```python
# ----------------------------------------------- py_functions.py
------------------------------------------------------------------------------------
######################################
#
#  Functions used for this project
#

def descriptive_tokens(df, name):
    '''
    '''
    all_words = [word for tokens in df[name] for word in tokens]
    sentence_lengths = [len(tokens) for tokens in df[name]]
    VOCAB = sorted(list(set(all_words)))
    print('Column ', name,  " have %s words total, with a vocabulary size of %s" %
(len(all_words), len(VOCAB)))
    print("Max sentence length is %s" % max(sentence_lengths))
    return sentence_lengths




from sklearn.decomposition import PCA, TruncatedSVD
import matplotlib
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt


def plot_LSA(data, text_labels, plot = True):
    '''
    Dimensionality reduction using truncated SVD (aka LSA).

    This transformer performs linear dimensionality reduction by means of
truncated singular value decomposition (SVD).
    Contrary to PCA, this estimator does not center the data before computing the
singular value decomposition.
    This means it can work with scipy.sparse matrices efficiently.
    '''
    lsa = TruncatedSVD(n_components=2)
    lsa.fit(data)
    lsa_scores = lsa.transform(data)
    color_mapper = {label:idx for idx,label in enumerate(set(text_labels))}
    color_column = [color_mapper[label] for label in text_labels]
    colors = ['orange','blue','red']
    if plot:
        plt.scatter(lsa_scores[:,0], lsa_scores[:,1], s=75, alpha=.1,
c=text_labels, cmap=matplotlib.colors.ListedColormap(colors))
        orange_patch = mpatches.Patch(color='orange', label='Digital_Software')
        green_patch = mpatches.Patch(color='blue', label='Digital_Video_Games')
        red_patch = mpatches.Patch(color='red', label='Software')
        plt.legend(handles=[orange_patch, green_patch, red_patch], prop={'size':
30})


#### EVALUATION

from sklearn.metrics import accuracy_score, f1_score, precision_score,
recall_score, classification_report
def get_metrics(y_test, y_predicted):
    '''
    Evaluation metrics
    '''
    # true positives / (true positives+false positives)
    precision = precision_score(y_test, y_predicted, pos_label=None,
                                    average='weighted')
```

```
    # true positives / (true positives + false negatives)
    recall = recall_score(y_test, y_predicted, pos_label=None,
                          average='weighted')

    # harmonic mean of precision and recall
    f1 = f1_score(y_test, y_predicted, pos_label=None, average='weighted')

    # true positives + true negatives/ total
    accuracy = accuracy_score(y_test, y_predicted)
    return accuracy, precision, recall, f1


import numpy as np
import itertools
from sklearn.metrics import confusion_matrix

def plot_confusion_matrix(cm, classes,
                          normalize=False,
                          title='Confusion matrix',
                          cmap=plt.cm.Blues):
    """
    This function prints and plots the confusion matrix.
    Normalization can be applied by setting `normalize=True`.
    """
    if normalize:
        cm = cm.astype('float') / cm.sum(axis=1)[:, np.newaxis]
        print("Normalized confusion matrix")
    else:
        print('Confusion matrix, without normalization')

    plt.imshow(cm, interpolation='nearest', cmap=cmap)
    plt.title(title, fontsize=30)
    plt.colorbar()
    tick_marks = np.arange(len(classes))
    plt.xticks(tick_marks, classes, fontsize=20, rotation=45)
    plt.yticks(tick_marks, classes, fontsize=20)

    fmt = '.2f' if normalize else 'd'
    thresh = cm.max() / 2.

    for i, j in itertools.product(range(cm.shape[0]), range(cm.shape[1])):
        plt.text(j, i, format(cm[i, j], fmt), horizontalalignment="center",
                 color="white" if cm[i, j] < thresh else "black", fontsize=40)

    plt.tight_layout()
    plt.ylabel('True label', fontsize=30)
    plt.xlabel('Predicted label', fontsize=30)

    return plt

###########
#Word2vect
############

def get_average_word2vec(tokens_list, vector, generate_missing=False, k=300):
    if len(tokens_list)<1:
        return np.zeros(k)
    if generate_missing:
        vectorized = [vector[word] if word in vector else np.random.rand(k) for
word in tokens_list]
    else:
        vectorized = [vector[word] if word in vector else np.zeros(k) for word in
tokens_list]
    length = len(vectorized)
    summed = np.sum(vectorized, axis=0)
    averaged = np.divide(summed, length)
```

```
    return averaged

def get_word2vec_embeddings(vectors, df, name = 'tokens', generate_missing=False):
    embeddings = df[name].apply(lambda x: get_average_word2vec(x, vectors,
generate_missing=generate_missing))
    return list(embeddings)


### IMPORTANCE OF WORDS

def get_most_important_features(vectorizer, model, n=5):
    index_to_word = {v:k for k,v in vectorizer.vocabulary_.items()}

    # loop for each class
    classes ={}
    for class_index in range(model.coef_.shape[0]):
        word_importances = [(el, index_to_word[i]) for i,el in
enumerate(model.coef_[class_index])]
        sorted_coeff = sorted(word_importances, key = lambda x : x[0],
reverse=True)
        tops = sorted(sorted_coeff[:n], key = lambda x : x[0])
        bottom = sorted_coeff[-n:]
        classes[class_index] = {
            'tops':tops,
            'bottom':bottom
        }
    return classes



def plot_important_words(top_scores, top_words, name = 'top_words_software'):
    y_pos = np.arange(len(top_words))
    top_pairs = [(a,b) for a,b in zip(top_words, top_scores)]
    top_pairs = sorted(top_pairs, key=lambda x: x[1])

    top_words = [a[0] for a in top_pairs]
    top_scores = [a[1] for a in top_pairs]

    plt.barh(y_pos,top_scores, align='center', alpha=0.5)
    plt.title(name, fontsize=20)
    plt.yticks(y_pos, top_words, fontsize=14)
    #plt.suptitle(name, fontsize=16)
    plt.xlabel('Importance', fontsize=20)
    plt.show()


#---------------------------
#### importance word2vec
#---------------------------

import random
from collections import defaultdict
from lime.lime_text import LimeTextExplainer
import pandas as pd

def get_statistical_explanation(test_set, sample_size, word2vec_pipeline,
label_dict):
    sample_sentences = random.sample(test_set, sample_size)
    explainer = LimeTextExplainer()

    labels_to_sentences = defaultdict(list)
    contributors = defaultdict(dict)

    # First, find contributing words to each class
    for sentence in sample_sentences:
        probabilities = word2vec_pipeline([sentence])
```

```
        curr_label = probabilities[0].argmax()
        labels_to_sentences[curr_label].append(sentence)
        exp = explainer.explain_instance(sentence, word2vec_pipeline,
num_features=6, labels=[curr_label])
        listed_explanation = exp.as_list(label=curr_label)

        for word,contributing_weight in listed_explanation:
            if word in contributors[curr_label]:
                contributors[curr_label][word].append(contributing_weight)
            else:
                contributors[curr_label][word] = [contributing_weight]

    # average each word's contribution to a class, and sort them by impact
    average_contributions = {}
    sorted_contributions = {}
    for label,lexica in contributors.items():
        curr_label = label
        curr_lexica = lexica
        average_contributions[curr_label] = pd.Series(index=curr_lexica.keys())
        for word,scores in curr_lexica.items():
            average_contributions[curr_label].loc[word] = np.sum(np.array(scores))/
sample_size
        detractors = average_contributions[curr_label].sort_values()
        supporters = average_contributions[curr_label].sort_values(ascending=False)
        sorted_contributions[label_dict[curr_label]] = {
            'bottom':detractors,
             'tops': supporters
        }
    return sorted_contributions


#### Cleaning text

def standardize_text(df, text_field):
    df[text_field] = df[text_field].str.replace(r"-", "")
    df[text_field] = df[text_field].str.replace(r",", "")
    df[text_field] = df[text_field].str.replace(r"?", "")
    df[text_field] = df[text_field].str.replace(r"\(.*\)","")
    df[text_field] = df[text_field].str.replace(r"http\S+", "")
    df[text_field] = df[text_field].str.replace(r"http", "")
    df[text_field] = df[text_field].str.replace(r"@\S+", "")
    df[text_field] = df[text_field].str.replace(r"\n", "")
    df[text_field] = df[text_field].str.replace(r"[^A-Za-z0-9(),!?@\'\`\"\_\n]", "
")
    df[text_field] = df[text_field].str.replace(r"@", "at")
    df[text_field] = df[text_field].str.replace('[0-9]+', "")
    df[text_field] = df[text_field].str.lower()
    df[text_field] = df[text_field].str.replace(r"th", "")
    df[text_field] = df[text_field].str.replace(r"h", "")
    df[text_field] = df[text_field].str.replace(r"stars", "")
    df[text_field] = df[text_field].str.replace(r"star", "")
    df[text_field] = df[text_field].str.replace(r"one", "")
    df[text_field] = df[text_field].str.replace(r"two", "")
    df[text_field] = df[text_field].str.replace(r"three", "")
    df[text_field] = df[text_field].str.replace(r"four", "")
    df[text_field] = df[text_field].str.replace(r"five", "")
    return df



import re
def clean_text(text):
    text = text.lower()
    text = re.sub(r"what’s", "what is ", text)
    text = re.sub(r"\’s", " ", text)
    text = re.sub(r"\’ve", " have ", text)
```

```
    text = re.sub(r"can’t", " can not ", text)
    text = re.sub(r"n’t", " not ", text)
    text = re.sub(r"i’m", "i am ", text)
    text = re.sub(r"\’re", " are ", text)
    text = re.sub(r"\’d", " would ", text)
    text = re.sub(r"\’ll", " will ", text)
    text = re.sub(r"\’scuse", " excuse ", text)
    text = re.sub('\W', ' ' , text)
    text = re.sub('\s+', ' ', text)
    text = text.strip(' ')
    return text


import nltk
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.tokenize import ToktokTokenizer
lemma=WordNetLemmatizer()
token=ToktokTokenizer()


def lemitizeWords(text):
    words=token.tokenize(text)
    listLemma=[]
    for w in words:
        x=lemma.lemmatize(w,'v')
        listLemma.append(x)
    return text


import unicodedata
def removeAscendingChar(data):
    data=unicodedata.normalize('NFKD', data).encode('ascii',
'ignore').decode('utf-8', 'ignore')
    return data
```