```
#sentiment analysis with Text Blob

from textblob import TextBlob
from textblob.en.sentiments import NaiveBayesAnalyzer

def sentiment_analysis(dict_of_comments):
    sentiment_dict = {}
    for category in dict_of_comments:
        if len(dict_of_comments[category]) > 10 :
            if category not in sentiment_dict:
                sentiment_dict[category] = {}
                sentiment_dict[category]['positive'] = []
                sentiment_dict[category]['negative'] = []
                sentiment_dict[category]['neutral'] = []
            for comment in dict_of_comments[category]:
                blob = TextBlob(comment)
                sent = TextBlob(comment, analyzer=NaiveBayesAnalyzer())
                polarity = blob.sentiment.polarity
                if polarity > 0:
                    sentiment_dict[category]['positive'].append(comment)
                if polarity < 0:
                    sentiment_dict[category]['negative'].append(comment)
                if polarity == 0:
                    sentiment_dict[category]['neutral'].append(comment)
        else:
            if category not in sentiment_dict:
                sentiment_dict[category] = ['not enough comments to process']
    return sentiment_dict


#TFIDF

from math import log
import nltk
from nltk import word_tokenize
from nltk.util import ngrams
from nltk.corpus import stopwords
import string
import spacy, re
from textblob import TextBlob
from textblob.en.sentiments import NaiveBayesAnalyzer
import warnings
warnings.filterwarnings('ignore', category=DeprecationWarning)

nlp = spacy.load('en')

def generate_n_grams(n, doc):
    token = nltk.word_tokenize(doc)
    return (list(ngrams(token, n)))

def remove_punctuations_stop_words(doc):
    exclude_punct = set(string.punctuation)
    s = ''.join(ch for ch in doc if ch not in exclude_punct)
    return s

def calculate_term_frequency(doc, n):
    list_of_terms = generate_n_grams(n, doc)
    dict_of_terms_tf = {}
    for term in list_of_terms:
        if term not in dict_of_terms_tf:
            dict_of_terms_tf[term] = {}
```

```
            dict_of_terms_tf[term]['term_count'] = 0
            dict_of_terms_tf[term]['term_frequency'] = 0
        dict_of_terms_tf[term]['term_count'] += 1
    for term in dict_of_terms_tf:
        dict_of_terms_tf[term]['term_frequency'] =
dict_of_terms_tf[term]['term_count'] / len(list_of_terms)
    return dict_of_terms_tf

def calculate_inverse_document_frequency(doc, list_of_comments, n):
    list_of_terms_doc = generate_n_grams(n, doc)
    dict_of_terms_idf = {}
    for term in list_of_terms_doc:
        if term not in dict_of_terms_idf:
            dict_of_terms_idf[term] = {}
            dict_of_terms_idf[term]['num_of_doc_with_term'] = 0
            dict_of_terms_idf[term]['idf'] = 0
    for term in list_of_terms_doc:
        for comment in list_of_comments:
            list_of_terms_comment = generate_n_grams(n, comment)
            if term in list_of_terms_comment:
                dict_of_terms_idf[term]['num_of_doc_with_term'] += 1
    for term in list_of_terms_doc:
        dict_of_terms_idf[term]['idf'] = log(len(list_of_comments) /
dict_of_terms_idf[term]['num_of_doc_with_term'])
    return dict_of_terms_idf

def calculate_tf_idf(dict_tf, dict_idf):
    dict_tf_idf = {}
    for term in dict_tf:
        if term not in dict_tf_idf:
            dict_tf_idf[term] = dict_tf[term]['term_frequency'] *
dict_idf[term]['idf']
    return dict_tf_idf

#Spacy - Identify universities and organizations

import spacy
def identify_org(doc):
    nlp = spacy.load('en_core_web_sm')
    for ent in doc.ents:
        if ent.label_ == 'ORG':
            print(ent.text, ent.label_)


#Spacy - Identify activities

def pos_entity_tagger_spacy(senetnce_chunk):
    #pattern = re.compile(
    #
r'((<JJ>|<JJR>|<JJS>)+(<NN>|<NNS>|<NNP>)+(<RB>|<RBR>|<RBS>)*(<VB>|<VBD>|<VBG>|<VBN>|<V
BP>>*))')
    #pattern = re.compile(r'((<VB>|<VBD>|<VBG>|<VBN>|<VBP>)+(<NN>|<NNS>|<NNP>)*)')
    pattern =
re.compile(r'(<NN>|<NNS>|<NNP>)+(<VB>|<VBD>|<VBG>|<VBN>|<VBP>)*(<JJ>|<JJR>|<JJS>)*(<RB
>|<RBR>|<RBS>)*')
    doc  = nlp(senetnce_chunk)
    signature = ''.join(['<%s>' % w.tag_ for w in doc])
    #print('signature','---->',signature,doc)
    if pattern.match(signature) is not None:
        yield doc
```

```
#Jaccard similarity

def compute_jaccard_similarity(list_of_activities, sentence_chunk):
    similarity_measure = 0.0
    string_a = str(sentence_chunk).replace('[', '').replace(']', '').split(' ')
    for activity in list_of_activities:
        string_b = str(activity).replace('[', '').replace(']', '').split(' ')
        similar_words = set()
        for word in string_a:
            if word in string_b:
                similar_words.add(word)
        # union of words in tweet_a and tweet_b
        union_of_words = list(set().union(string_a, string_b))
        # compute jaccard similarity
        js = float(len(similar_words) / len(union_of_words))
        similarity_measure = max(similarity_measure, js)
    return similarity_measure


#LDA - Topic Modelling

from gensim import corpora
from gensim.models import LdaModel
import warnings

warnings.filterwarnings('ignore',category=DeprecationWarning)
from nltk import RegexpTokenizer, PorterStemmer
from nltk.corpus import stopwords
import gensim

def tokenization(doc):
    tokenizer = RegexpTokenizer(r'\w+')
    raw = doc.lower()
    tokens = tokenizer.tokenize(raw)
    return tokens

def remove_stop_words(list_of_tokens):
    tokens_without_stop_words = []
    stop_words = set(stopwords.words('english'))
    for token in list_of_tokens:
        if token not in stop_words:
            tokens_without_stop_words.append(token)
    return tokens_without_stop_words

def stemming_words(tokens_without_stop_words):
    p_stemmer = PorterStemmer()
    list_of_stemmed_tokens = [p_stemmer.stem(i) for i in tokens_without_stop_words]
    return list_of_stemmed_tokens

def construct_document_term_matrix(list_of_tokenized_tweets):
    dictionary = corpora.Dictionary(list_of_tokenized_tweets)
    dictionary.save('dictionary.dict')

    document_term_matrix = [dictionary.doc2bow(text) for text in
list_of_tokenized_tweets]
    corpora.MmCorpus.serialize('corpus.mm', document_term_matrix)

    return document_term_matrix,dictionary

def lda(list_of_comments):
    tweet_count = 0
```

```
    list_of_tokenized_comments = []
    for comment in list_of_comments:
        list_of_tokens = tokenization(comment)
        tokens_without_stop_words = remove_stop_words(list_of_tokens)
        list_of_stemmed_tokens = stemming_words(tokens_without_stop_words)
        list_of_tokenized_comments.append(list_of_stemmed_tokens)
    matrix_dict = construct_document_term_matrix(list_of_tokenized_comments)
    #print('done constructing matrix')
    document_term_matrix = matrix_dict[0]
    dictionary = matrix_dict[1]
    #print('started calculating lda')
    #generate lda model
    Lda = gensim.models.ldamodel.LdaModel
    ldamodel = Lda(document_term_matrix, num_topics=5, id2word=dictionary, passes=20)
    ldamodel.save('topic.model')
    loading = LdaModel.load('topic.model')
    #print('done calculating lda')
    return (ldamodel)

def display_topics(model, feature_names, no_top_words):
    for topic_idx, topic in enumerate(model.components_):
        print ("Topic %d:" % (topic_idx))
        print (" ".join([feature_names[i] for i in topic.argsort()[:-no_top_words -
1:-1]]))

#Summarizer - Gensim

from gensim.summarization import summarize,keywords
from nltk.stem import PorterStemmer

def root_of_word(k,list_of_keys):
    ps = PorterStemmer()
    word_with_the_same_base = ''
    for key in list_of_keys:
        if ps.stem(k) == ps.stem(key):
            word_with_the_same_base = key
            return word_with_the_same_base
    return word_with_the_same_base

def summarization_of_comments(dict_of_comments):
    summarization = {}
    for category in dict_of_comments:
        if len(dict_of_comments[category]) > 50:
            summary_of_comments = ''
            if category not in summarization:
                summarization[category] = {}
                summarization[category]['keywords'] = {}
                summarization[category]['overall_summary'] = ''
                summarization[category]['input_text'] = []
            for comment in dict_of_comments[category]:
                    summary_of_comments += comment
                    try:
                        key = keywords(comment, split=True)
                        #print(key)
                        if len(key) > 0:
                            for k in key:
                                if k not in summarization[category]['keywords']:
                                    summarization[category]['keywords'][k]= 0
                                summarization[category]['keywords'][k] += 1

                    except:
```

```
                    continue
            summarization[category]['overall_summary'] =
summarize(summary_of_comments, ratio=0.05)
    return summarization

#Identify Suggestions - Multinomial Naive Bayes

import pandas as pd
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.feature_extraction.text import TfidfTransformer
from sklearn.naive_bayes import MultinomialNB
from sklearn.utils import shuffle

def analysze_comments(dict_of_commnets):
    #load training data
    df = pd.read_csv('/Users/User/yelp/training.txt', delimiter='\t',
names=['comment', 'type'])
    df['type_id'] = df['type'].factorize()[0]
    df = shuffle(df).reset_index(drop=True)
    type_id_df = df[['type', 'type_id']].drop_duplicates().sort_values('type_id')
    training_data = df
    print('Number of observations in the training data:', len(training_data))
    X_train = training_data['comment']
    Y_train = training_data['type']
    count_vect = CountVectorizer()
    X_train_counts = count_vect.fit_transform(X_train.values.astype('U'))
    tfidf_transformer = TfidfTransformer()
    X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
    clf = MultinomialNB().fit(X_train_tfidf, Y_train)

    list_of_analyzed_commnets = {}
    for category in dict_of_commnets:
        if category not in list_of_analyzed_commnets:
            list_of_analyzed_commnets[category] = {}
            list_of_analyzed_commnets[category]['suggestions'] = []
            list_of_analyzed_commnets[category]['good_practices'] = []
        for comment in dict_of_commnets[category]:
            predicted_y = clf.predict(count_vect.transform([comment]))
            if predicted_y == 'low':
                list_of_analyzed_commnets[category]['suggestions'].append(comment)
            else:
                list_of_analyzed_commnets[category]['good_practices'].append(comment)
    return list_of_analyzed_commnets
```