Pipeline

```
1) Removing HTML tagas and URLs, Punctiation*, Replacing emoticons
*.
2) Tokenization
3) Removing Stop Words
4) Splitting data: Training, Validation, Test
5) TF-IDF Calculation
```

In previous code I created another pipeline, it happens that if I apply TF-IDF I can avoid a many other preprocessing steps.

Note about the code:

```
Note that the dataset takes two paths, one towards the sentiment an
alysis
and the other towards the analysis of some statistics related to th
e words/text
```

In [1]:

```python
# Multilayer perceptron working environment.
# Getting ready the work environment. Importing libraries and modules:
import time
import pandas as pd
import re
import nltk
import torch
import torch.nn as nn
import numpy as np
import string
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.feature_extraction.text import CountVectorizer, TfidfVecto
from sklearn.model_selection import train_test_split
from sklearn.metrics import precision_recall_fscore_support, classifica
from collections import Counter
from bs4 import BeautifulSoup
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize

#===========          Extra tools for the statistic analysis
from nltk.stem import WordNetLemmatizer

lemmatizer = WordNetLemmatizer()
#-----------------------------------------------------------------------

stop_words = stopwords.words('english')
tfidf_vectorizer = TfidfVectorizer()
vectorizer = CountVectorizer()
```

```
KeyboardInterrupt
```

In [ ]:

```python
# Support Vector Machine working environment.
import re
import nltk
import torch
import torch.nn as nn
import numpy as np
import string

from sklearn.feature_extraction.text import CountVectorizer, TfidfVecto
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score, classification_report
from sklearn.model_selection import train_test_split, GridSearchCV
from collections import Counter
from bs4 import BeautifulSoup


tfidf_vectorizer = TfidfVectorizer(stop_words='english')
vectorizer = CountVectorizer()
```

1) Importing data set

In [ ]:

```
#Importing dataset
imdb_path = 'IMDB.csv'
imdb = pd.read_csv(imdb_path)

#Convert sentiment column to binary class
imdb['sentiment'] = imdb['sentiment'].map({'positive': 1, 'negative': 0

#Checking data and columns
print(imdb.head())
```

2) Reducing size

In [ ]:

```
#During the applycation of the model in earlier stages, the machine pro
#Hence data will be cut off to 5000 rows:

#Firstly, let's segregate the sentiment column:
positive_reviews = imdb[imdb['sentiment'] == 1]
negative_reviews = imdb[imdb['sentiment'] == 0]

#Secondly, sampling randomly 2500 reviews from each (+/-)
positive_sample = positive_reviews.sample(n=2500, random_state=42)
negative_sample = negative_reviews.sample(n=2500, random_state=42)

#Putting them together again
imdb_reduced = pd.concat([positive_sample, negative_sample])

#Suffling the new dataset
imdb_reduced = imdb_reduced.sample(frac=1, random_state=42).reset_index


#Sources:
# https://stackoverflow.com/questions/71758460/effect-of-pandas-datafra
# https://stackoverflow.com/questions/57300260/how-to-drop-added-column
# https://docs.python.org/3/library/fractions.html
# https://datascience.stanford.edu/news/splitting-data-randomly-can-rui
# https://stats.stackexchange.com/questions/484000/how-to-appropriately
```

3) Preprocessing

3.1) Removing HTML tags and URLs, lower

Note: I used a function to get rid of the punctuations however the dataset became massive and my machine was unable to manage. That's why I am avoiding it.

In [ ]:
```
#Function to remove HTML tags:
def remove_html(text):
    soup = BeautifulSoup(text, "html.parser")
    return soup.get_text()

#Sources:
#https://stackoverflow.com/questions/328356/extracting-text-from-html-f
#https://beautiful-soup-4.readthedocs.io/en/latest/
#https://www.datacamp.com/tutorial/web-scraping-using-python
#https://www.geeksforgeeks.org/how-to-write-the-output-to-html-file-wit
```

In [ ]:
```
#Function to remove URLs characteres:
def remove_urls(text):
    return re.sub(r'https?://\S+|www\.\S+', '', text)

#Source: https://www.geeksforgeeks.org/remove-urls-from-string-in-pytho
```

In [ ]:
```
#Function to put together all the previous functions:
def preprocess_1(text):
    text = remove_html(text)
    text = remove_urls(text)
    text = text.lower()
    #text = remove_punctuation(text)///\\\\Initially used a function to
    return text
```

In [ ]:
```
#Running the function to make the first preprocessing step.
imdb_reduced['review'] = imdb_reduced['review'].apply(preprocess_1)
imdb['review_preprocess_1'] = imdb['review'].apply(preprocess_1)
```

3.2) Tokenization and stopwords elimination.

In [ ]:
```
#Function to tokenize and convert to lower case the text in review colu
def tokenize(text):
    tokens = re.findall(r'\b\w+\b', text)
    return tokens

#Tokenization
imdb_reduced['Token'] = imdb_reduced['review'].apply(tokenize)
imdb['Token'] = imdb['review_preprocess_1'].apply(tokenize)
```

In [ ]:
```
#Function to remove stop words from the tokenized review column
def remove_stopwords(tokens):
    filtered_tokens = [word for word in tokens if word not in stop_word
    return filtered_tokens

#Remove stopwords
imdb_reduced['Token'] = imdb_reduced['Token'].apply(remove_stopwords)
imdb['Token'] = imdb['Token'].apply(remove_stopwords)
```

** Statistic Analysis **

....................Starts

A) Checking text

In [ ]:
```
print(imdb.head())
```

B) Avg Words Positive Vs Negative:

In [ ]:
```
#Calculating the total tokens for each review
imdb['token_count'] = imdb['Token'].apply(lambda x: len(x) if isinstanc

#Dispersion and central tendency measurements
statistics = imdb.groupby('sentiment')['token_count'].agg(['min', 'max'

#Avg words per review:
avg_words = imdb['Token'].apply(len).mean()

#Print the statistics
print("Statistics by Sentiment: ")
print('\n')
print(statistics)
print('\n')
print('\n')
print('Average Words: ', f"{avg_words:.0f}")

#Resources:
#https://www.geeksforgeeks.org/pandas-groupby-one-column-and-get-mean-m
#https://www.kaggle.com/code/akshaysehgal/ultimate-guide-to-pandas-grou
```

C) Word Frequency

In [ ]:
```
#Iterating through the list of lists(each row) to create a new list wit
def word_freq(list_of_list):
    single_list = [item for sublist in list_of_list for item in sublist
    token_freq = Counter(single_list)
    return token_freq

#Counting the frequency for each word.
word_frequency = word_freq(imdb['Token'])
print(word_frequency)

#Sources: https://www.datacamp.com/tutorial/pandas-apply
```

D) Unique Words

In [ ]:
```
unique_words = len(word_frequency.keys())
print('Unique_words: ',f'{unique_words}')
```

E) Most common words

In [ ]:

```
positive_words = Counter()
negative_words = Counter()

for index, row in imdb.iterrows():
    words = row['Token']
    sentiment = row['sentiment']
    if sentiment == 1:
        positive_words.update(words)
    else:
        negative_words.update(words)

#Resources:
#https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.iterrows
#https://www.kaggle.com/code/juicykn/imdb-movie-list-analysis-in-python
```

In [ ]:

```
top_positive_words = positive_words.most_common(10)
top_negative_words = negative_words.most_common(10)

print('Positive: ', top_positive_words)
print('\n')
print('Negative: ', top_negative_words)
```

In [ ]:

```
#Spliting the tupple we got earlier
positive_words, positive_counts = zip(*top_positive_words)
negative_words, negative_counts = zip(*top_negative_words)

#Charts-----------------------------------------------------
fig, axs = plt.subplots(2,1,figsize=(10,8))

#Positive words plot
axs[0].bar(positive_words, positive_counts, color='green')
axs[0].set_title('Most Frequent Positive Words')
axs[0].set_ylabel('Frequency')

#Negative words plot
axs[1].bar(negative_words, negative_counts, color='red')
axs[1].set_title('Most Frequent Negative Words')
axs[1].set_ylabel('Frequency')

#Space between charts
plt.tight_layout(pad=4.0)
plt.show()

#Resources:
# https://realpython.com/python-zip-function/#using-zip-in-python
# https://matplotlib.org/stable/index.html
```

** Statistic Analysis **

..................................Finishes

4) Splitting Data

In [ ]:

```
#Splitting data into train 70%, validation 15%, test 15%

X_train_val, X_test, y_train_val, y_test = train_test_split(imdb_reduce

X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_
```

5) TF-IDF Calculation

In [ ]:

```
#Feature extraction: Transforming data into TF-IDF features.
X_train = tfidf_vectorizer.fit_transform(X_train)
X_val = tfidf_vectorizer.transform(X_val)
X_test = tfidf_vectorizer.transform(X_test)
```

In [ ]:

```
print(X_train.shape)  # Should output (number_of_samples, 33154)
print(X_test.shape)   # Should also output (number_of_samples, 33154)
print(X_val.shape)
```

6) Format conversion

In [ ]:

```
#Turning sparse matrix into dense
X_train = X_train.toarray()
X_val = X_val.toarray()
X_test = X_test.toarray()

#Turning into PyTorch tensors
X_train = torch.tensor(X_train, dtype=torch.float32)
X_val = torch.tensor(X_val, dtype=torch.float32)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_train = torch.tensor(y_train.values, dtype=torch.float32)
y_val = torch.tensor(y_val.values, dtype=torch.float32)
y_test = torch.tensor(y_test.values, dtype=torch.float32)

#Resources:
# https://pytorch.org/docs/stable/tensors.html
```

In [ ]:

```
#We need to know the shape of the input vector to set the in put dimens
input_dim = X_train.shape[1]
print(input_dim)
```

7) MLP model

In [ ]:

```
# Time consumed (starts)
start_time = time.time()

#Building the Multilayer Perceptron model with back propagation.
class MLPmodel(nn.Module):
    def __init__(self):
        super(MLPmodel, self).__init__()
        self.fc1 = nn.Linear(33154,610)
        self.fc2 = nn.Linear(610,377)
        self.fc3 = nn.Linear(377,23)
        self.fc4 = nn.Linear(23,1)
        self.sigmoid = nn.Sigmoid()
        self.relu = nn.ReLU()

    def forward(self, x):
        hidden = self.relu(self.fc1(x))
        hidden = self.relu(self.fc2(hidden))
        hidden = self.relu(self.fc3(hidden))
        output = self.sigmoid(self.fc4(hidden))
        return output

```

In [ ]:

```
model = MLPmodel()
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.00001, weight_dec

#Note: SDG was used earlier in during the experimentation, however, the
```

In [ ]:

```
epochs = 5000
patience = 10 #Here we define how many epochs wait until we stop
best_val_loss = float('inf') #To save the best model/early stop
patience_counter = 0 #This one starts a counter to track number of epoc

for epoch in range(epochs):
    model.train()
    optimizer.zero_grad()
  #Forward Pass
    outputs = model(X_train)
    loss = criterion(outputs.squeeze(), y_train)
  #Backward and Optimize
    loss.backward()
    optimizer.step()

  #Validation
    model.eval()
    with torch.no_grad():
        val_outputs = model(X_val)
        val_loss = criterion(val_outputs.squeeze(), y_val)

  #Early stop
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        best_model_state = model.state_dict()  # Save the best model st
        patience_counter=0
    else:
        patience_counter += 1

  #Print the early stopping
    if patience_counter > patience:
        print(f'Stopping early at epoch {epoch+1}.')
        break

    if (epoch+1) % 5 == 0:
        print(f'Epoch {epoch+1}/{epochs}, Loss: {loss.item()}, Val Loss

if best_model_state:
    model.load_state_dict(best_model_state)

#Resources:
#https://pythonguides.com/pytorch-early-stopping/
#https://discuss.pytorch.org/t/can-i-deepcopy-a-model/52192
#https://machinelearningmastery.com/how-to-stop-training-deep-neural-ne
#https://stackoverflow.com/questions/71998978/early-stopping-in-pytorch
#https://github.com/Bjarten/early-stopping-pytorch
#https://debuggercafe.com/using-learning-rate-scheduler-and-early-stopp
```

In [ ]:

```
#Accuracy on validation set
model.eval()
with torch.no_grad():
    val_outputs = model(X_val)
    val_predicted_bin = (val_outputs.squeeze() > 0.5).int()

    #validation accuracy
    val_accuracy = accuracy_score(y_val.numpy(), val_predicted_bin.nump
    print(f'Validation set accuracy: {val_accuracy:.2f}')

# Resources (for the "bin"):
# https://scikit-learn.org/stable/modules/generated/sklearn.metrics.cla
# https://pytorch.org/docs/stable/generated/torch.Tensor.numpy.html
# https://www.youtube.com/watch?v=E35CVhVKISA&t=135s
```

In [ ]:

```
#Accuracy on test set
model.eval()

with torch.no_grad():
    test_outputs = model(X_test)
    predicted_bin = (test_outputs.squeeze() > 0.5).int()

#accuracy
test_accuracy = accuracy_score(y_test.numpy(), predicted_bin.numpy())
print(f'Accuracy on test set: {test_accuracy:.2f}')

#classification report
print(classification_report(y_test.numpy(), predicted_bin.numpy()))
```

In [ ]:

```
#Turning tensors into NumPy arrays so we can use Scikit-Learn functions
test_outputs_np = test_outputs.squeeze().numpy()
y_test_np = y_test.numpy()

#Calculation of the ROC-AU
fpr, tpr, thresholds = roc_curve(y_test_np, test_outputs_np)
roc_auc = roc_auc_score(y_test_np, test_outputs_np)

#Plotting the ROC curve
plt.figure(figsize=(8, 6))

plt.plot(fpr, tpr, color='firebrick', lw=2, label='ROC curve (area = %0
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])

plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC-AU Chart: Best MLP Model')
plt.legend(loc="lower right")

plt.show()

#Calculation of the confusion matrix
cm = confusion_matrix(y_test.numpy(), predicted_bin.numpy())

#Plotting the confusion matrix
plt.figure(figsize=(8, 6))

sns.heatmap(cm, annot=True, fmt="d", cmap='seismic')
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('C.M. Chart: Best MLP Model')

plt.show()
```

In [ ]:

```
#Total Time Consumed
end_time = time.time()
execution_time = end_time - start_time
print(f"Total Execution Time: {execution_time} seconds")
```

8) SVM model

In [ ]:

```
# Time Consumed (starts)
start_time = time.time()

# Definition of the SVM model and hyperparameter for tuning on the trai
param_grid = {'C':  [1.25892541179416], 'kernel': ['linear'], 'tol' : [

# Hyperparameters tuning, cross-validation using training set.
grid_search = GridSearchCV(SVC(), param_grid, cv=3, scoring='accuracy',
grid_search.fit(X_train, y_train)

# Getting the best estimator
Best_SVMmodel = grid_search.best_estimator_
Best_score = grid_search.best_score_

# Print results
print(f'Best Model: {Best_SVMmodel}')
print('\n')
print(f'Best CV Score: {Best_score:.2f}')
```

In [ ]:

```
# Evaluation of the model on the validation set witht he best parameter
y_pred = Best_SVMmodel.predict(X_val)
val_accuracy = accuracy_score(y_val, y_pred)

# Print results
print(f'Validation set accuracy: {val_accuracy:.2f}')
```

In [ ]:

```
# Evaluation of the final model using the test set
y_pred = Best_SVMmodel.predict(X_test)
test_accuracy = accuracy_score(y_test, y_pred)

#Print results
print(f'Accuracy on test set: {test_accuracy:.2f}')
print(classification_report(y_test, y_pred))
```

In [ ]:

```
# Predict probabilities for the test set
y_pred_proba = Best_SVMmodel.decision_function(X_test)

# Calculation of the ROC-AUC
fpr, tpr, thresholds = roc_curve(y_test, y_pred_proba)
roc_auc = roc_auc_score(y_test, y_pred_proba)

# Plotting the ROC curve
plt.figure(figsize=(8, 6))
plt.plot(fpr, tpr, color='firebrick', lw=2, label='ROC curve (area = %0
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.0])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC-AU Chart: SVM Model')
plt.legend(loc="lower right")
plt.show()

# Calculation of the confusion matrix
cm = confusion_matrix(y_test, y_pred)

# Plotting the confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt="d", cmap='seismic')
plt.xlabel('Predicted labels')
plt.ylabel('True labels')
plt.title('C.M. Chart: SVM Model')
plt.show()

```

In [ ]:

```
# Total Time Consumed
end_time = time.time()
execution_time = end_time - start_time
print(f"Total Execution Time: {execution_time} seconds")
```