Timothy Mutua 18M01ABT018

```
import numpy as np
import pandas as pd
pd.options.display.max_colwidth = 100
train_data = pd.read_csv("../input/train.csv", encoding='ISO-8859-1')
rand_indexs = np.random.randint(1,len(train_data),50).tolist()
train_data["SentimentText"][rand_indexs]
import re
tweets_text = train_data.SentimentText.str.cat()
emos = set(re.findall(r" ([xX:;][-']?.) ",tweets_text))
emos_count = []
for emo in emos:
emos_count.append((tweets_text.count(emo), emo))
sorted(emos_count,reverse=True)
HAPPY_EMO = r" ([xX;:]-?[dD)]|:-?[\)]|[;:][pP]) "
SAD_EMO = r" (:'?[/|\(]) "
print("Happy emoticons:", set(re.findall(HAPPY_EMO, tweets_text)))
print("Sad emoticons:", set(re.findall(SAD_EMO, tweets_text)))
import nltk
from nltk.tokenize import word_tokenize

def most_used_words(text):
tokens = word_tokenize(text)
frequency_dist = nltk.FreqDist(tokens)
print("There is %d different words" % len(set(tokens)))
return sorted(frequency_dist,key=frequency_dist.__getitem__, reverse=True)

from nltk.corpus import stopwords
mw = most_used_words(train_data.SentimentText.str.cat())
most_words = []
for w in mw:
if len(most_words) == 1000:
break
if w in stopwords.words("english"):
continue
else:
most_words.append(w)
sorted(most_words)
from nltk.stem.snowball import SnowballStemmer
from nltk.stem import WordNetLemmatizer
def stem_tokenize(text):
stemmer = SnowballStemmer("english")
stemmer = WordNetLemmatizer()
return [stemmer.lemmatize(token) for token in word_tokenize(text)]
def lemmatize_tokenize(text):
lemmatizer = WordNetLemmatizer()
```

```
return [lemmatizer.lemmatize(token) for token in word_tokenize(text)]


from sklearn.base import TransformerMixin, BaseEstimator
from sklearn.pipeline import Pipeline

# We need to do some preprocessing of the tweets.
# We will delete useless strings (like @, # ...)
# because we think that they will not help
# in determining if the person is Happy/Sad
class  TextPreProc(BaseEstimator,TransformerMixin):
def __init__(self, use_mention=False):
self.use_mention = use_mention

def fit(self, X, y=None):
return self

def transform(self, X, y=None):
# We can choose between keeping the mentions
# or deleting them
if self.use_mention:
X = X.str.replace(r"@[a-zA-Z0-9_]* ", " @tags ")
else:
X = X.str.replace(r"@[a-zA-Z0-9_]* ", "")


# Keeping only the word after the #
X = X.str.replace("#", "")
X = X.str.replace(r"[-\.\n]", "")
# Removing HTML garbage
X = X.str.replace(r"&\w+;", "")
# Removing links
X = X.str.replace(r"https?://\S*", "")
# replace repeated letters with only two occurences
# heeeelllloooo => heelloo
X = X.str.replace(r"(.)\1+", r"\1\1")
# mark emoticons as happy or sad
X = X.str.replace(HAPPY_EMO, " happyemoticons ")
X = X.str.replace(SAD_EMO, " sademoticons ")
X = X.str.lower()
return X
# In[ ]:
# This is the pipeline that will transform our tweets to something eatable.
# You can see that we are using our previously defined stemmer, it will
# take care of the stemming process.
# For stop words, we let the inverse document frequency do the job


from sklearn.model_selection import train_test_split
sentiments = train_data['Sentiment']
```

```
tweets = train_data['SentimentText']
# I get those parameters from the 'Fine tune the model' part
vectorizer = TfidfVectorizer(tokenizer=lemmatize_tokenize, ngram_range=(1,2))
pipeline = Pipeline([
('text_pre_processing', TextPreProc(use_mention=True)),
('vectorizer', vectorizer),
])
# Let's split our data into learning set and testing set
# This process is done to test the efficency of our model at the end.
# You shouldn't look at the test data only after choosing the final model
learn_data, test_data, sentiments_learning, sentiments_test =
train_test_split(tweets, sentiments,
test_size=0.3)
# This will tranform our learning data from simple text to vector
# by going through the preprocessing tranformer.
learning_data = pipeline.fit_transform(learn_data)


# # Select a model
# When we have our data ready to be processed by ML models, the question we
should ask is which model
to use?
#
# The answer varies depending on the problem and data, for example, it's known
that Naive Bias has
proven good efficacy against Text Based Problems.
#
# A good way to choose a model is to try different candidate, evaluate them
using cross validation, then
chose the best one which will be later tested against our test data.
# In[ ]:
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.naive_bayes import BernoulliNB, MultinomialNB
lr = LogisticRegression()
bnb = BernoulliNB()
mnb = MultinomialNB()

models = {
'logitic regression': lr,
'bernoulliNB': bnb,
'multinomialNB': mnb,
}
for model in models.keys():
scores = cross_val_score(models[model], learning_data, sentiments_learning,
scoring="f1", cv=10)
print("===", model, "===")
print("scores = ", scores)
print("mean = ", scores.mean())
```

```
print("variance = ", scores.var())


models[model].fit(learning_data, sentiments_learning)
print("score on the learning data (accuracy) = ",
accuracy_score(models[model].predict(learning_data),
sentiments_learning))
print("")
# None of those models is likely to be overfitting, I will choose the
multinomialNB.
# # Fine tune the model
# I'm going to use the GridSearchCV to choose the best parameters to use.
#
# What the GridSearchCV does is trying different set of parameters, and for
each one, it runs a cross
validation and estimate the score. At the end we can see what are the best
parameter and use them to
build a better classifier.
# In[ ]:
from sklearn.model_selection import GridSearchCV
grid_search_pipeline = Pipeline([
('text_pre_processing', TextPreProc()),
('vectorizer', TfidfVectorizer()),
('model', MultinomialNB()),
])

params = [
{
'text_pre_processing__use_mention': [True, False],
'vectorizer__max_features': [1000, 2000, 5000, 10000, 20000, None],
'vectorizer__ngram_range': [(1,1), (1,2)],
},
]
grid_search = GridSearchCV(grid_search_pipeline, params, cv=5, scoring='f1')
grid_search.fit(learn_data, sentiments_learning)
print(grid_search.best_params_)
# # Test
# Testing our model against data other than the data used for training our
model will show how well the
model is generalising on new data.
#
# ### Note
# We shouldn't test to choose the model, this will only let us confirm that the
choosen model is doing well.
mnb.fit(learning_data, sentiments_learning)
testing_data = pipeline.transform(test_data)
mnb.score(testing_data, sentiments_test)
# Predecting on the test.csv
sub_data = pd.read_csv("../input/test.csv", encoding='ISO-8859-1')
sub_learning = pipeline.transform(sub_data.SentimentText)
```

```
sub = pd.DataFrame(sub_data.ItemID, columns=("ItemID", "Sentiment"))
sub["Sentiment"] = mnb.predict(sub_learning)
print(sub)

model = MultinomialNB()
model.fit(learning_data, sentiments_learning)
tweet = pd.Series([input(),])
tweet = pipeline.transform(tweet)
proba = model.predict_proba(tweet)[0]
print("The probability that this tweet is sad is:", proba[0])
print("The probability that this tweet is happy is:", proba[1])
```