**Simple (Unsmoothed) N-gram in NLP**

**Overview**

N-grams are **continuous sequences** of words or symbols or tokens in a document and are defined as the **neighboring sequences of items** in a document. They are used most importantly in tasks dealing with text data in NLP (Natural Language Processing).

N-gram models are widely used in statistical natural language processing, speech recognition, phonemes and sequences of phonemes, machine translation and predictive text input, and many others for which the modeling inputs are n-gram distributions.

In Natural Language Processing (NLP), "unsmoothed n-grams" refers to a sequence of "n" words extracted from a text without applying any smoothing techniques, meaning that if a particular combination of words ("n-gram") isn't seen in the training data, its probability will be calculated as zero, potentially causing issues with unseen sequences during prediction; smoothing methods are typically applied to assign non-zero probabilities to these unseen n-grams, making the model more robust

**What are n-grams?**

N-grams are defined as the **contiguous sequence of n items** that can be extracted from a given sample of text or speech. The items can be letters, words, or base pairs, according to the application. The N-grams typically are collected from a **text or speech corpus** (Usually a corpus of long text dataset).

* N-grams can also be seen as a **set of co-occurring words** within a given window computed by basically moving the window some k words forward (k can be from 1 or more than 1).
* The co-occurring words are called "n-grams," and "n" is a number saying how long a string of words we have considered in the construction of n-grams.
* Unigrams are single words, bigrams are two words, trigrams are three words, 4-grams are four words, 5-grams are five words, etc.

**Applications of N-grams:** N-grams of texts are extensively used in the n-gram model in NLP, text mining, and natural language processing tasks.

* For example, when **developing language models** in natural language processing, n-grams are used to develop not just unigram models but also bigram and trigram models.
* Tech companies like Google and Microsoft have developed **web-scale n-gram models** that can be used in a variety of NLP-related tasks such as spelling correction, word breaking, and text summarization.
* One other major usage of n-grams is for developing features for supervised Machine Learning models such as SVMs, MaxEnt models, Naive Bayes, etc. The main idea is to use tokens such as bigrams (and trigrams and advanced n-grams) in the feature space instead of just unigrams.

**Importance of order of words in text & NLP**

* In general, the **order of the words** that are used in the natural language of the text is not random. If we consider the English language, we can use the words "the green apple" but not "apple green the" in a sentence.
* The **relationships** between words in the flow of the text are also very complex. N-grams are tooted as a relatively simple way of capturing some of these relationships between the words.
* We can capture quite a bit of information by just looking at which words tend to show up next to each other more often. By constructing the n-grams, we can construct the co-occurring words and use them in modeling relationships and other applications.
* **Example:** The general idea with n-grams is that we can look at each pair or triple or set of four or more words that occur next to each other in a sufficiently large corpus.
  + When we construct these pairs of words, we are more likely to see "the green" and "green apple" several times but less likely to see "apple green" and "green the".
  + This kind of context is useful to know and utilize in many use cases of NLP. We can figure out what someone is more likely to say to help decide between the possible outputs for an automatic speech recognition system.

**How are n-grams Classified?**

N-grams are classified into different types depending on the value that n takes. When n=1*n*=1, it is said to be a unigram. When n=2*n*=2, it is said to be a **bigram**. When n=3*n*=3, it is said to be a **trigram**. When n=4*n*=4, it is said to be a 4-gram, and so on.

* Different types of n-grams are suitable for different types of applications in the n-gram model in nlp, and we need to try different n-grams on the dataset to confidently conclude which one works the best among all for the text corpus analysis.
* It is also established with research and substantiated that trigrams and 4 grams work the best in the case of **spam filtering**.

**An Example of n-grams**

Let us look at the example sentence **Cowards die many times before their deaths; the valiant never taste of death but once** and generate the associated n-grams related to the sentence.

* **Unigrams**: These are simply the **unique words in the sentence**.
  + Cowards, die, many, times, before, their, deaths, the valiant, the valiant, never, taste, of, death, but, once.
* **Bigrams**: These are simply the **pairs of co-occurring words** in the sentence formed by sliding one word at a time in the forward direction to generate the next bigram.
  + cowards die, die many, many times, times before, before their, their deaths, deaths the, the valiant, valiant never, never taste, the taste of, of death, death but, but once
* **Trigrams**: These are the **3 pairs of co-occurring words** in the sentence formed by sliding two words at a time in the forward direction to generate the next trigram.
  + cowards die many, die many times, many times before, times before their, before their deaths, their deaths the, deaths the valiant, the valiant never, valiant never taste, never taste of, taste of death, of death but, death but once
* **4-grams**: Here we have the window such that we have **combinations of 4 words together**
  + cowards die many times, die many times before, many, times before their, times before their deaths, before their deaths the, their deaths the valiant, deaths the valiant never, the valiant taste, valiant never taste of, never taste of death, taste of death but, of death but once
* Simialary we can pick n>4*n*>4 and generate 5-grams etc.

**Step-by-Step Implementation of n-grams in Python**

**Load the Dataset and Libraries**

**Import the libraries for processing data and other utilities**

* import os, sys, gc, warnings
* import logging, math, re
* import pandas as pd
* import numpy as np
* import seaborn as sns
* import matplotlib.pyplot as plt
* from matplotlib.pyplot import figure
* from IPython.display import display, HTML
* from IPython.core.interactiveshell import InteractiveShell

**The following settings help in proper formatting and display of the output of code we run.**

* warnings.filterwarnings("ignore")
* pd.set\_option('max\_rows', None)
* pd.set\_option('max\_columns', None)
* InteractiveShell.ast\_node\_interactivity = "all"
* display(HTML(data=""""""))

If a particular library is not found, it can simply be downloaded from within the notebook by running a system command pip install sample\_library command. For example to install seaborn, run the command from a cell within jupyter notebook like:

! pip install seaborn

**Getting and Loading the Dataset**

* For this particular text classification problem, we are getting data from Kaggle datasets related to Stock market sentiment classification which can be found here [stock market sentiment analysis dataset](https://www.kaggle.com/datasets/yash612/stockmarket-sentiment-dataset).
* The dataset is a corpus of tweets that are pulled from Twitter and tagged manually for training models on this tagged variable that we can use for our n-gram model in NLP.
* There is a single dataset CSV file that is to be downloaded and kept in the same location in the place where the jupyter notebook with the code which we run here is there.

**Sample data looks like below:**

| **Text** | **Sentiment** |
| --- | --- |
| OI Over 21.37 | 1 |
| PGNX Over 3.04 | 1 |
| AAP - user if so then the current downtrend will break. Otherwise just a short-term correction in a med-term downtrend. | -1 |
| Monday's relative weakness. NYX WIN TIE TAP ICE INT BMC AON C CHK BIIB | -1 |
| GOOG - ower trend line channel test & volume support. | 1 |
| AAP will watch tomorrow for ONG entry. | 1 |

**Load the dataset here**

df = pd.read\_csv('stock\_data.csv',header=0,encoding='utf8')

**Summary of the data set**

df.shape

df.head()

**Summary of target variable**, We also plot the value counts of the target variable

plt.figure(figsize=(10,4))

sns.countplot(x='Sentiment',data=df)

plt.show()

print('% Counts by target variable:')

df['Sentiment'].value\_counts()/df.shape[0]\*100

**Text Pre-processing**

* **Preprocess the text in the corpus:** We will clean the text by stripping punctuation and whitespace, converting to lowercase, and removing stopwords, these steps can be generally followed for the n-gram model in nlp.
* Let us also see how to use the spacy NLP library in python to get stop words and them with stopwords from nltk and remove the combined set of stop words on the pandas raw dataset directly

**This is to download required libraries from nltk**

* import nltk
* nltk.download('punkt')
* nltk.download("stopwords")
* nltk.download('wordnet')

**Importy the necessary utilities**

* from nltk.corpus import stopwords
* from nltk import word\_tokenize, sent\_tokenize

**Download spacy if it doesn't exist and import the library**

try:

import spacy

except:

! pip install -U pip setuptools wheel

! pip install -U spacy

! python -m spacy download en

import spacy

sp = spacy.load('en\_core\_web\_sm')

**Extract list of stop words from spacy**

cachedStopWords = sp.Defaults.stop\_words

cachedStopWords = [x.lower() for x in cachedStopWords]

**Combine the stop words with the ones from nltk**

cachedStopWords.extend(list(stopwords.words('english')))

cachedStopWords = list(set(cachedStopWords))

We will write a function for utilizing the stop words and process them on the pandas data frame column directly for our n gram model in nlp.

def remove\_stopwords(x):

*# Initiate the lists*

word\_list = []

my\_list = x

*# Tokenise the sentence*

tokenized\_my\_list = word\_tokenize(my\_list)

word\_list = [w for w in tokenized\_my\_list if not w in cachedStopWords]

return " ".join(word\_list)

**Function for cleaning the text**

def cleantext(x):

s = re.sub(r'[^a-zA-Z0-9\s]', '', x)

s = re.sub('\s+',' ', s)

s = str(s).lower()

tokens = [token for token in s.split(" ") if token != ""]

sent = ' '.join(tokens)

return sent

**Let us apply the processing functions here**

df['text\_clean'] = df['Text'].apply(cleantext)

df['text\_clean'] = df['text\_clean'].apply(remove\_stopwords)

**Dataset after pre-processing the text looks like this:**

| **Text** | **Sentiment** | **text\_clean** |
| --- | --- | --- |
| OI Over 21.37 | 1 | oi 2137 |
| PGNX Over 3.04 | 1 | pgnx 304 |
| AAP - user if so then the current downtrend will break. Otherwise just a short-term correction in a med-term downtrend. | -1 | aap user current downtrend break short-term correction medterm downtrend |
| Monday's relative weakness. NYX WIN TIE TAP ICE INT BMC AON C CHK BIIB | -1 | mondays relative weakness nyx win tie tap ice int bmc aon c chk biib |
| GOOG - ower trend line channel test & volume support. | 1 | goog ower trend line channel test volume support |

**Explore the Dataset**

We will also do the EDA on the dataset related to the text in the corpus. For understanding the text in the dataset we will look at the distribution of words across different analyses.

**Plot the length of the text appearing in the dataset**

plt.figure(figsize=(10,4))

length = df['Text'].str.len()

sns.kdeplot(length)

plt.title('Distribution of length of text appearing in the dataset')

plt.show()

**Output:**

Let us now look at the distribution of words by different sentences in the corpus. Compute the number of words in each line

pattern = re.compile(r'\w+')

df['Number of words'] = df['Text'].apply(lambda x: max(1, len(pattern.findall(x))))

x = df['Number of words']

y = df["Sentiment"]

**Plot the distribution of the number of words**

plt.figure(figsize=(20, 4))

plt.hist(x, bins=25)

plt.title('Distribution of number of words per sentence')

plt.ylabel("counts in bin")

plt.xlabel("Number of words")

print();

**Output:**

**Distribution of context vs. number of words**

sorted\_list = [[j, i] for j, i in sorted(zip(x, y))]

x = [i[0] for i in sorted\_list]

y = [i[1] for i in sorted\_list]

plt.figure(figsize=(20, 4))

plt.hist2d(x, y, bins=[40, 2], range=[[0, 41], [0,1]], cmap='viridis')

plt.ylabel("Negative | Positive")

plt.ylim(0, 1)

plt.yticks([])

plt.xticks(np.arange(1,41))

plt.xlabel("Number of words in a sentence")

cb = plt.colorbar()

cb.set\_label('Counts in bin')

print();

**Output:**

* One main finding from the eda: We can see that positive sentences tend to have more words than negative ones

**n-gram Analysis and Visualization**

* For understanding the most frequent words we can look at the wordclouds of unigrams and bigrams in our further analysis.
* We will write the Code to generate N-grams: Here let us write a function to extract n-grams and use it to plot top bigrams and trigrams in our raw text corpus.
* nltk also provides the function ngrams. It can be used to determine a set of all possible n consecutive words appearing in a text.

**Load the function from the nltk library**

* from nltk.util import ngrams

def get\_ngrams(text, n=2):

text = str(text)

n\_grams = ngrams(text.split(), n)

returnVal = []

try:

for grams in n\_grams:

returnVal.append('\_'.join(grams))

except(RuntimeError):

pass

return ' '.join(returnVal).strip()

**Code for n-grams without using nltk:** If we do not want to use the nltk package for generating n-grams, we can use the following function directly on the corpus to generate n-grams of different types for the n-gram model in nlp.

**Load the text utilities library**

import re

**We will Write the function here**

def generate\_ngrams(text,n):

*# Split the sentences into tokens*

*# We are using the identifier "\\s+" for splitting*

tokens=re.split("\\s+",text)

ngrams=[]

*# Collect the n-grams*

for i in range(len(tokens)-n+1):

temp=[tokens[j] for j in range(i,i+n)]

ngrams.append(" ".join(temp))

return ngrams

**Creating Unigrams**

* We will generate unigrams and draw the wordclouds to look at the most frequent words. We will then plot for unigrams, bigrams, and trigrams
* We will import the wordcloud library for plotting the word counts. Then We shall check if the library exists, if not we install and import it

try:

import wordcloud

except:

! pip install wordcloud

import wordcloud

from wordcloud import WordCloud

from sklearn.feature\_extraction.text import CountVectorizer

**Generate the Unigrams**

* Here we will tokenize all the words in the corpus and lemmatize them and add to the overall corpus as we iterate over the entire dataframe of raw data.
* We will extract and plot unigrams (or distinct words in the dataset) using wordcloud which is a good visualization for the n gram model in nlp.

word\_list=[]

for a word in df.Text:

word = re.sub("[^a-zA-z]"," ",word)

word = word.lower()

word = nltk.word\_tokenize(word)

lemma = nltk.WordNetLemmatizer()

word = [lemma.lemmatize(w) for w in word]

word=" ".join(word)

word\_list.append(word)

print(len(word\_list))

print('Sample processed sentences')

print(word\_list[:2])

max\_features = 500

count\_vectorizer = CountVectorizer(

max\_features=max\_features,

stop\_words="english")

sparce\_matrix = count\_vectorizer.fit\_transform(

word\_list).toarray()

print("Top {} the most used words: {}".format(

10,

count\_vectorizer.get\_feature\_names()[:10]))

print(sparce\_matrix.shape)

data = pd.DataFrame(

count\_vectorizer.get\_feature\_names(),

columns=["Words"])

data.head(4)

plt.subplots(figsize=(15,10))

wordcloud=WordCloud(background\_color="black",width=1024,height=768).generate(" ".join(data.Words[0:400]))

plt.imshow(wordcloud)

plt.axis("off")

plt.show()

**Creating Bigrams**

**Extract and plot bigrams**

df["bigram\_text"] = df["text\_clean"].apply(get\_ngrams, n=2)

df.head()

bigram\_string\_list = df['bigram\_text'].tolist()

bigram\_string = ' '.join(bigram\_string\_list)

wordcloud = WordCloud(

width = 2000, height = 1334, random\_state=1,

background\_color='black', colormap='Pastel1',

max\_words = 75,

collocations=False,

normalize\_plurals=False).generate(bigram\_string)

fig = plt.figure(figsize=(15, 10),dpi=80)

plt.tight\_layout(pad=0)

plt.imshow(wordcloud)

plt.axis("off")

plt.box(False)

plt.show()

plt.close()

**Creating Trigrams**

We can **create trigrams** simplifying using the function created for trigrams with n=3*n*=3

df["trigram\_text"] = df["text\_clean"].apply(get\_ngrams, n=3)

Let us now plot the moccurringring trigrams instead of wordclouds for alternative representation.

**Load the required library** for counting the n-grams (trigramheree).

from collections import Counter

trigram\_string\_list = df['trigram\_text'].str.split(' ').values

trigram\_string\_list= list(trigram\_string\_list)

trigram\_string\_list = sum(trigram\_string\_list,[])

trigram\_counts = Counter(trigram\_string\_list)

df\_trigram\_counts = pd.DataFrame.from\_dict(

trigram\_counts, orient='index')

df\_trigram\_counts = df\_trigram\_counts.reset\_index()

df\_trigram\_counts.columns = ['trigram\_word','n']

df\_trigram\_counts = df\_trigram\_counts.sort\_values('n',ascending=False)

df\_trigram\_counts = df\_trigram\_counts.reset\_index(drop=True)

fig = plt.figure(figsize = (20, 7))

plt.barh(df\_trigram\_counts[1:15]['trigram\_word'], df\_trigram\_counts[1:15]['n'],align='center', alpha=.5)

plt.title('Top 15 Trigrams in the dataset')

plt.xlabel('Count')

| **bigram\_text** | **trigram\_text** |
| --- | --- |
| kickers\_watchlist watchlist\_xide xide\_tit tit\_... | kickers\_watchlist\_xide watchlist\_xide\_tit xide... |
| user\_aap aap\_movie movie\_55 55\_return return\_f... | user\_aap\_movie aap\_movie\_55 movie\_55\_return 55... |
| user\_id id\_afraid afraid\_short short\_amzn amzn... | user\_id\_afraid id\_afraid\_short afraid\_short\_am... |

**Train-test Split and Results of the Model**

* We will write a function to utilize the vectorizer in different cases and train aasgdclassifier which is a better model for handling high dimensional data like n-gram features and output the performance of the model for comparison.

**Load the required libraries and functions**

* from sklearn.linear\_model import SGDClassifier
* from sklearn.model\_selection import train\_test\_split
* from scipy.sparse import csr\_matrix

def get\_model\_performance(X: csr\_matrix, y: np.array, title: str) -> None:

*# We shall split the data into 75, 25 ratios*

X\_train, X\_valid, y\_train, y\_valid = train\_test\_split(

X, y, train\_size=0.75, stratify=y)

*# Fit an sgd classifier*

clf = SGDClassifier()

clf.fit(X\_train, y\_train)

*# Get the train and test scores*

train\_score = clf.score(X\_train, y\_train)

valid\_score = clf.score(X\_valid, y\_valid)

return [train\_score,valid\_score]

Let us **compare the model with input features** for the following different combinations:

* Unigrams, Bigrams, Trigrams individually
* Unigrams & Bigrams, Bigrams & Trigrams , Unigrams & Bigrams & Trigrams as combinations

**Initiate the scores list**

Overall\_scores = []

y\_train = df['Sentiment'].values

**Unigram model**

vectorizer\_c = CountVectorizer(analyzer='word',stop\_words='english', ngram\_range=(1, 1))

vectorizer\_c.fit(df['text\_clean'].values)

X\_train\_c = vectorizer\_c.transform(df['text\_clean'].values)

score\_c = get\_model\_performance(X\_train\_unigram, y\_train, 'ngram Counts')

Overall\_scores.append(score\_c)

**Bigram model**

vectorizer\_c = CountVectorizer(analyzer='word',stop\_words='english', ngram\_range=(2, 2))

vectorizer\_c.fit(df['text\_clean'].values)

X\_train\_c = vectorizer\_c.transform(df['text\_clean'].values)

score\_c = get\_model\_performance(X\_train\_c, y\_train, 'ngram Counts')

Overall\_scores.append(score\_c)

**Trigram model**

vectorizer\_c = CountVectorizer(analyzer='word',stop\_words='english', ngram\_range=(3, 3))

vectorizer\_c.fit(df['text\_clean'].values)

X\_train\_c = vectorizer\_c.transform(df['text\_clean'].values)

score\_c = get\_model\_performance(X\_train\_c, y\_train, 'ngram Counts')

Overall\_scores.append(score\_c)

**Unigrams & Bigrams model**

vectorizer\_c = CountVectorizer(analyzer='word',stop\_words='english', ngram\_range=(1, 2))

vectorizer\_c.fit(df['text\_clean'].values)

X\_train\_c = vectorizer\_c.transform(df['text\_clean'].values)

score\_c = get\_model\_performance(X\_train\_unigram, y\_train, 'ngram Counts')

Overall\_scores.append(score\_c)

**Bigrams & Trigrams model**

vectorizer\_c = CountVectorizer(analyzer='word',stop\_words='english', ngram\_range=(2, 3))

vectorizer\_c.fit(df['text\_clean'].values)

X\_train\_c = vectorizer\_c.transform(df['text\_clean'].values)

score\_c = get\_model\_performance(X\_train\_c, y\_train, 'ngram Counts')

Overall\_scores.append(score\_c)

**Unigrams & Bigrams & Trigrams combined model**

vectorizer\_c = CountVectorizer(analyzer='word',stop\_words='english', ngram\_range=(1, 3))

vectorizer\_c.fit(df['text\_clean'].values)

X\_train\_c = vectorizer\_c.transform(df['text\_clean'].values)

score\_c = get\_model\_performance(X\_train\_c, y\_train, 'ngram Counts')

Overall\_scores.append(score\_c)

df\_scores = pd.DataFrame(Overall\_scores)

df\_scores.columns = ['Train','Validation']

df\_scores['Model'] = ['Uni','Bi','Tri','Uni-Bi','Tri-Bi','Uni-Bi-Tri']

**Let us look at sample model performance from the above models:**

| **Model** | **Validation** |
| --- | --- |
| Uni | 0.772790 |
| Bi | 0.691298 |
| Tri | 0.662983 |
| Uni-Bi | 0.761050 |
| Tri-Bi | 0.691989 |
| Uni-Bi-Tri | 0.769337 |
|  |  |

From the preliminary models, it can be seen that the unigram models give the best performance along with multi-gram models of 3 grams (unigram plus bigram plus trigram grams models). The model can be further tuned for finding the best hyperparameters of the model and experimenting with other sets of models.

**Smoothing**

**Need for Smoothing:** The main purpose of smoothing is to prevent a language model (in particular n-gram models) from assigning zero probability to unseen events and handling out-of-vocabulary words.

* **Handling Out-of-vocabulary Words**: One of the issues with n-gram language models is **out-of-vocabulary (OOV)** words, encountered in computational linguistics and natural language processing when the input includes words not present in a system's dictionary or database during its preparation.
  + By default, when a language model is estimated, the entire observed vocabulary is used but in some cases, some words can appear in the same context but they may not be in the train set. Smoothing can help assign non-zero probabilities in such cases.
* In some specific cases it may also be necessary to estimate the language models with a specific fixed vocabulary. In such a scenario, the n-grams in the corpus that contain an out-of-vocabulary word are ignored. Still, the n-gram probabilities are smoothed over all the words in the vocabulary even if they were not observed.
* Smoothing is a tool to make the models more generalizable and realistic and prevent overfitting.

**Kneser-Ney Smoothing** (also known as Kneser-Essen-Ney smoothing): This is a method primarily used to calculate the **probability distribution** of n-grams in a document based on their histories. The smoothing is done in Kneser-Ney Smoothing by moving some probability towards unknown n-grams from known ones for the n-gram model in nlp.

* It is widely considered the most effective method of smoothing due to its use of absolute discounting by subtracting a fixed value from the probability's lower order terms to omit n-grams with lower frequencies.
* Kneser-Ney Smoothing approach has been considered equally effective for both higher and lower-order n-grams.

**Example for Kneser-Ney Smoothing:** One common example used to demonstrate the efficacy of Kneser-Ney smoothing generally is the phrase San Francisco. Let us say that this particular phrase is abundant in a given training corpus we used.

* Let us also consider an example context: I can’t see without my reading \_\_\_\_. By common sense, it is known that the word glasses should fill in the blank.
* But since San Francisco is a common term and the **unigram probability** of Francisco will also be high. If we simply use the absolute frequency of unigram models, then the term Francisco will take over and lead to some **strange results** in such scenarios.
* Kneser-Ney fixes such oddities by looking at the question of how likely a word is to appear in an unfamiliar bigram context so eventually it outputs such that P(glasses)>P(Francisco)*P*(*glasses*)>*P*(*Francisco*) in the end.

**Conclusion**

* N-grams are contiguous sequences of n items extracted from a given sample of text or speech where the items can be letters, words, or base pairs as per the application.
* N-grams are combinations of words that come together to express one better meaning from the context of a sentence or snippet of data.
* N-Grams are classified differently according to the number of combinations of words, where they are unigrams when n=1*n*=1, they are called bigrams when n=2*n*=2, and are called trigrams when n=3*n*=3 for the n-gram model in NLP.
* N-gram models are widely used in many applications like probability, communication theory, computational linguistics, etc.
* N-grams can also be used as input features for machine learning models, and by experimenting, we can find combinations of n-grams providing the desired accuracy depending on the application domain.
* The goal of smoothing is to prevent overfitting and handling out of vocabulary words in language models.