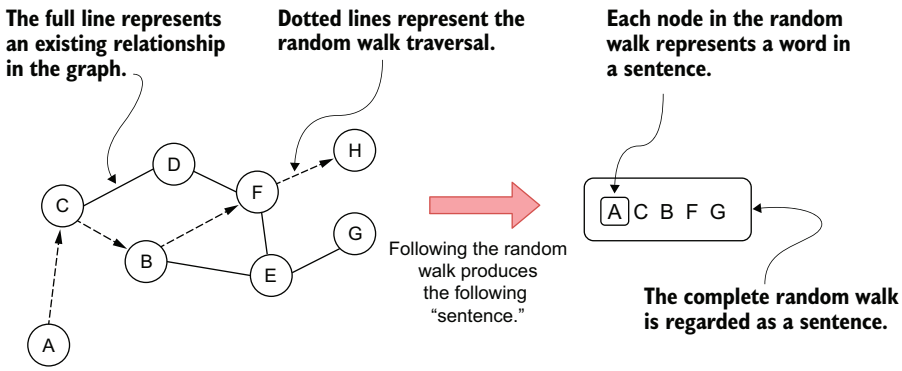

Figure 9.10 Using random walks to produce sentences

However, the node2vec algorithm implements second-order biased random walks. A step in the first-order random walk only depends on its current state.

Imagine you somehow wound up at node A in figure 9.11. Because the first-order random walk only looks at its current state, the algorithm doesn't know which node it was at in the previous step. Therefore, the probability of returning to a previous node or any other node is equal. There is no advanced math concept behind the calculation of probability. Node A has four neighbors, so the chance of traversing to any of them is 25% (1/4).

Suppose your graph is weighted, meaning each relationship has a property that stores its weight. In that case, those weights will be included in the calculation of the traversal probability.

In a weighted graph, the chance of traversing a particular connection is its weight divided by the sum of all neighboring weights. For example, the probability of traversing from node A to node E in figure 9.12 is 2 divided by 8 (25%) and the probability of traversing from node A to node D is 37.5%.

On the other hand, second-order walks consider both the current as well as the previous state. To put it simply, when the algorithm calculates the traversal probabilities, it also considers where it was at the previous step.

In figure 9.13, the walk just traversed from node D to node A in the previous step and is now evaluating its next move. The likelihood of backtracking the walk and immediately revisiting a node in the walk is controlled by the return parameter p. If the value of return parameter p is

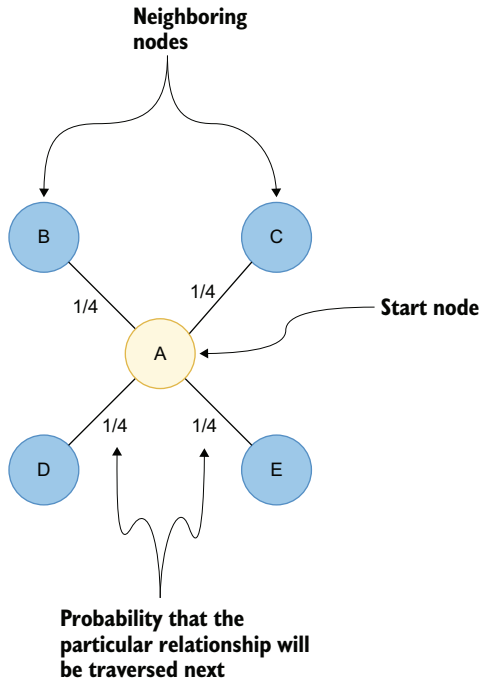

Figure 9.11 First-order random walks

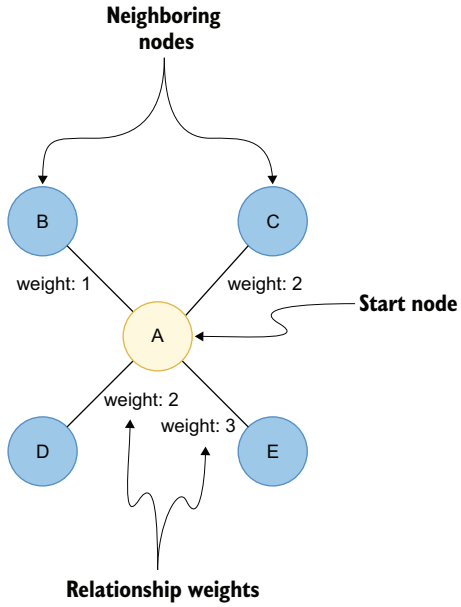

Figure 9.12 First-order weighted random walks

low, then the chance of revisiting node D is higher, keeping the random walk closer to the starting node of the walk. Conversely, setting a high value to parameter p ensures lower chances of revisiting node D and avoids two-hop redundancy in sampling. A higher value of parameter p also encourages moderate graph exploration.

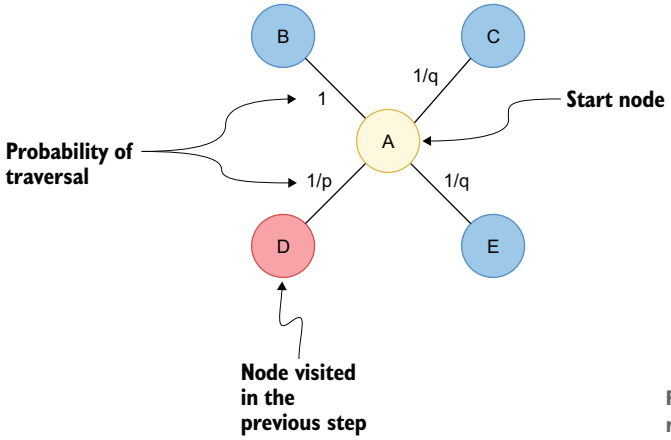

Figure 9.13 Second-order random walks

The `inOut` parameter q allows the traversal calculation to differentiate between inward and outward nodes. Setting a high value for parameter q ($q > 1$) biases the random walk to move toward nodes closer to the node in the previous step. Looking at figure 9.13, if you set a high value for parameter q, the random walk from node A is more biased toward node B. Such walks obtain a local view of the underlying graph with respect to the starting node in the walk and approximate breadth-first search. In contrast, if the value of q is low ($q < 1$), the walk is more inclined to visit nodes further away from node D. In figure 9.13, nodes C and E are further away, since they are not neighbors of the node in the previous step. This strategy encourages outward exploration and approximates depth-first search.

Authors of the node2vec algorithm claim approximating depth-first search will produce more community- or homophily-based node embeddings. On the other hand, the breadth-first search strategy for random walks encourages structural role embeddings.

9.3.3 Calculate node2vec embeddings

Now that you have a theoretical understanding of node embeddings and the node2vec algorithm, you will use it in a practical example. As mentioned, your task as a data scientist at Twitch is to predict the languages of new streamers based on shared audiences or chatters between different streams. The graph is already constructed, so you only need to execute the node2vec algorithm and train a classification model. As always, you first must project an in-memory graph. Relationships represent shared audiences between streams. When stream A shares an audience with stream B, that directly implies that stream B also shares an audience with stream A. Therefore, you