> importing the required libraries and the nltk resources (install BeautifulSoup and Contractions first if it's not already installed)

```
import pandas as pd
import numpy as np
import nltk

from nltk.tokenize import word_tokenize

nltk.download('punkt')
nltk.download('wordnet')
nltk.download('stopwords')
nltk.download('omw-1.4')

import re
from bs4 import BeautifulSoup
import contractions

import warnings
warnings.filterwarnings("ignore")
```

```
[nltk_data] Downloading package punkt to /Users/rutuja/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package wordnet to /Users/rutuja/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
[nltk_data] Downloading package stopwords to
[nltk_data]     /Users/rutuja/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package omw-1.4 to /Users/rutuja/nltk_data...
[nltk_data]   Package omw-1.4 is already up-to-date!
```

```
! pip install bs4 # in case you don't have it installed
! pip install contractions
# Dataset: https://s3.amazonaws.com/amazon-reviews-pds/tsv/amazon_reviews_us_Beauty_v1
```

```
Requirement already satisfied: bs4 in /Users/rutuja/opt/anaconda3/lib/python3.9/s
Requirement already satisfied: beautifulsoup4 in /Users/rutuja/opt/anaconda3/lib/
Requirement already satisfied: soupsieve>1.2 in /Users/rutuja/opt/anaconda3/lib/p
Requirement already satisfied: contractions in /Users/rutuja/opt/anaconda3/lib/py
Requirement already satisfied: textsearch>=0.0.21 in /Users/rutuja/opt/anaconda3/
Requirement already satisfied: pyahocorasick in /Users/rutuja/opt/anaconda3/lib/p
Requirement already satisfied: anyascii in /Users/rutuja/opt/anaconda3/lib/python
```

Read Data

> Read the dataset using pandas read_csv function and sepcify the delimiter as tab.
> Clean the dataset by skipping lines with delimiter issues

```
data = pd.read_csv("amazon_reviews_us_Beauty_v1_00.tsv", sep = '\t', on_bad_lines='ski
```

```
data.head()
```

	marketplace	customer_id	review_id	product_id	product_parent	produc
0	US	1797882	R3I2DHQBR577SS	B001ANOOOE	2102612	The N Mc Sun
1	US	18381298	R1QNE9NQFJC2Y4	B0016J22EQ	106393691	Alba Sunles Lotion,
2	US	19242472	R3LIDG2Q4LJBAO	B00HU6UQAG	375449471	Elyse Skir E
3	US	19551372	R3KSZHPAEVPEAL	B002HWS7RM	255651889	Di Color, I C F
4	US	14802407	RAI2OIG50KZ43	B00SM99KWU	116158747	Biore Ric SPF50+

Keep Reviews and Ratings

> Keep only required columns (reviews and ratings)

```
data = data[["review_body", "star_rating"]]
data
```

	review_body	star_rating
0	Love this, excellent sun block!!	5
1	The great thing about this cream is that it do...	5
2	Great Product, I'm 65 years old and this is al...	5
3	I use them as shower caps & conditioning caps....	5
4	This is my go-to daily sunblock. It leaves no ...	5
...	...	...
5094302	After watching my Dad struggle with his scisso...	5
5094303	Like most sound machines, the sounds choices a...	3
5094304	I bought this product because it indicated 30 ...	5
5094305	We have used Oral-B products for 15 years; thi...	5

We form three classes and select 20000 reviews randomly from each class.

> 1. Check if there are any missing values in the dataframe and drop those rows.
> 2. Convert all ratings to integer datatype to maintain uniformity

```
data.isnull().sum()
```

```
review_body    400
star_rating     10
dtype: int64
```

```
data.dropna(inplace = True)
```

```
data = data.astype({'star_rating': 'int'})
data['star_rating'].value_counts()
```

```
5    3240662
4     738494
1     455028
3     396760
2     262963
Name: star_rating, dtype: int64
```

> Creating 3 classes based on ratings (rating 1, 2 => class 1; rating 3 => class 2; rating 4, 5 => class 3).
> I've created a pandas conditional column called "class" using numpy.select() to set the respective values (rating_class) using a list of conditions called rating_conditions

```
rating_conditions = [
    (data['star_rating'] <= 2),
    (data['star_rating'] > 2) & (data['star_rating'] < 4),
    (data['star_rating'] > 3)
]
rating_class = [1, 2, 3]
data['class'] = np.select(rating_conditions, rating_class)
data['class'].value_counts()
```

```
3    3979156
1     717991
2     396760
Name: class, dtype: int64
```

> selecting 20,000 random reviews from each class to create a balanced dataset.
> I've used the pandas Series.sample function to achieve this, i've grouped the dataframe according to classes and then selected 20000 random reviews. Pandas series sample function allows us to view the original index of the row, and reset_index function is used to serially index all these randomly selected rows

```
np.random.seed(0)
data = data.groupby(['class'])['review_body'].apply(pd.Series.sample, n=20000).reset_i
data['class'].value_counts()
```

```
1    20000
2    20000
3    20000
Name: class, dtype: int64
```

```
data
```

	class	level_1	review_body
0	1	124207	I generally don't write the reviews, but this ...
1	1	3083395	I have extremely light eyebrows, however my ha...
2	1	4179658	I'm disappointed to find out that real Shea bu...
3	1	4525912	I bought this product at a hair salon for $19 ...
4	1	3099231	Ive been a heavy perspirator for soo long now ...
...	...	...	...
59995	3	3362667	I ordered the No No around the end of May 2013...
59998	3	1641782	These attach to the fingernail with ease. Jus...

Data Cleaning

Pre-processing

```
# print average length of reviews before cleaning
data['review_length'] = data['review_body'].str.len()
review_len_before_cleaning = data['review_length'].mean()
```

> Performing the preprocessing steps on the data to improve the performance by removing the unwanted information, as it does not add any value in predicting the class of ratings. Steps preformed are:
>
> 1. conversion of all reviews to lowercase using the string function lower()
> 2. removing URLs and links from the reviews using regular expressions
> 3. removing all special characters from the reviews using regular expressions
> 4. removing the extra spaces from the reviews using the string function strip()
> 5. perform contractions using the contractions library to get the original words

```
# Convert all reviews to lowercase
data['review_body'] = data['review_body'].str.lower()

# Remove HTML and URLs from the reviews
data['review_body'] = data['review_body'].apply(lambda x: re.sub(r'(<.*?>|https?://\S+
#data['review_body'] = data['review_body'].apply(lambda x: BeautifulSoup(x, 'lxml').ge

# remove non-alphabetical characters
data['review_body'] = data['review_body'].apply(lambda x: re.sub('[^a-zA-Z]', ' ', x))
```

```
# remove extra spaces
data['review_body'] = data['review_body'].str.strip()

# Perform contractions on the reviews
data['review_body'] = data['review_body'].apply(lambda x: contractions.fix(x))
```

> Print the average length of reviews before and after cleaning to get an idea of how much the reviews have shortened by the cleaning steps

```
# Print average length of reviews before and after cleaning
review_lengths = data['review_body'].str.len()
review_len_after_cleaning = review_lengths.mean()
print("Average review length before and after cleaning:", review_len_before_cleaning,'
```

```
Average review length before and after cleaning: 269.42358333333334 , 266.15375
```

remove the stop words

> create a list of all the english stop words using NLTK's stopwords corpus, and then retain all other words in the reviews which do not match with the stopwords list i.e retain only meaningful information

```
from nltk.corpus import stopwords

# remove stopwords
stopwords_list = stopwords.words('english')
data['review_body'] = data['review_body'].apply(lambda x: ' '.join([word for word in x
```

perform lemmatization

> perform lemmatization on the remaining words after stopword removal using NLTK's word net lemmatizer for all the words which match the words obtained by tokenizing sentences into words by word_tokenize function

```
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
data['review_body'] = data['review_body'].apply(lambda x: ' '.join([lemmatizer.lemmati
```

> Print the average length of reviews before and after preprocessing to get an idea of how much the reviews have shortened by the preprocessing steps

```
# Print average length of reviews before and after preprocessing
review_lengths = data['review_body'].str.len()
review_len_after_preprocessing = review_lengths.mean()
print("Average review length before and after preprocessing:", review_len_after_cleani
```

```
Average review length before and after preprocessing: 266.15375 , 155.4249
```

TF-IDF Feature Extraction

> For feature extraction create a tf-idf feature matrix from review_body column using TfidfVectorizer class from sklearn which computes the tf-idf weights. It converts our collection of reviews to a matrix of TF-IDF numerical features
> Split the dataset into 80% training and 20% testing set, X is the feature vector and Y is the class column. To keep the proportion of the three classes balanced, use stratify parameter on the class column

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split

# create the tf-idf feature matrix
tfidf = TfidfVectorizer()
features = tfidf.fit_transform(data['review_body'])

# split the dataset into training (80%) and testing (20%) sets
X_train, X_test, y_train, y_test = train_test_split(features, data['class'], stratify=
```

Perceptron

> Train a Perceptron model using sklearn on the training dataset, make predictions on the testing dataset. As this is a multi-class classification problem, report the precision, recall, and f1-score for each class and their averages, as well as the accuracy

```
from sklearn.linear_model import Perceptron
from sklearn.metrics import precision_recall_fscore_support, accuracy_score

# Initialize the Perceptron model
clf = Perceptron(tol=1e-3, random_state=42)
#clf = Perceptron()

# Train the model on the training dataset
clf.fit(X_train, y_train)

# Make predictions on the testing dataset
y_pred = clf.predict(X_test)

# Compute the precision, recall, and f1-score per class
precision, recall, f1, _ = precision_recall_fscore_support(y_test, y_pred, average=Nor

# Compute the average precision, recall, and f1-score
average_precision = precision.mean()
average_recall = recall.mean()
average_f1 = f1.mean()

# Compute the accuracy
acc = accuracy_score(y_test, y_pred)

# Print the precision, recall, f1-score per class
print("Class 1:", precision[0],",", recall[0],",", f1[0])
print("Class 2:", precision[1],",", recall[1],",", f1[1])
print("Class 3:", precision[2],",", recall[2],",", f1[2])

# Print the average precision, recall, f1-score
print("Average of each metric:", average_precision, ",", average_recall, ",", average_

# Print the accuracy
print("Accuracy:", acc)
```

```
Class 1: 0.6107331821617535 , 0.606 , 0.608357384866357
Class 2: 0.5079485238455715 , 0.50325 , 0.5055883461007158
Class 3: 0.6647000983284169 , 0.676 , 0.6703024293505206
Average of each metric: 0.5944606014452473 , 0.5950833333333333 , 0.594749386772
Accuracy: 0.5950833333333333
```

SVM

> Train Support Vector Machine (SVM) model on the training dataset, make predictions on the testing dataset. As LinearSVC scales better to large numbers of samples, I've used LinearSVC with GridSearchCV to find the optimal parameters of the model, which can decrease the runtime of the code by performing a grid search over the

> specified parameter grid and use 5 fold cross-validation to evaluate the performance of the model for each combination of parameters. The best parameters are the ones that result in the highest accuracy (as specified, scoring method = accuracy)

```
from sklearn.svm import LinearSVC
from sklearn.model_selection import GridSearchCV

# Create an instance of the LinearSVC classifier
svc = LinearSVC()

# Define the parameter grid to search
param_grid = {'C': [0.1, 1, 10], 'loss': ['hinge', 'squared_hinge']}

# Create an instance of the GridSearchCV class
grid_search = GridSearchCV(svc, param_grid, cv=5, scoring='accuracy')

# Fit the grid search to the data
grid_search.fit(X_train, y_train)

# Get the best parameters
best_params = grid_search.best_params_

# Re-train the model with the best parameters
svc = LinearSVC(**best_params)
svc.fit(X_train, y_train)

# Make predictions on the testing dataset
y_pred = svc.predict(X_test)

# Compute the precision, recall, and f1-score per class
precision, recall, f1, _ = precision_recall_fscore_support(y_test, y_pred, average=Nor

# Compute the average precision, recall, and f1-score
average_precision = precision.mean()
average_recall = recall.mean()
average_f1 = f1.mean()

# Compute the accuracy
acc = accuracy_score(y_test, y_pred)

# Print the precision, recall, f1-score per class
print("Class 1:", precision[0],",", recall[0],",", f1[0])
print("Class 2:", precision[1],",", recall[1],",", f1[1])
print("Class 3:", precision[2],",", recall[2],",", f1[2])

# Print the average precision, recall, f1-score
print("Average of each metric:", average_precision, ",", average_recall, ",", average_
```

```
# Print the accuracy
print("Accuracy: 0.678")
```

```
Class 1: 0.6808510638297872 , 0.704 , 0.6922320550639135
Class 2: 0.6012435793457691 , 0.556 , 0.5777373684894143
Class 3: 0.7433373349339736 , 0.774 , 0.7583588487446418
Average of each metric: 0.6751439927031765 , 0.6779999999999999 , 0.676109424099
Accuracy: 0.678
```

Logistic Regression

> Train a Logistic Regression model using sklearn on the training dataset, make predictions on the testing dataset. As this is a multi-class classification problem, report the precision, recall, and f1-score for each class and their averages, as well as the accuracy

```
from sklearn.linear_model import LogisticRegression

# Initialize the Logistic Regression model
logreg = LogisticRegression(random_state=42)

# Train the model on the training dataset
logreg.fit(X_train, y_train)

# Make predictions on the testing dataset
y_pred = logreg.predict(X_test)

# Compute the precision, recall, and f1-score per class
precision, recall, f1, _ = precision_recall_fscore_support(y_test, y_pred, average=Nor

# Compute the average precision, recall, and f1-score
average_precision = precision.mean()
average_recall = recall.mean()
average_f1 = f1.mean()

# Compute the accuracy
acc = accuracy_score(y_test, y_pred)

# Print the precision, recall, f1-score per class
print("Class 1:", precision[0],",", recall[0],",", f1[0])
print("Class 2:", precision[1],",", recall[1],",", f1[1])
print("Class 3:", precision[2],",", recall[2],",", f1[2])

# Print the average precision, recall, f1-score
print("Average of each metric:", average_precision, ",", average_recall, ",", average_

# Print the accuracy
print("Accuracy:", acc)
```

```
Class 1: 0.6816625916870416 , 0.697 , 0.6892459826946848
Class 2: 0.5867768595041323 , 0.568 , 0.5772357723577235
Class 3: 0.7535908865775136 , 0.76075 , 0.7571535207763126
Average of each metric: 0.6740101125895626 , 0.67525 , 0.6745450919429069
Accuracy: 0.67525
```

Naive Bayes

> Train a Naive Bayes model using sklearn on the training dataset, make predictions on the testing dataset. I've used Multinomial Naive Bayes model as it is suitabe for classification with discrete features like text classification. As this is a multi-class classification problem, report the precision, recall, and f1-score for each class and their averages, as well as the accuracy

```
from sklearn.naive_bayes import MultinomialNB

# train the Multinomial Naive Bayes model on the training dataset
mnb = MultinomialNB()
mnb.fit(X_train, y_train)

# predict labels for the testing dataset
y_pred = mnb.predict(X_test)

# Compute the precision, recall, and f1-score per class
precision, recall, f1, _ = precision_recall_fscore_support(y_test, y_pred, average=Nor

# Compute the average precision, recall, and f1-score
average_precision = precision.mean()
average_recall = recall.mean()
average_f1 = f1.mean()

# Compute the accuracy
acc = accuracy_score(y_test, y_pred)

# Print the precision, recall, f1-score per class
print("Class 1:", precision[0],",", recall[0],",", f1[0])
print("Class 2:", precision[1],",", recall[1],",", f1[1])
print("Class 3:", precision[2],",", recall[2],",", f1[2])

# Print the average precision, recall, f1-score
print("Average of each metric:", average_precision, ",", average_recall, ",", average_

# Print the accuracy
print("Accuracy:", acc)
```

```
Class 1: 0.6977813852813853 , 0.64475 , 0.6702182952182952
Class 2: 0.5691207015924302 , 0.6165 , 0.5918636745469819
Class 3: 0.7328128934777134 , 0.7275 , 0.7301467820850583
Average of each metric: 0.6665716601171764 , 0.6629166666666667 , 0.664076250616
Accuracy: 0.6629166666666667
```