Ngrams are a specialist in finding what word comes next in a sequence, acting as a sliding window over a sequence of words. They take n words at a time within that sliding window. Given the nature of ngrams, they are said to be a probabilistic language model, meaning that we need a corpus to train a model on so that we can make predictions on a separate corpus we want to predict upcoming words on.

Ngrams have quite a few use cases. One such use case would be spell checking, as ngrams can be used to correct a misspelled word by finding the closest matching ngram within a certain corpus. Additionally, ngrams can prove to be useful in 'thing' recognition. (I like to think of it like 'did you mean x' section of a botched google search.) Where a similar entity like a famous person, event, etc. may be suggested based on frequent ngrams that correspond to those people, events and so on.

Ngram probability is calculated with a Markov assumption, due to how hairy the computation can get if we want to predict the next sequence of n words. (If n>3 things can get really messy.) This means that the probability only looks at the previous word, meaning we can generalize to:

$$\Pi_{k=1}^{n} P(w_k|w_{k-1})$$

The source text of any language model is vital, as with ngrams it can greatly skew what words we may predict to come later on in the text. Training on Shakespeare and then trying to predict the next few words in the python documentation will most likely end in less than desirable results.

With that said, even in a language source as closely related to the problem domain as possible, we can't contain every sequence of every word in that corpus. This is where smoothing comes in. So to prevent zero-ing out our probability in the language model, we opt to smooth the data by filling in zero values with a bit of probability of the overall mass, thus smoothing the distribution. Laplace smoothing is simple to implement at the cost of performance due to the heavy probability adjusting it does. In laplace smoothing we add a +1 to the count function on the numerator and add the total vocabulary count to the denominator, N, the total amount of unigrams.

Text generation from language models is typically done probabilistically where the probability of the next word in the sequence is calculated based on the word prior to the word being predicted. The limitations to this text generation technique is that our context is limited to the sliding window of the ngram model, and we can't expect superb pattern recognition based on the training data, thus hampering creativity.

To evaluate a language model, a common method is evaluating perplexity. Where human annotators evaluate results based on some previously determined metric. These methods are expensive and time consuming so they aren't used all too often. This is an extrinsic approach, whereas an intrinsic approach evaluates the model itself. Perplexity measures how well the language models predict text in the test data set.

Google ngram viewer is a cool tool to search and compare ngrams in the Google Books corpus. You can edit the time frame (spanning from 1800-Present-ish) and allow for smoothing and case-sensitivity toggling. Hitting search gives a frequency graph of the ngrams over the set time frame. So as an example I searched Oscar Wilde and The Importance of Being Earnest from 1800-1900, and saw a bump in both frequencies towards the end of the 1890s.