lab10b

October 31, 2019

# Classification with word2vec

– Prof. Dorien Herremans

We will be tackling a classification problem by first creating word embeddings, and comparing this to alternative approaches.

During this tutorial, you will need some of the following libraries, let’s install them first if you don’t have them:

In [0]: !pip install bs4

!pip install sklearn

!pip install nltk

!pip install gensim

!pip install lxml

Requirement already satisfied: bs4 in /usr/local/lib/python3.6/dist-packages (0.0.1) Requirement already satisfied: beautifulsoup4 in /usr/local/lib/python3.6/dist-packages (from Requirement already satisfied: sklearn in /usr/local/lib/python3.6/dist-packages (0.0) Requirement already satisfied: scikit-learn in /usr/local/lib/python3.6/dist-packages (from sk Requirement already satisfied: scipy>=0.17.0 in /usr/local/lib/python3.6/dist-packages (from Requirement already satisfied: numpy>=1.11.0 in /usr/local/lib/python3.6/dist-packages (from Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.6/dist-packages (from Requirement already satisfied: nltk in /usr/local/lib/python3.6/dist-packages (3.2.5) Requirement already satisfied: six in /usr/local/lib/python3.6/dist-packages (from nltk) (1.12 Requirement already satisfied: gensim in /usr/local/lib/python3.6/dist-packages (3.6.0) Requirement already satisfied: scipy>=0.18.1 in /usr/local/lib/python3.6/dist-packages (from Requirement already satisfied: six>=1.5.0 in /usr/local/lib/python3.6/dist-packages (from gens Requirement already satisfied: smart-open>=1.2.1 in /usr/local/lib/python3.6/dist-packages (fr Requirement already satisfied: numpy>=1.11.3 in /usr/local/lib/python3.6/dist-packages (from Requirement already satisfied: boto>=2.32 in /usr/local/lib/python3.6/dist-packages (from Requirement already satisfied: boto3 in /usr/local/lib/python3.6/dist-packages (from smart-ope Requirement already satisfied: requests in /usr/local/lib/python3.6/dist-packages (from smart- Requirement already satisfied: s3transfer<0.3.0,>=0.2.0 in /usr/local/lib/python3.6/dist-packa Requirement already satisfied: jmespath<1.0.0,>=0.7.1 in /usr/local/lib/python3.6/dist-package Requirement already satisfied: botocore<1.14.0,>=1.13.2 in /usr/local/lib/python3.6/dist-packa Requirement already satisfied: chardet<3.1.0,>=3.0.2 in /usr/local/lib/python3.6/dist-packages Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.6/dist-packages (f Requirement already satisfied: urllib3<1.25,>=1.21.1 in /usr/local/lib/python3.6/dist-packages Requirement already satisfied: idna<2.9,>=2.5 in /usr/local/lib/python3.6/dist-packages (from

Requirement already satisfied: docutils<0.16,>=0.10 in /usr/local/lib/python3.6/dist-packages Requirement already satisfied: python-dateutil<3.0.0,>=2.1; python\_version >= "2.7" in Requirement already satisfied: lxml in /usr/local/lib/python3.6/dist-packages (4.2.6)

Now we can import some libraries that we will use:

In [0]: **import logging**

## import pandas as pd import numpy as np

**from numpy import** random

## import gensim import nltk import lxml

**from sklearn.model\_selection import** train\_test\_split

**from sklearn.feature\_extraction.text import** CountVectorizer, TfidfVectorizer

**from sklearn.metrics import** accuracy\_score, confusion\_matrix, classification\_report

## import matplotlib.pyplot as plt matplotlib inline

* 1. **TFIDF with logistic regression**
     1. **Preparing the dataset**

The classification problem at hand is to predict the tag that belongs to a stack overflow post. The data from Google BigQuery is publicly available at this Cloud Storage URL:

https://storage.googleapis.com/tensorflow-workshop-examples/stack-overflow-data.csv. We can read it directly into a pandas dataframe.

In [0]: url = "https://storage.googleapis.com/tensorflow-workshop-examples/stack-overflow-data df = pd.read\_csv(url, encoding = 'latin-1')

Let’s start by having a look at our data:

In [0]: *# only keep data that has a tag (is labeled):*

df = df[pd.notnull(df['tags'])]

*# display first ten rows:*

df.head(10)

Out[0]: post tags

1. what is causing this behavior in our c# datet... c#
2. have dynamic html load as if it was in an ifra... asp.net
3. how to convert a float value in to min:sec i ... objective-c
4. .net framework 4 redistributable just wonderi. net
5. trying to calculate and print the mean and its... python
6. how to give alias name for my website i have ... asp.net
7. window.open() returns null in angularjs it wo... angularjs
8. identifying server timeout quickly in iphone ... iphone
9. unknown method key error in rails 2.3.8 unit ... ruby-on-rails
10. from the include how to show and hide the con... angularjs

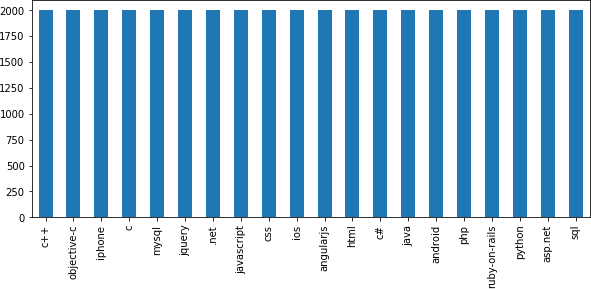
The size of our model will depend on how many unqiue words are in the dataset (meaning in the article text or posts):

In [0]: *# Count the number of words:*

df['post'].apply(**lambda** x: len(x.split(' '))).sum() Out[0]: 10286120

We have over 10 million words in the data. That’s a lot! Let’s visualise our dataset:

In [0]: *# visualising dataset* plt.figure(figsize=(10,4)) df.tags.value\_counts().plot(kind='bar');



As you can see, the classes are very well balanced.

Now let’s have a look at the data of the posts (‘post’ columns) in more detail:

In [0]: print(df['post'].values[10])

when we need interface c# <blockquote> <strong>possible duplicate:</strong><br> <a href=

As you can see, the text needs to be cleaned up a bit. Below we use the nltk toolkit to remove spaces, html tags, stopwords, symbols etc. Below we define a function to remove stop words, replace / and other symbols with spaces, . . .

In [0]: *# note: slower students may wish to skip this step to finish the lab in class*

**from nltk.corpus import** stopwords

## import re

**from bs4 import** BeautifulSoup

*# load a list of stop words*

nltk.download('stopwords')

REPLACE\_BY\_SPACE\_RE = re.compile('[/()**{}**\[\]\|@,;]') BAD\_SYMBOLS\_RE = re.compile('[^0-9a-z #+\_]') STOPWORDS = set(stopwords.words('english'))

**def** clean\_text(text):

*"""*

*"""*

*text: a string*

*return: modified initial string*

text = BeautifulSoup(text, 'html.parser').text *# HTML decoding*

text = text.lower() *# lowercase text*

text = REPLACE\_BY\_SPACE\_RE.sub(' ', text) *# replace REPLACE\_BY\_SPACE\_RE symbols by* text = BAD\_SYMBOLS\_RE.sub('', text) *# delete symbols which are in BAD\_SYMBOLS\_RE f* text = ' '.join(word **for** word **in** text.split() **if** word **not in** STOPWORDS) *# delete s* **return** text

[nltk\_data] Downloading package stopwords to /root/nltk\_data... [nltk\_data] Unzipping corpora/stopwords.zip.

Now we can apply the newly defined function on the column of df ‘post’.

In [0]: df['post'] = df['post'].apply(clean\_text)

Let’s check the results:

In [0]: print(df['post'].values[10])

need interface c# possible duplicate would want use interfaces need interface want know use ex

This looks a lot better!

Now how many unique words do we have in this cleaned up dataset?

In [0]: df['post'].apply(**lambda** x: len(x.split(' '))).sum() Out[0]: 3424194

Now we have over 3 million words to work with.

Before we start creating some classifiers, let’s split our dataset in a test set (for evaluation) and training set:

In [0]: X = df.post

y = df.tags

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state

## Logistic regression

Now that we have our features, we can train a classifier to try to predict the tag of a post. We will start with logistic regression and TFIDF representation which provides a nice baseline for this task.

To make the vectorizer => transformer => classifier easier to work with, we will use Pipeline class in Scikit-Learn that behaves like a compound classifier.

In [0]: **from sklearn.linear\_model import** LogisticRegression

**from sklearn.pipeline import** Pipeline

**from sklearn.feature\_extraction.text import** TfidfTransformer

*# we define a Pipeline, which first represents our features as TFID # Then performs logistic regression*

logreg = Pipeline([('vect', CountVectorizer()),

('tfidf', TfidfTransformer()),

('clf', LogisticRegression(n\_jobs=1, C=1e5)),

])

logreg.fit(X\_train, y\_train)

/usr/local/lib/python3.6/dist-packages/sklearn/linear\_model/logistic.py:432: FutureWarning: De FutureWarning)

/usr/local/lib/python3.6/dist-packages/sklearn/linear\_model/logistic.py:469: FutureWarning: De "this warning.", FutureWarning)

Out[0]: Pipeline(memory=None,

steps=[('vect',

CountVectorizer(analyzer='word', binary=False,

decode\_error='strict',

dtype=<class 'numpy.int64'>, encoding='utf-8', input='content', lowercase=True, max\_df=1.0, max\_features=None, min\_df=1,

ngram\_range=(1, 1), preprocessor=None, stop\_words=None, strip\_accents=None, token\_pattern='(?u)\\b\\w\\w+\\b', tokenizer=None, vocabulary=None)),

('tfidf',

TfidfTransformer(norm='l2', smooth\_idf=True,

sublinear\_tf=False, use\_idf=True)),

('clf',

LogisticRegression(C=100000.0, class\_weight=None, dual=False,

fit\_intercept=True, intercept\_scaling=1, l1\_ratio=None, max\_iter=100, multi\_class='warn', n\_jobs=1, penalty='l2', random\_state=None, solver='warn', tol=0.0001, verbose=0, warm\_start=False))],

verbose=False)

How well does it work?

In [0]: *# to show the computation time:*

time



y\_pred = logreg.predict(X\_test)

print('accuracy **s**' accuracy\_score(y\_pred, y\_test)) print(classification\_report(y\_test, y\_pred))

accuracy 0.7826666666666666

|  |  |  |  |
| --- | --- | --- | --- |
|  | precision | recall f1-score | support |
| .net | 0.70 | 0.62 0.66 | 613 |
| android | 0.91 | 0.90 0.91 | 620 |
| angularjs | 0.97 | 0.94 0.95 | 587 |
| asp.net | 0.78 | 0.77 0.78 | 586 |
| c | 0.77 | 0.81 0.79 | 599 |
| c# | 0.60 | 0.58 0.59 | 589 |
| c++ | 0.77 | 0.76 0.76 | 594 |
| css | 0.81 | 0.86 0.84 | 610 |
| html | 0.69 | 0.71 0.70 | 617 |
| ios | 0.61 | 0.59 0.60 | 587 |
| iphone | 0.64 | 0.64 0.64 | 611 |
| java | 0.83 | 0.83 0.83 | 594 |
| javascript | 0.78 | 0.78 0.78 | 619 |
| jquery | 0.84 | 0.85 0.84 | 574 |
| mysql | 0.80 | 0.83 0.82 | 584 |
| objective-c | 0.65 | 0.64 0.65 | 578 |
| php | 0.82 | 0.84 0.83 | 591 |
| python | 0.91 | 0.91 0.91 | 608 |
| ruby-on-rails | 0.96 | 0.94 0.95 | 638 |
| sql | 0.78 | 0.83 0.80 | 601 |
| accuracy |  | 0.78 | 12000 |
| macro avg | 0.78 | 0.78 0.78 | 12000 |
| weighted avg | 0.78 | 0.78 0.78 | 12000 |
| CPU times: user Wall time: 1.15 | 1.13 s, sys: s | 1.35 ms, total: | 1.14 s |

That’s quite a good accuracy. Now let’s see if we can combine word2vec with logistic regres- sion by feeding the new embedded representation to our logistic regression instead of the bag of words.

# Word2vec embedding and Logistic Regression

Let’s load a pretrained word2vec model, and use the embedding representation as input to a simple classifier (i.e. logistic regression).

You can use the word2vec model you trained in lab 10a, or load this (quite big, 1.5GB) pre- trained word2vec model: https://s3.amazonaws.com/dl4j-distribution/GoogleNews-vectors- negative300.bin.gz

Note: it can take a while to load. (takes 2min for me)

In [0]: !wget "https://s3.amazonaws.com/dl4j-distribution/GoogleNews-vectors-negative300.bin.g

--2019-10-31 03:30:16-- https://s3.amazonaws.com/dl4j-distribution/GoogleNews-vectors-negativ Resolving s3.amazonaws.com (s3.amazonaws.com)... 52.216.177.197

Connecting to s3.amazonaws.com (s3.amazonaws.com)|52.216.177.197|:443... connected. HTTP request sent, awaiting response... 200 OK

Length: 1647046227 (1.5G) [application/x-gzip] Saving to: GoogleNews-vectors-negative300.bin.gz

GoogleNews-vectors- 100 [===================>] 1.53G 56.3MB/s in 23s

2019-10-31 03:30:40 (67.3 MB/s) - GoogleNews-vectors-negative300.bin.gz saved [1647046227/1647

Once the file is on your system:

In [0]: time



**from gensim.models import** Word2Vec

wv = gensim.models.KeyedVectors.load\_word2vec\_format("GoogleNews-vectors-negative300.b wv.init\_sims(replace=**True**)

print('Model loaded')

/usr/local/lib/python3.6/dist-packages/smart\_open/smart\_open\_lib.py:398: UserWarning: This fun

'See the migration notes for details: s' \_MIGRATION\_NOTES\_URL

Model loaded

CPU times: user 2min 3s, sys: 4.3 s, total: 2min 7s Wall time: 2min 7s

If you are interested how good these pretrained embeddings are, you could try some of the similarity tests we did in Lab 10a.

As we have multiple words for each post, we will need to somehow combine them. A common way to achieve this is by averaging the word vectors per document. It could also be summation or weighted addition. The function below takes as input a list of words and the w2v model wv. Then it retrieves the vector embeddings for each of the words and averages them.

In [0]: **def** word\_averaging(wv, words):

*# averages a set of words 'words' given their wordvectors 'wv'*

all\_words, mean = set(), []

*# for each word in the list of words*

**for** word **in** words:

*# if the words are alread vectors, then just append them*

**if** isinstance(word, np.ndarray): mean.append(word)

*# if not: first get the vector embedding for the words*

**elif** word **in** wv.vocab: mean.append(wv.syn0norm[wv.vocab[word].index]) all\_words.add(wv.vocab[word].index)

**if not** mean:

*# error handling in case mean cannot be calculated* logging.warning("cannot compute similarity with no input **s**", words) **return** np.zeros(wv.vector\_size,)

*# use gensim's method to calculate the mean of all the words appended to mean list* mean = gensim.matutils.unitvec(np.array(mean).mean(axis=0)).astype(np.float32) **return** mean

**def** word\_averaging\_list(wv, text\_list):

**return** np.vstack([word\_averaging(wv, post) **for** post **in** text\_list ])

Below, we explore a different way to create tokens out of sentences, by using the nltk toolkit.

In [0]: **import nltk.data**

nltk.download('punkt')

**def** w2v\_tokenize\_text(text):

*# create tokens, a list of words, for each post. This function will do some cleani*

tokens = []

**for** sent **in** nltk.sent\_tokenize(text, language='english'):

**for** word **in** nltk.word\_tokenize(sent, language='english'):

**if** len(word) < 2:

## continue

tokens.append(word)

**return** tokens

[nltk\_data] Downloading package punkt to /root/nltk\_data... [nltk\_data] Package punkt is already up-to-date!

Let’s split the dataset in training and test set like before, and tokenize each of the datasets

In [0]: train, test = train\_test\_split(df, test\_size=0.3, random\_state = 42)

test\_tokenized = test.apply(**lambda** r: w2v\_tokenize\_text(r['post']), axis=1).values train\_tokenized = train.apply(**lambda** r: w2v\_tokenize\_text(r['post']), axis=1).values

We can then average the position per post in this new dataset using the functions we defined above and based on our word2vec model wv.

In [0]: X\_train\_word\_average = word\_averaging\_list(wv,train\_tokenized) X\_test\_word\_average = word\_averaging\_list(wv,test\_tokenized)

/usr/local/lib/python3.6/dist-packages/ipykernel\_launcher.py:13: DeprecationWarning: Call to del sys.path[0]

/usr/local/lib/python3.6/dist-packages/gensim/matutils.py:737: FutureWarning: Conversion of th if np.issubdtype(vec.dtype, np.int):

WARNING:root:cannot compute similarity with no input [] WARNING:root:cannot compute similarity with no input ['ngrepeat']

Now we can feed this new representation into the logistic regression:

In [0]: **from sklearn.linear\_model import** LogisticRegression logreg = LogisticRegression(n\_jobs=1, C=1e5)

logreg = logreg.fit(X\_train\_word\_average, train['tags']) y\_pred = logreg.predict(X\_test\_word\_average)

/usr/local/lib/python3.6/dist-packages/sklearn/linear\_model/logistic.py:432: FutureWarning: De FutureWarning)

/usr/local/lib/python3.6/dist-packages/sklearn/linear\_model/logistic.py:469: FutureWarning: De "this warning.", FutureWarning)

How accurate is this averaged word2vec model with logistic regression?

In [0]: print('accuracy **s**' accuracy\_score(y\_pred, test.tags)) print(classification\_report(test.tags, y\_pred))

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| accuracy 0.6375 | precision | recall | f1-score | support |
| .net | 0.62 | 0.59 | 0.61 | 613 |
| android | 0.74 | 0.76 | 0.75 | 620 |
| angularjs | 0.65 | 0.67 | 0.66 | 587 |
| asp.net | 0.53 | 0.52 | 0.52 | 586 |
| c | 0.70 | 0.77 | 0.73 | 599 |
| c# | 0.44 | 0.39 | 0.41 | 589 |
| c++ | 0.65 | 0.60 | 0.63 | 594 |
| css | 0.73 | 0.80 | 0.76 | 610 |
| html | 0.60 | 0.61 | 0.60 | 617 |
| ios | 0.56 | 0.52 | 0.54 | 587 |
| iphone | 0.55 | 0.50 | 0.52 | 611 |
| java | 0.61 | 0.61 | 0.61 | 594 |
| javascript | 0.65 | 0.65 | 0.65 | 619 |
| jquery | 0.61 | 0.57 | 0.59 | 574 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| mysql | 0.70 | 0.71 | 0.71 | 584 |
| objective-c | 0.42 | 0.43 | 0.43 | 578 |
| php | 0.68 | 0.70 | 0.69 | 591 |
| python | 0.76 | 0.78 | 0.77 | 608 |
| ruby-on-rails | 0.82 | 0.83 | 0.82 | 638 |
| sql | 0.65 | 0.71 | 0.68 | 601 |
| accuracy |  |  | 0.64 | 12000 |
| macro avg | 0.63 | 0.64 | 0.63 | 12000 |
| weighted avg | 0.63 | 0.64 | 0.64 | 12000 |

Now you can see that the accuracy went down! Oh no! Why is that? Because we used a very naive approach, to average our vectors. The way around it would be doc2vec, which learns relationships between documents (posts in this case), instead of words. The accuracy could also improve by using a different classifier instead of logistic regression, or by changing the aggrega- tion strategy.

# Doc2vec and Logistic Regression (advanced)

The idea of word2vec can be extended to documents whereby instead of learning feature repre- sentations for words, we learn it for sentences or documents. To get a general idea of a word2vec, think of it as a mathematical average of the word vector representations of all the words in the document. Doc2Vec extends the idea of word2vec, however words can only capture so much, there are times when we need relationships between documents and not just words.

The way to train doc2vec model for our Stack Overflow questions and tags data is very similar with when we trained multi-class text classification with word2vec and logistic regression above. First, we label the sentences. Gensim’s Doc2Vec implementation requires each docu- ment/paragraph to have a label associated with it that indicates if it’s part of the test or training set. We do this by using the TaggedDocument method. The format will be “TRAIN\_i” or “TEST\_i”

where “i” is a dummy index of the post.

First let’s import the necessary libraries.

In [0]: **from tqdm import** tqdm

## from gensim.models import doc2vec

**from sklearn import** utils

**import gensim**

**from gensim.models.doc2vec import** TaggedDocument

## import re

Let’s start by defining a function that labels our documents in the corpus. We just give them dummy labels TRAIN\_i or TEST\_i for post i. Given a corpus and labels, we return a variable that includes a label indicating if it’s test or training data.

In [0]: **def** label\_sentences(corpus, label\_type):

*"""*

*Gensim's Doc2Vec implementation requires each document/paragraph to have a label a We do this by using the TaggedDocument method. The format will be "TRAIN\_i" or "TE*

*a dummy index of the post. """*

labeled = []

**for** i, v **in** enumerate(corpus):

label = label\_type + '\_' + str(i) labeled.append(doc2vec.TaggedDocument(v.split(), [label]))

**return** labeled

Just like above we split our dataset up in test and training data.

In [0]: X\_train, X\_test, y\_train, y\_test = train\_test\_split(df.post, df.tags, random\_state=0, X\_train = label\_sentences(X\_train, 'Train')

X\_test = label\_sentences(X\_test, 'Test') all\_data = X\_train + X\_test

Let’s have a look how our data looks at this moment:

In [0]: all\_data[:10]

Out[0]: [TaggedDocument(words=['fulltext', 'search', 'php', 'pdo', 'returning', 'result', 'sea TaggedDocument(words=['select', 'everything', '1', 'table', 'x', 'rows', 'another', ' TaggedDocument(words=['r', 'cannot', 'resolved', 'variable', 'importing', 'project', TaggedDocument(words=['efficient', 'way', 'get', 'values', 'object', 'based', 'id', ' TaggedDocument(words=['aspnet', 'limit', 'parameter', 'length', 'querystring', 'probl TaggedDocument(words=['ruby', 'rails', 'fetch', 'display', 'descendent', 'records', ' TaggedDocument(words=['canceling', 'fade', 'effect', 'tooltip', 'hover', 'need', 'too TaggedDocument(words=['ajax', 'calender', 'working', 'ie', 'using', 'ajax', 'calender TaggedDocument(words=['c++', 'random', 'number', 'generator', 'hung', 'whenever', 'at TaggedDocument(words=['bit', 'vector', 'looked', 'online', 'good', 'seem', 'find', 'g

Gensim allows us to build a model very easily. We can vary the parameters to fit your data:

* dm=0 , distributed bag of words (DBOW) is used.
* vector\_size=300 , 300 vector dimensional feature vectors.
* negative=5 , specifies how many “noise words” should be drawn.
* min\_count=1, ignores all words with total frequency lower than this.
* alpha=0.065 , the initial learning rate.

We initialize the model and train for 30 epochs. (slower computers may want to train for less epochs). Be sure to set your runtime to TPU/GPU hardware acceleration! Maybe test with a lower amount of epochs first to see how high you can go during class time!

In [0]: model\_dbow = Doc2Vec(dm=0, vector\_size=300, negative=5, min\_count=1, alpha=0.065, min\_ model\_dbow.build\_vocab([x **for** x **in** tqdm(all\_data)])

100 || 40000/40000 [00:00<00:00, 1619281.72it/s]

In [0]: **for** epoch **in** range(30):

model\_dbow.train(utils.shuffle([x **for** x **in** tqdm(all\_data)]), total\_examples=len(al model\_dbow.alpha -= 0.002

model\_dbow.min\_alpha = model\_dbow.alpha

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 100 | || | 40000/40000 | [00:00<00:00, | 1742363.28it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2388283.79it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2230775.45it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2113079.34it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2212710.82it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2216657.55it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2333279.93it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 3057295.72it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2741688.76it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 3026903.13it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2683882.20it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2664657.41it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2356847.09it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2838640.34it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2127765.22it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2252701.00it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2347185.99it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2467709.42it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2876998.37it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2293253.87it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2717838.33it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2094481.54it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 3184680.62it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2300549.32it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2630936.03it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2480809.13it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2181182.04it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2462963.68it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2129736.98it/s] |
| 100 | || | 40000/40000 | [00:00<00:00, | 2881247.49it/s] |



Now let’s define a function to the vectors out of this trained model, so that we can feed them into the logistic regression:

In [0]: **def** get\_vectors(model, corpus\_size, vectors\_size, vectors\_type):

*"""*

*Get vectors from trained doc2vec model*

*:param doc2vec\_model: Trained Doc2Vec model*

*:param corpus\_size: Size of the data*

*:param vectors\_size: Size of the embedding vectors*

*:param vectors\_type: Training or Testing vectors*

*:return: list of vectors """*

vectors = np.zeros((corpus\_size, vectors\_size))

**for** i **in** range(0, corpus\_size):

prefix = vectors\_type + '\_' + str(i) vectors[i] = model.docvecs[prefix]

**return** vectors

We can use this function to create a vectorised training and test set with 1 entry per document for the input in classification models such as logistic regression.

In [0]: train\_vectors\_dbow = get\_vectors(model\_dbow, len(X\_train), 300, 'Train') test\_vectors\_dbow = get\_vectors(model\_dbow, len(X\_test), 300, 'Test')

We can now feed these vectors to the classifier again:

In [0]: logreg = LogisticRegression(n\_jobs=1, C=1e5) logreg.fit(train\_vectors\_dbow, y\_train)

logreg = logreg.fit(train\_vectors\_dbow, y\_train) y\_pred = logreg.predict(test\_vectors\_dbow)

print('accuracy **s**' accuracy\_score(y\_pred, y\_test)) print(classification\_report(y\_test, y\_pred))

/usr/local/lib/python3.6/dist-packages/sklearn/linear\_model/logistic.py:432: FutureWarning: De FutureWarning)

/usr/local/lib/python3.6/dist-packages/sklearn/linear\_model/logistic.py:469: FutureWarning: De "this warning.", FutureWarning)

/usr/local/lib/python3.6/dist-packages/sklearn/linear\_model/logistic.py:432: FutureWarning: De FutureWarning)

/usr/local/lib/python3.6/dist-packages/sklearn/linear\_model/logistic.py:469: FutureWarning: De "this warning.", FutureWarning)

accuracy 0.8081666666666667

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| precision | | recall | f1-score | support |
| .net | 0.72 | 0.69 | 0.70 | 589 |
| android | 0.89 | 0.91 | 0.90 | 661 |
| angularjs | 0.95 | 0.95 | 0.95 | 606 |
| asp.net | 0.79 | 0.78 | 0.79 | 613 |
| c | 0.82 | 0.89 | 0.85 | 601 |
| c# | 0.73 | 0.72 | 0.72 | 585 |
| c++ | 0.85 | 0.79 | 0.82 | 621 |
| css | 0.83 | 0.86 | 0.84 | 587 |
| html | 0.70 | 0.67 | 0.69 | 560 |
| ios | 0.68 | 0.65 | 0.66 | 611 |
| iphone | 0.67 | 0.68 | 0.68 | 593 |
| java | 0.81 | 0.84 | 0.83 | 581 |
| javascript | 0.80 | 0.79 | 0.79 | 608 |
| jquery | 0.86 | 0.85 | 0.85 | 593 |
| mysql | 0.83 | 0.83 | 0.83 | 592 |
| objective-c | 0.71 | 0.65 | 0.68 | 597 |
| php | 0.83 | 0.85 | 0.84 | 604 |
| python | 0.89 | 0.93 | 0.91 | 610 |
| ruby-on-rails | 0.94 | 0.95 | 0.94 | 595 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| sql | 0.81 | 0.84 | 0.82 | 593 |
| accuracy |  |  | 0.81 | 12000 |
| macro avg | 0.81 | 0.81 | 0.81 | 12000 |
| weighted avg | 0.81 | 0.81 | 0.81 | 12000 |

80%, that is the best result so far! Remember, we can actually use any classifier with this method! So up to you to make your project as efficient as possible :)

Try using a different classifiers, e.g. Decision tree or SVM. Does that influence the results? New methods are coming out every day in the field of data science. Just at the end of Au-

gust 2019, the first implementation of BERT for document classfication was published: DocBERT: https://arxiv.org/abs/1904.08398

# References

* https://radimrehurek.com/gensim/models/word2vec.html
* https://towardsdatascience.com/multi-class-text-classification-model-comparison-and- selection-5eb066197568
* https://github.com/kavgan/nlp-text-mining-working-examples/tree/master/word2vec
* [https://medium.com/@mishra.thedeepak/doc2vec-simple-implementation-example-](https://medium.com/%40mishra.thedeepak/doc2vec-simple-implementation-example-) df2afbbfbad5]