HW4

February 20, 2022

DSCC 465

Assignment 4

Uzair Tahamid Siam

0.1 Importing Packages

[1]:
```
import nltk
import pandas as pd
import numpy as np
!pip install emoji
```

```
Requirement already satisfied: emoji in
/Users/usiam/opt/anaconda3/lib/python3.9/site-packages (1.6.3)
```

0.2 Downloading additional NLTK packages

[2]:
```
nltk.download('punkt')
nltk.download('averaged_perceptron_tagger')
nltk.download('wordnet')
```

```
[nltk_data] Downloading package punkt to /Users/usiam/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package averaged_perceptron_tagger to
[nltk_data]     /Users/usiam/nltk_data...
[nltk_data]   Package averaged_perceptron_tagger is already up-to-
[nltk_data]       date!
[nltk_data] Downloading package wordnet to /Users/usiam/nltk_data...
[nltk_data]   Package wordnet is already up-to-date!
```

[2]:
```
True
```

Q1

0.3 a) Reading in data

[3]:
```
## Reading in data

df = pd.read_csv('corona_fake.csv')
```

0.4 b) i) Tokenizing 'text' attribute

[4]:
```
# tokenizing the sentences into separate words
tokenized_sentences = df["text"].fillna("").apply(nltk.word_tokenize)
```

0.5 ii) POS Tagging

[5]:
```
# adding the parts of speech that each word is associated with using pos_tag
pos_tag_tokenized_sentences = tokenized_sentences.apply(nltk.pos_tag)
```

0.6 iii) Lemmatizing

[6]:
```
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet

wnl = WordNetLemmatizer()

# nltk to wordnet tag
def pos_tagger(nltk_tag):
    if nltk_tag.startswith('J'):
        return wordnet.ADJ
    elif nltk_tag.startswith('V'):
        return wordnet.VERB
    elif nltk_tag.startswith('N'):
        return wordnet.NOUN
    elif nltk_tag.startswith('R'):
        return wordnet.ADV
    else:
        return None

# nltk to wordnet tagged words
wn_tagged = [list(map(lambda x: (x[0], pos_tagger(x[1])), x)) for x in␣
 ↪pos_tag_tokenized_sentences]

# lemmatized words using wordnetlemmatizer
lemmatized_sentences = [[word if not tag else wnl.lemmatize(word, tag) for word,␣
 ↪tag in row] for row in wn_tagged]
```

0.7 iv) Filtering Stopwords

[7]:
```
from nltk.corpus import stopwords

stop_words = set(stopwords.words('english'))

# filtered sentences after removing stop words
filtered_sentences = [[w for w in sentence if w not in stop_words] for sentence␣
 ↪in lemmatized_sentences]
```

0.8 v) Removing numbers, words that are shorter than 2, characters, punctuation, links and emojis.

[8]:
```
### Remove non-words

import re
import emoji

# regex expressions to detect valid word
def valid_word(string: str)->bool:
    return False if re.match(r'[^\w\s]', string) or len(string) < 2 or re.
 ↪match(r'[0-9]+', string) or\
                            bool(re.search('html', string)) or\
                            re.match(r"http\S+", string) or re.match(emoji.
 ↪get_emoji_regexp(), string) else True

final_step = [[w for w in sentence if valid_word(w)] for sentence in␣
 ↪filtered_sentences]
```

[9]:
```
# cleaning and inserting into df

cleaned = list(map(lambda x: " ".join(x), final_step))
df["text_clean"] = cleaned
```

Q2

0.9 a)

N-gram is a probabilistic language model used to predict the next item in a sequence whether it be a sentence, DNA, or even a password. If you think of a sentence being covered by a sliding window, then, the *'n'* in n-gram represents the length of that window. For example, if we look at a sentence and look at each word as an item, then using a 1-gram (or unigram) the sentence will be divided into a set containing every word.

Sentence: "This is a sentence"

unigram: {{This}, {is}, {a}, {sentence}}

Similarly for a bigram or 2-gram,

bi-gram: {{This, is}, {is, a}, {a, sentence}}

And so on.

One of the key aspects of n-grams is that it is a Markov model in the sense that there is only dependence on the last n-1 words.

N-grams are useful in NLP to make sense of words in context so that the meaning of a world can be better understood by comparing the presence of the said word in the presence of words before and after it.

0.10 b) Importing

[10]:
```
from sklearn.feature_extraction.text import CountVectorizer, TfidfVectorizer
```

0.11 c) i, ii, iii)

[11]:
```
# CountVectorizer

# ngram_range=(1,1)
countVec1 = CountVectorizer(lowercase=True, ngram_range=(1,1))
text_clean_cv1 = countVec1.fit_transform(df.text_clean)


# ngram_range=(1,2)
countVec2 = CountVectorizer(lowercase=True, ngram_range=(1,2))
text_clean_cv2 = countVec2.fit_transform(df.text_clean)


# ngram_range=(1,3)
countVec3 = CountVectorizer(lowercase=True, ngram_range=(1,3))
text_clean_cv3 = countVec3.fit_transform(df.text_clean)
```

0.12 d) i, ii, iii)

[12]:
```
# TfidfVectorizer

# ngram_range=(1,1)
tfidVec1 = TfidfVectorizer(lowercase=True, ngram_range=(1,1))
text_clean_tf1 = tfidVec1.fit_transform(df.text_clean)


# ngram_range=(1,2)
tfidVec2 = TfidfVectorizer(lowercase=True, ngram_range=(1,2))
text_clean_tf2 = tfidVec2.fit_transform(df.text_clean)
```

```
# ngram_range=(1,3)
tfidVec3 = TfidfVectorizer(lowercase=True, ngram_range=(1,3))
text_clean_tf3 = tfidVec3.fit_transform(df.text_clean)
```

Q3

[13]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import r2_score
from sklearn.linear_model import LogisticRegressionCV
```

0.13 a)

[14]:
```
# finding accuracy in the countvectorized vectors

accuracy_dict_cv = {}

# model for fitting
lrcv = LogisticRegressionCV(cv=5, random_state=265, max_iter=1000, n_jobs=-1)

for i in range(3):
    n_gram_range = f"CV ngram_range=(1,{i+1})"
    eval_statement = f"train_test_split(text_clean_cv{i+1}, df.label,␣
 ↪test_size=0.3, random_state=265)"

    # splitting into train test
    x_train, x_test, y_train, y_test = eval(eval_statement)

    # fitting data using the model
    lrcv.fit(x_train, y_train)

    # finding accuracy
    accuracy = lrcv.score(x_test, y_test)

    # inserting accuracy and model(as key) into dictionary
    accuracy_dict_cv[n_gram_range] = accuracy
```

[15]:
```
accuracy_dict_cv
```

[15]:
```
{'CV ngram_range=(1,1)': 0.9051724137931034,
 'CV ngram_range=(1,2)': 0.9080459770114943,
 'CV ngram_range=(1,3)': 0.9166666666666666}
```

0.14 b)

[16]:
```
accuracy_dict_tfid = {}

for i in range(3):
    n_gram_range = f"TFID ngram_range=(1,{i+1})"
    eval_statement = f"train_test_split(text_clean_tf{i+1}, df.label,␣
 ↪test_size=0.3, random_state=265)"

    # splitting into train test
    x_train, x_test, y_train, y_test = eval(eval_statement)

    # fitting data using the model
    lrcv.fit(x_train, y_train)

    # finding accuracy
    accuracy = lrcv.score(x_test, y_test)

    # inserting accuracy and model(as key) into dictionary
    accuracy_dict_tfid[n_gram_range] = accuracy
```

[17]:
```
accuracy_dict_tfid
```

[17]:
```
{'TFID ngram_range=(1,1)': 0.9166666666666666,
 'TFID ngram_range=(1,2)': 0.8936781609195402,
 'TFID ngram_range=(1,3)': 0.896551724137931}
```

0.15 c)

[18]:
```
# creating table to display all the accuracies

accuracy_dict = {**accuracy_dict_cv, **accuracy_dict_tfid}
accuracy_table = pd.DataFrame.from_dict(accuracy_dict, orient='index',␣
 ↪columns=['Accuracy'])
accuracy_table
```

[18]:

	Accuracy
CV ngram_range=(1,1)	0.905172
CV ngram_range=(1,2)	0.908046
CV ngram_range=(1,3)	0.916667
TFID ngram_range=(1,1)	0.916667
TFID ngram_range=(1,2)	0.893678
TFID ngram_range=(1,3)	0.896552

Q4

0.16 Comparing the different solvers in sklearn

0.16.1 a) Newton-CG

Newton's method is an improved gradient descent algorithm as it uses a "*better*" quadratic function minimisation as it uses the quadratic approximation (i.e. first AND second partial derivatives with the Hessian (the Hessian is a square matrix of second-order partial derivatives of order $n \times n$). However, it's computationally expensive because of the Hessian Matrix (i.e. second partial derivatives calculations). And it also attracts to Saddle Points which are common in multivariable optimization.

0.16.2 b) LBFGS

This is analogous to Newton's Method but here the Hessian matrix is approximated using updates specified by gradient evaluations. In other words, it uses an estimation to the inverse Hessian matrix. The term "L" which represents "Limited-memory" mean it stores only a few vectors that represent the approximation implicitly. When the dataset is small, LBFGS relatively performs the best compared to other methods especially it saves a lot of memory, however there are some important drawbacks such that if it is unsafeguarded, it may not converge to anything.

0.16.3 c) Liblinear

The solver uses a Coordinate Descent (CD) algorithm that solves optimization problems by successively performing approximate minimization along coordinate directions or coordinate hyperplanes. It applies automatic parameter selection i.e. L1 Regularization and is usually a good solver for high dimension datasets and large-scale classification problems. However, one of the drawbacks of Liblinear is that it may get stuck at a non-stationary point if the level curves of a function are not smooth.

0.16.4 d) SAG

SAG **(Stochastic Average Gradient)** method optimizes the sum of a finite number of smooth convex functions. The SAG method's iteration cost is independent of the number of terms in the sum just like other stochastic gradient (SG) methods. However, since it incorporates a memory of previous gradient values the SAG method achieves a faster convergence rate than other SG methods. It is faster than other solvers for large datasets, when both the number of samples and the number of features are large. Although it is faster, it does have a O(N) memory complexity so for large datasets it may get impractical.

0.16.5 e) SAGA

The SAGA solver is a variant of SAG that also supports L1 Regularization instead of just the smooth L2 Regularization like SAG. This makes SAGA a popular choice for sparse multinomial logistic regression and very large dataset.