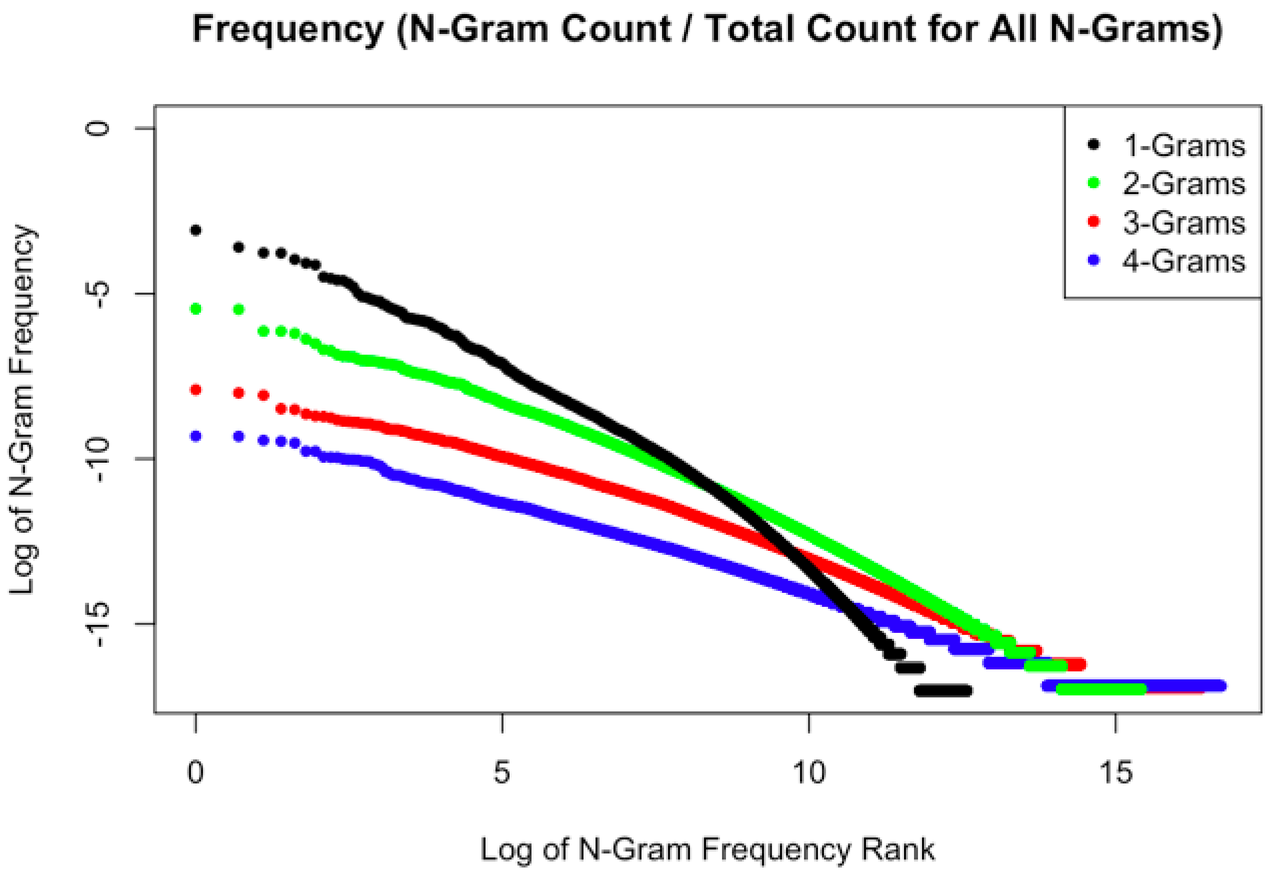
Predicting the Next Word: Back-Off Language Modeling

David Masse Follow
Aug 30, 2018 · 7 min read

Expertly tuned machine-learning algorithms are generally able to achieve fairly strong accuracy in a variety of settings. The winning entries in Kaggle contests, for example, often boast accuracy (measured in a variety of ways) of 80–99.99% on unseen test cases. So why is it so difficult to predict with confidence what the next word in passage of text will be? After all, natural-language data is abundant everywhere and easily stored and manipulated, and it is fairly easy to train a computer to recognize and classify writing styles accurately.



To answer this question, we have to ask another one: How are words statistically distributed in speech and writing? Remarkably, this question went largely unexplored until the early twentieth century. In 1932, American linguist George Kingsley Zipf formalized the empirical observation that the frequency of the most common word in a given natural language is roughly double that of the second-most-common word, triple that of the third-most-common word, and so on (now called Zipf's law). Since the frequency of a given word is thus inversely proportional to its frequency rank, the log of a word's frequency is negatively proportional to the log of its rank. A log-log plot of word frequency vs. rank generally adheres to to a line with slope negative one — even in the case of signals produced by whales and dolphins and other natural phenomena, such as sequences of amino acids. Theories about why this is the case and the implications of Zipf's law are fascinating and still under development.

This is great to know but actually makes word prediction really difficult. Zipf's law implies that most words are quite rare, and word combinations are rarer still. However, assigning no probability to phrases or sentences that have never been observed before, provided they are gramatically correct, will lead to greater error than necessary. A variety of approaches have been taken to address these issues, including neural networks. Most study sequences of words grouped as n-grams and assume that they follow a Markov process, i.e. that the next word only depends on the last few, with no relation to those that came hundreds of paragraphs before or after. (As an aside, one definition of literature or poetry could be language that violates this assumption.)

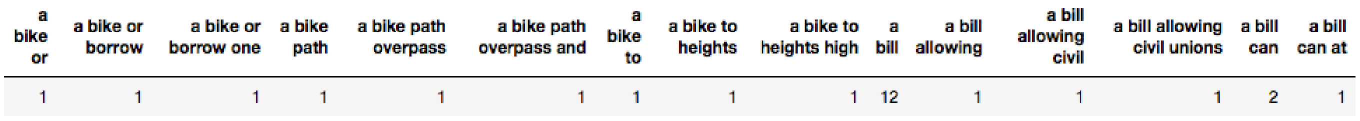
Some of the simplest training algorithms were developed quite recently, though relying on seminal work done by Alan Turing and I. J. Good during World War II. In 1987, Slava M. Katz introduced his back-off model for calculating the conditional probabilities for each word that might complete an n-gram. It takes the maximum-likelihood estimator of each potential completed n-gram as the ratio between the number of occurrences of that n-gram in the training set and the number occurrences of the (n-1)-word "prefix" for this n-gram. If this ratio is zero (i.e. the n-gram is not in the training set), the model "backs off" to look at the ratio between the n-gram and the (n-2)-word prefix obtained by lopping the first word from the original prefix. This process continues recursively, multiplying by a factor between zero and one (we'll call it k) each time. If the prefix is reduced to no words (i.e. the completing word is not in the training set at all (or is only found at the very beginning of the training set), a smoothing algorithm is needed, typically Kneser-Ney or the aforementioned Good-Turing.

Katz's back-off requires the computationally intensive estimation of two other parameters besides k to generate true conditional probabilities for each candidate for completion of an n-gram. In 2007, Google linguists proposed a simpler model that they called Stupid Back-off : "The name originated at a time when we thought that such a simple scheme cannot possibly be good. Our view of the scheme changed, but the name stuck." Stupid Back-off only keeps the k parameter (heuristically set at a constant 0.4 in practice) and was shown to achieve results prediction results comparable to Katz's back-off without ever generating true probabilities, just scores for identifying the most likely next word. In the Python code that follows, I simply renormalize these scores to add up to one for all potential words to complete an n-gram and then list them in descending order. In my next post, I will examine the accuracy and (pseudo-)perplexity of this algorithm in an attempt to explore information theory and the concept of entropy as inaugurated by Claude Shannon.

```
# Python library imports:
import re
import pandas as pd
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer,
TfidfVectorizer
from nltk.tokenize import word_tokenize, WhitespaceTokenizer,
TweetTokenizer
np.random.seed(seed=234)

# The below reads in N lines of text from the 40-million-word news
corpus I used (provided by SwiftKey for educational purposes).
N = 10000
with open("news.txt") as myfile:
    articles = [next(myfile) for x in range(N)]
joined_articles = [" ".join(articles)]

# The below takes out anything that's not a letter, replacing it
with a space, as well as any single letter that is not the pronoun
"I" or the article "a."
def clean_article(article):
    art1 = re.sub("[^A-Za-z]", ' ', article)
    art2 = re.sub("\s[B-HJ-Zb-hj-z]\s", ' ', art1)
    art3 = re.sub("^[B-HJ-Zb-hj-z]\s", ' ', art2)
    art4 = re.sub("\s[B-HJ-Zb-hj-z]$", ' ', art3)
    return art4.lower()
```

```
# The below breaks up the words into n-grams of length 1 to 5 and
puts their counts into a Pandas dataframe with the n-grams as column
names.  The maximum number of n-grams can be specified if a large
corpus is being used.
ngram_bow = CountVectorizer(stop_words = None, preprocessor =
clean_article, tokenizer = WhitespaceTokenizer().tokenize,
ngram_range=(1,5), max_features = None, max_df = 1.0, min_df = 1,
binary = False)
ngram_count_sparse = ngram_bow.fit_transform(joined_articles)
ngram_count = pd.DataFrame(ngram_count_sparse.toarray())
ngram_count.columns = ngram_bow.get_feature_names()
```

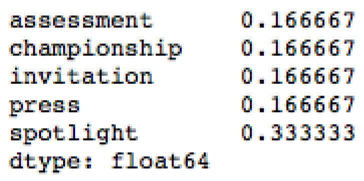
sample of n-gram counts:



a bike or	a bike or borrow	a bike or borrow one	a bike path	a bike path overpass	a bike path overpass and	a bike to	a bike to heights	a bike to heights high	a bill	a bill allowing	a bill allowing civil	a bill allowing civil unions	a bill can	a bill can at
1	1	1	1	1	1	1	1	1	12	1	1	1	2	1

```
# The below turns the n-gram-count dataframe into a Pandas series
with the n-grams as indices for ease of working with the counts.
The second line can be used to limit the n-grams used to those with
a count over a cutoff value.
sums = ngram_count.sum(axis = 0)
sums = sums[sums > 0]
ngrams = list(sums.index.values)

# The function below gives the total number of occurrences of 1-
grams in order to calculate 1-gram frequencies
def number_of_onegrams(sums):
    onegrams = 0
    for ng in ngrams:
        ng_split = ng.split(" ")
        if len(ng_split) == 1:
            onegrams += sums[ng]
    return onegrams

# The function below makes a series of 1-gram frequencies.  This is
the last resort of the back-off algorithm if the n-gram completion
does not occur in the corpus with any of the prefix words.
def base_freq(og):
    freqs = pd.Series()
    for ng in ngrams:
        ng_split = ng.split(" ")
        if len(ng_split) == 1:
            freqs[ng] = sums[ng] / og
    return freqs
```

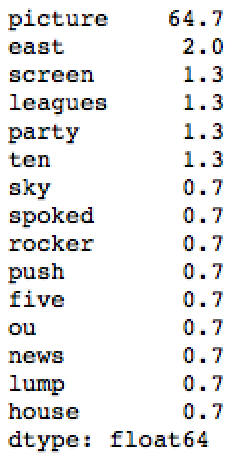
```
# For use in later functions so as not to re-calculate multiple
times:
bf = base_freq(number_of_onegrams(sums))

# The function below finds any n-grams that are completions of a
given prefix phrase with a specified number (could be zero) of words
'chopped' off the beginning.  For each, it calculates the count
ratio of the completion to the (chopped) prefix, tabulating them in
a series to be returned by the function.  If the number of chops
equals the number of words in the prefix (i.e. all prefix words are
chopped), the 1-gram base frequencies are returned.
def find_completion_scores(prefix, chops, factor = 0.4):
    cs = pd.Series()
    prefix_split = prefix.split(" ")
    l = len(prefix_split)
    prefix_split_chopped = prefix_split[chops:l]
    new_l = l - chops
    if new_l == 0:
        return factor**chops * bf
    prefix_chopped = ' '.join(prefix_split_chopped)
    for ng in ngrams:
        ng_split = ng.split(" ")
        if (len(ng_split) == new_l + 1) and (ng_split[0:new_l] ==
prefix_split_chopped):
            cs[ng_split[-1]] = factor**chops * sums[ng] /
sums[prefix_chopped]
    return cs

# Example of completion scores:
find_completion_scores('in the national', 0, 0.4)
```



```
assessment      0.166667
championship    0.166667
invitation      0.166667
press           0.166667
spotlight       0.333333
dtype: float64
```

We can chek the above by looking at the count for ‘in the national spotlight’ (`sums[‘in the national spotlight’]`), which is 2, and the count for ‘in the national’ (`sums[‘in the national’`], which is 6.

```
# The below tries different numbers of 'chops' up to the length of
the prefix to come up with a (still unordered) combined list of
scores for potential completions of the prefix.
def score_given(given, fact = 0.4):
    sg = pd.Series()
    given_split = given.split(" ")
    given_length = len(given_split)
    for i in range(given_length+1):
        fcs = find_completion_scores(given, i, fact)
        for i in fcs.index:
            if i not in sg.index:
                sg[i] = fcs[i]
    return sg

#The below takes the potential completion scores, puts them in
descending order and re-normalizes them as a percentage (pseudo-
probability).
def score_output(given, fact = 0.4):
    sg = score_given(given, fact)
    ss = sg.sum()
    sg = 100 * sg / ss
    sg.sort_values(axis=0, ascending=False, inplace=True)
    return round(sg,1)

# The below shows that even with a corpus that is way too small
(only 1.7 million words vs. billions or even trillions in current
research), results start to become intuitive.
score_output('on the big', fact = 0.4)[0:15]
```



```
picture    64.7
east        2.0
screen      1.3
leagues     1.3
party       1.3
ten         1.3
sky         0.7
spoked      0.7
rocker      0.7
push        0.7
five        0.7
ou          0.7
news        0.7
lump        0.7
house       0.7
dtype: float64
```

Machine Learning NLP Markov Chains Python