Datasets:

https://www.kaggle.com/bittlingmayer/amazonreviews
http://deepyeti.ucsd.edu/jianmo/amazon/index.html

In [ ]:

```
import pandas as pd
# Reading the json dataset into a pandas dataframe
df = pd.read_json('E:/Internships/TCS-iON/Code/MyCode/Amazon/Prime_Pantry.json', orient = '
# Adding a column representing 1 for 'pos' and 0 for 'neg' sentiments
df['senti'] = df.apply(lambda x: 1 if x['overall'] >= 4 else 0, axis = 1)
# Deleting unnecessary columns
df = df.drop(['verified', 'reviewTime', 'asin', 'reviewerName', 'summary', 'unixReviewTime'
# Converting the data type to string
df['reviewText'] = df["reviewText"].astype("str")
# Converting all text to lowercase for use
df['reviewText'] = df['reviewText'].str.lower()
df.head()
```

| overall | review | senti |
|---|---|---|
| 5 | good clinging | 1 |
| 4 | fantastic buy and a good plastic wrap. even t... | 1 |
| 4 | ok | 1 |
| 3 | saran cling plus is kind of like most of the c... | 0 |
| 4 | this is my go to plastic wrap so there isn't m... | 1 |

saran cling plus is kind of like most of the cling wrap from glad. it is a very good quality plastic wrap, but the delivery system is poorly executed, and this makes usage more frustrating and less time-efficient. as convenience is one of the core selling points of this sort of product, how easy it is to use is just as important as how good the wrap itself is.

as another user here noted, getting this stuff out of the box and tearing off the portion you need to use is very difficult. substantial force is required to do this due to a cutting system that is lacking, and this can result in the box being damaged as it is not a very robust box. it can also cause the piece you are attempting to tear off fold up from the pressure upon being torn, leaving you with a tangled mess to untangle. as the blade itself does not do a great job cutting, sometimes the wrap will tear, leaving you with a piece that is a different size than you wanted. the physical location of where the wrap comes out relative to where it is cut at also results in one having to hold at an awkward angle to see what they are doing, making all of this more difficult.

i use clear wrap multiple times a day...the difference between a delivery system that is easy versus difficult means a lot of time savings, and a lot of frustration that does not have to occur. this is a good quality wrap, but like glad's wrap, it's delivery system is lacking compared to other products (such as the kirkland clear wraps) and so the convenience factor is reduced. so i think that you will find that an alternative clear wrap product with a better system for dispensing & cutting is more enjoyable to use, and much more convenient.

In [ ]:

```
import re
import string
from nltk import WordNetLemmatizer
from nltk.stem.snowball import SnowballStemmer
from nltk.corpus import stopwords
# Initialising the nltk stop_words, stemmer and lemmatizer functions
stop_words = set(stopwords.words("english"))
lemmatizer = WordNetLemmatizer()
stemmer = SnowballStemmer("english")
# Creating a function for text cleaning
def textCleanser(myText):
    # Removing the name titles and the period symbols after it
    myText = re.sub(r'[mdsr]r(s)?\.', '', myText)
    # Removing punctuation
    myPunct = string.punctuation
    punctToSpace = str.maketrans(myPunct, len(myPunct)*' ')
    myText = myText.translate(punctToSpace)
    # Removing the '@username' mentions
    myText = re.sub(r'@\w+', '', myText)
    # Removing urls
    myText = re.sub(r'(http(s)?://)?(www\.)?.+\.com', '', myText)
    # Removing numbers
    myText = re.sub(r'\d+', '', myText)
    # Removing stopwords
    myText = [word for word in myText.split(' ') if not word in stop_words]
    myText = [word for word in myText if word != '']
    # Lemmatizing the text
    myText = [lemmatizer.lemmatize(token) for token in myText]
    # Stemming the text
    # myText = [stemmer.stem(token) for token in myText]
    return myText
for i in range(len(df['reviewText'])):
    df['reviewText'][i] = textCleanser(df['reviewText'][i])
df.head()
```

| | overall | review | senti |
|---|---|---|---|
| | 5 | [good, clinging] | 1 |
| | 4 | [fantastic, buy, good, plastic, wrap, even, th... | 1 |
| | 4 | [ok] | 1 |
| | 3 | [saran, cling, plus, kind, like, cling, wrap, ... | 0 |
| | 4 | [go, plastic, wrap, much, bad, say, plastic, w... | 1 |

'[saran', 'cling', 'plus', 'kind', 'like', 'cling', 'wrap', 'glad', 'good', 'quality', 'plastic', 'wrap', 'delivery', 'system', 'poorly', 'executed', 'make', 'usage', 'frustrating', 'le', 'time', 'efficient', 'convenience', 'one', 'core', 'selling', 'point', 'sort', 'product', 'easy', 'use', 'important', 'good', 'wrap', '\n\nas', 'another', 'user', 'noted', 'getting', 'stuff', 'box', 'tearing', 'portion', 'need', 'use', 'difficult', 'substantial', 'force', 'required', 'due', 'cutting', 'system', 'lacking', 'result', 'box', 'damaged', 'robust', 'box', 'also', 'cause', 'piece', 'attempting', 'tear', 'fold', 'pressure', 'upon', 'torn', 'leaving', 'tangled', 'mess', 'untangle', 'blade', 'great', 'job', 'cutting', 'sometimes', 'wrap', 'tear', 'leaving', 'piece', 'different', 'size', 'wanted', 'physical', 'location', 'wrap', 'come', 'relative', 'cut', 'also', 'result', 'one', 'hold', 'awkward', 'angle', 'see', 'making', 'difficult', '\n\ni', 'use', 'clear', 'wrap', 'multiple', 'time', 'day', 'difference', 'delivery', 'system', 'easy',

'versus', 'difficult', 'mean', 'lot', 'time', 'saving', 'lot', 'frustration', 'occur', 'good', 'quality', 'wrap', 'like', 'glad', 'wrap', 'delivery', 'system', 'lacking', 'compared', 'product', 'kirkland', 'clear', 'wrap', 'convenience', 'factor', 'reduced', 'think', 'find', 'alternative', 'clear', 'wrap', 'product', 'better', 'system', 'dispensing', 'cutting', 'enjoyable', 'use', 'much', 'convenient']"

In [ ]:

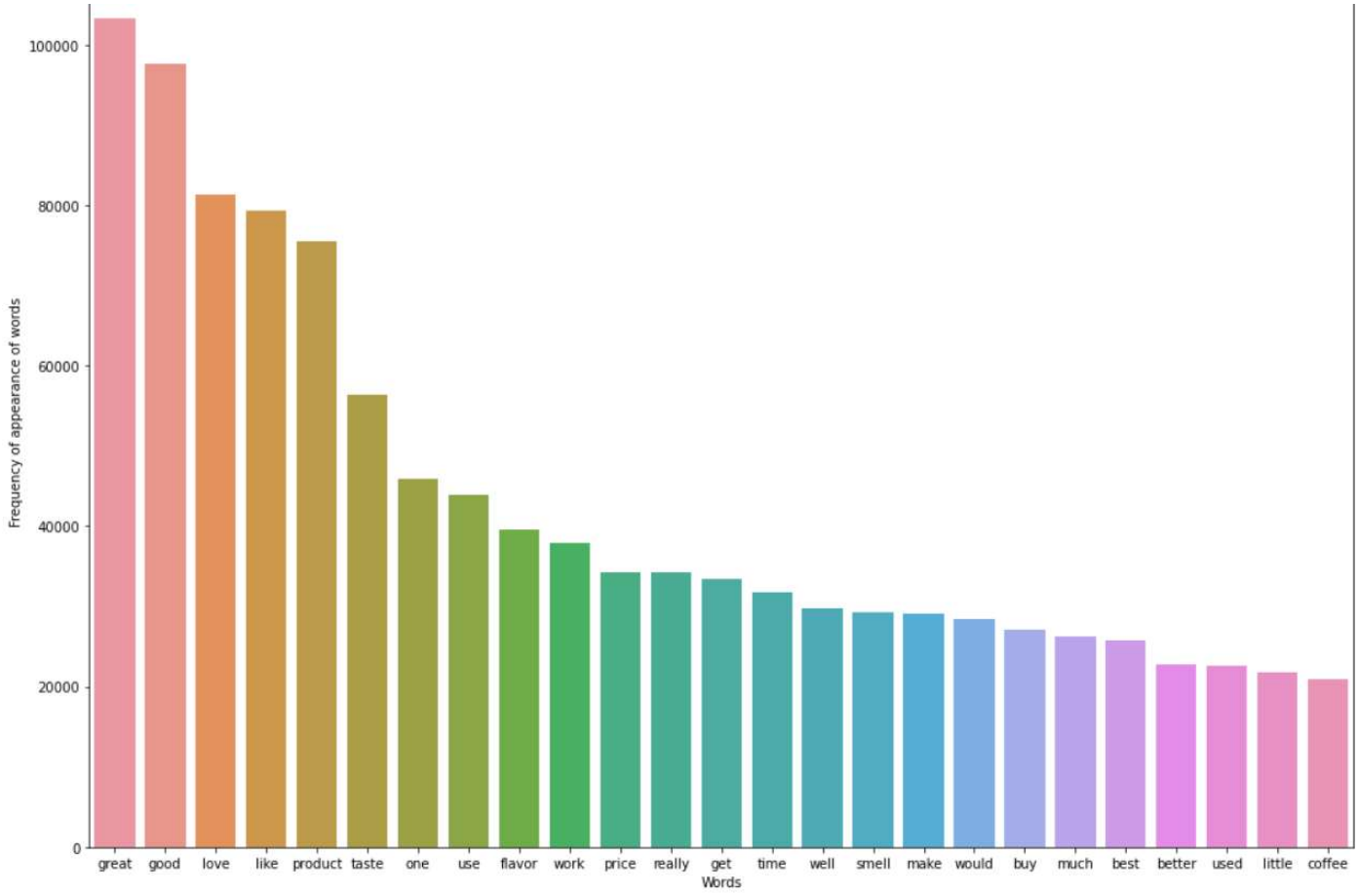
```
myReviews = []
for i in range(len(df['reviewText'])):
    for j in df['reviewText'][i]:
        if j != 'br' and j != 'http':
            myReviews.append(j)
```

In [ ]:

```
from collections import Counter
import collections
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.pipeline import Pipeline, FeatureUnion
from sklearn.metrics import classification_report
from sklearn.naive_bayes import MultinomialNB
from sklearn.linear_model import LogisticRegression
import numpy as np
np.random.seed(1234)
# Initialising the Count Vectorizer
cv = CountVectorizer()
myBow = cv.fit_transform(myReviews)
wordFrequency = dict(zip(cv.get_feature_names(), np.asarray(myBow.sum(axis = 0)).ravel()))
wordCounter = collections.Counter(wordFrequency)
# Storing the frequency of appearance of words
dfWordCounter = pd.DataFrame(wordCounter.most_common(25), columns = ['word', 'frequency'])
```

In [ ]:

```
# Plotting the top 25 most frequently occuring words
plt.close('all')
fig, ax = plt.subplots(figsize = (17, 12))
sns.barplot(x = 'word', y = 'frequency', data = dfWordCounter, ax = ax)
sns.set_palette('pastel')
plt.xlabel('Words')
plt.ylabel('Frequency of appearance of words')
plt.show()
```


Frequency of appearance of words
100000
80000
60000
40000
20000
0
great
good
love
like
product
taste
one
use
flavor
work
price
really
get
time
well
smell
make
would
buy
much
best
better
used
little
coffee
Words

In [ ]:

```
from sklearn.model_selection import train_test_split
import random
import torch
import torchtext
from torchtext import data
import spacy
import en_core_web_sm
import torch.nn as nn
import torch.nn.functional as fn
SEED = 1234
torch.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
# Creating functions to generate bigrams
def myBigrams(x):
    n_grams = set(zip(*[x[i:] for i in range(2)]))
    for n_gram in n_grams:
        x.append(' '.join(n_gram))
    return x
def dummyfn(doc):
    return doc
# Generating bigrams
TEXT = data.Field(tokenize = 'spacy', preprocessing = myBigrams)
LABEL = data.LabelField(dtype = torch.float)
# Creating training and testing datasets
training, testing = train_test_split(df, test_size=0.2, random_state=42)
training, validating = train_test_split(training, test_size=0.2, random_state=42)
training.to_csv('training.csv')
validating.to_csv('validating.csv')
testing.to_csv('testing.csv')
fields = [(None, None), (None, None), ('c', TEXT), ('s', LABEL)]
train_data, valid_data, test_data = data.TabularDataset.splits(
    path = 'E:/Internships/TCS-iON/Code/MyCode/Amazon/',
    train = 'training.csv',
    validation = 'validating.csv',
    test = 'testing.csv',
    format = 'csv',
    fields = fields,
    skip_header = True
)
MAX_VOCAB_SIZE = 50000
# Building the vocabularies
TEXT.build_vocab(
    train_data,
    max_size = MAX_VOCAB_SIZE,
    # Using the glove.6B.100d vector
    vectors = "glove.6B.100d",
    unk_init = torch.Tensor.normal_)
LABEL.build_vocab(train_data)
BATCH_SIZE = 64
device = torch.device('cpu')
# Creating vocabulary iterators
train_iterator, valid_iterator, test_iterator = data.BucketIterator.splits(
    (train_data, valid_data, test_data),
    sort = False,
    batch_size = BATCH_SIZE,
    device = device)
# Building the training model
class myModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, output_dim, pad_idx):
```

```
        super().__init__()
        # Initialising the embedding layer
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=pad_idx)
        # Initialising the linear layer
        self.fc = nn.Linear(embedding_dim, output_dim)
    def forward(self, text):
        embedded = self.embedding(text)
        embedded = embedded.permute(1, 0, 2)
        # Averaging the sentences in 2d
        pooled = fn.avg_pool2d(embedded, (embedded.shape[1], 1)).squeeze(1)
        return self.fc(pooled)
inputDimension = len(TEXT.vocab)
embeddingDimension = 100
outputDimension = 1
padding = TEXT.vocab.stoi[TEXT.pad_token]
model = myModel(inputDimension, embeddingDimension, outputDimension, padding)
```

In [ ]:

```
# Copying the pre-trained vectors to the embedding layer
pretrained_embeddings = TEXT.vocab.vectors
model.embedding.weight.data.copy_(pretrained_embeddings)
# Setting the unknown and padding elements to zero since they are of no use
unknown = TEXT.vocab.stoi[TEXT.unk_token]
model.embedding.weight.data[unknown] = torch.zeros(embeddingDimension)
model.embedding.weight.data[padding] = torch.zeros(embeddingDimension)
# Counting the number of trainable parameters
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)
print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 5,000,301 trainable parameters
```

In [ ]:

```
import time
import torch.optim as optim
optimizer = optim.Adam(model.parameters())
criterion = nn.BCEWithLogitsLoss()
# Defining function to calculate accuracy
def binary_accuracy(preds, y):
    rounded_preds = torch.round(torch.sigmoid(preds))
    correct = (rounded_preds == y).float()
    acc = correct.sum() / len(correct)
    return acc
# Defining function for traning
def train(model, iterator, optimizer, criterion):
    epoch_loss = 0
    epoch_acc = 0
    model.train()
    for batch in iterator:
        optimizer.zero_grad()
        predictions = model(batch.c).squeeze(1)
        loss = criterion(predictions, batch.s)
        acc = binary_accuracy(predictions, batch.s)
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()
        epoch_acc += acc.item()
    return epoch_loss / len(iterator), epoch_acc / len(iterator)
# Defining function for testing
def evaluate(model, iterator, criterion):
    epoch_loss = 0
    epoch_acc = 0
    model.eval()
    with torch.no_grad():
        for batch in iterator:
            predictions = model(batch.c).squeeze(1)
            loss = criterion(predictions, batch.s)
            acc = binary_accuracy(predictions, batch.s)
            epoch_loss += loss.item()
            epoch_acc += acc.item()
    return epoch_loss / len(iterator), epoch_acc / len(iterator)
# Defining function to calculate time
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

In [ ]:

```
# Traning the model
N_EPOCHS = 5
best_valid_loss = float('inf')
for epoch in range(N_EPOCHS):
    start_time = time.time()
    train_loss, train_acc = train(model, train_iterator, optimizer, criterion)
    valid_loss, valid_acc = evaluate(model, valid_iterator, criterion)
    end_time = time.time()
    epoch_mins, epoch_secs = epoch_time(start_time, end_time)
    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'myModelBigram.pt')
    print(f'Epoch: {epoch+1:02}')
    print(f'\tEpoch Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train Acc: {train_acc*100:.2f}%')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. Acc: {valid_acc*100:.2f}%')
```

Epoch: 01

```
    Epoch Time: 18m 36s
    Train Loss: 0.377 | Train Acc: 84.68%
     Val. Loss: 0.309 |  Val. Acc: 87.11%
```

Epoch: 02

```
    Epoch Time: 18m 56s
    Train Loss: 0.288 | Train Acc: 88.30%
     Val. Loss: 0.295 |  Val. Acc: 88.20%
```

Epoch: 03

```
    Epoch Time: 18m 51s
    Train Loss: 0.277 | Train Acc: 88.98%
     Val. Loss: 0.295 |  Val. Acc: 88.33%
```

Epoch: 04

```
    Epoch Time: 18m 47s
    Train Loss: 0.271 | Train Acc: 89.25%
     Val. Loss: 0.293 |  Val. Acc: 88.59%
```

Epoch: 05

```
    Epoch Time: 18m 59s
    Train Loss: 0.267 | Train Acc: 89.48%
     Val. Loss: 0.294 |  Val. Acc: 88.56%
```

In [ ]:

```
# Testing the model
model.load_state_dict(torch.load('myModelBigram.pt'))
test_loss, test_acc = evaluate(model, test_iterator, criterion)
# Calculating test accuracy
print(f'Test Loss: {test_loss:.3f} | Test Acc: {test_acc*100:.2f}%')
```

```
Test Loss: 0.286
Test Acc: 88.86%
```