UDACITY

‹ Return to Classroom

NLP on Financial Statements

REVIEW

CODE REVIEW

HISTORY

Meets Specifications

Congratulations!

You have demonstrated an excellent understanding of the concepts in Project 5 and the ability to implement those concepts in Python.

- It has been a pleasure reviewing your project.

Great News!

- you are more than 1/2 way through this nanodegree!!
- Good luck on your last 3 projects!

Here are some additional resources which can help you take your newly acquired skills even further!

Blueshift (Formerly Quantopian) is a site for learning and practicing Quantitatve Python Programming that provides Python workspace similar to what you've been using in your nanodegree.

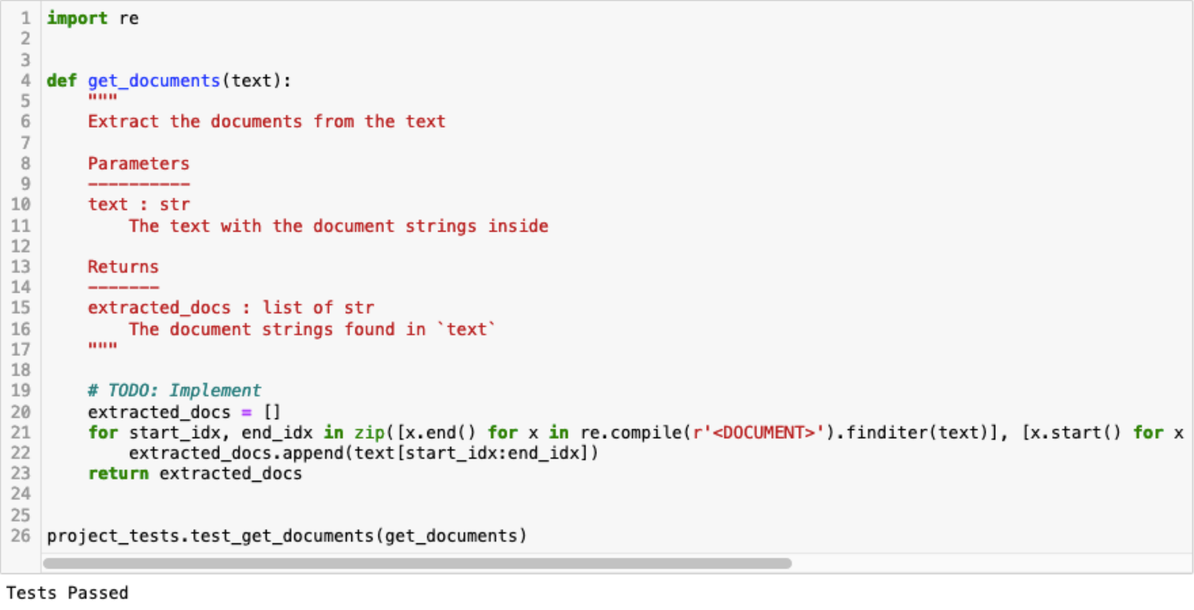
- Blueshift Tutorials : 3 full tutorials by Quantopian to assist with quant workflow.

10-Ks

The function `get_documents` extracts the documents from the text.

Well done!

- you set up a nice start pattern and a nice end pattern for using regular expressions to pull just the documents from an input string.



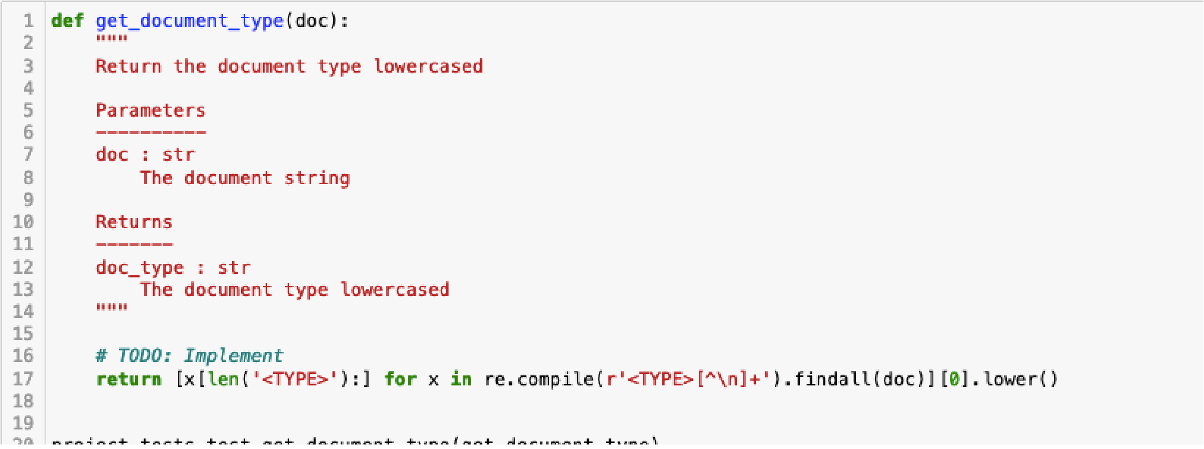
```
import re


def get_documents(text):
    """
    Extract the documents from the text

    Parameters
    ----------
    text : str
        The text with the document strings inside

    Returns
    -------
    extracted_docs : list of str
        The document strings found in `text`
    """

    # TODO: Implement
    extracted_docs = []
    for start_idx, end_idx in zip([x.end() for x in re.compile(r'<DOCUMENT>').finditer(text)], [x.start() for x
        extracted_docs.append(text[start_idx:end_idx])
    return extracted_docs


project_tests.test_get_documents(get_documents)
```

```
Tests Passed
```

The function `get_document_type` returns the document type lowercased.

Document Type is being determined properly.

- You have properly constructed a regular expression for obtaining the document type.
- You are returning the document type as a lowercase string.



```
def get_document_type(doc):
    """
    Return the document type lowercased

    Parameters
    ----------
    doc : str
        The document string

    Returns
    -------
    doc_type : str
        The document type lowercased
    """

    # TODO: Implement
    return [x[len('<TYPE>'):] for x in re.compile(r'<TYPE>[^\n]+').findall(doc)][0].lower()


project_tests.test_get_document_type(get_document_type)
```

```
project_tests.test_get_document_type(get_document_type)
```

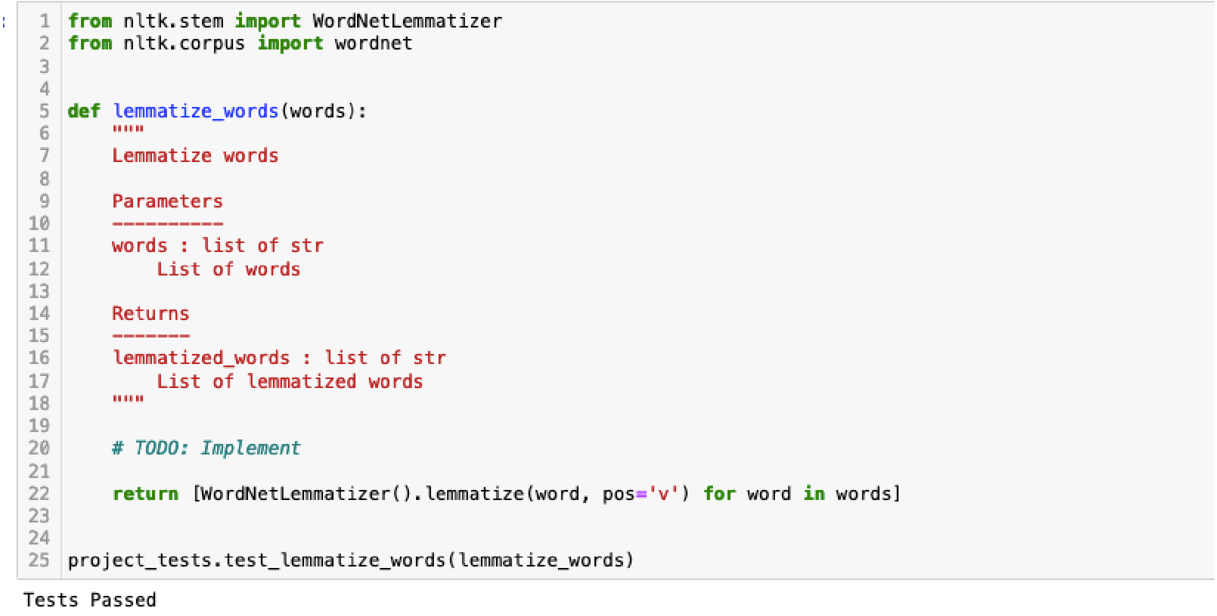
```
Tests Passed
```

Preprocess the Data

The function `lemmatize_words` lemmatizes verbs.

Verbs are being properly lemmatized!

- You are properly using **WordNetLemmatizer** to lemmatize verbs in the list of words.
- Nice use of a list comprehension!



```
from nltk.stem import WordNetLemmatizer
from nltk.corpus import wordnet


def lemmatize_words(words):
    """
    Lemmatize words

    Parameters
    ----------
    words : list of str
        List of words

    Returns
    -------
    lemmatized_words : list of str
        List of lemmatized words
    """

    # TODO: Implement

    return [WordNetLemmatizer().lemmatize(word, pos='v') for word in words]


project_tests.test_lemmatize_words(lemmatize_words)
```

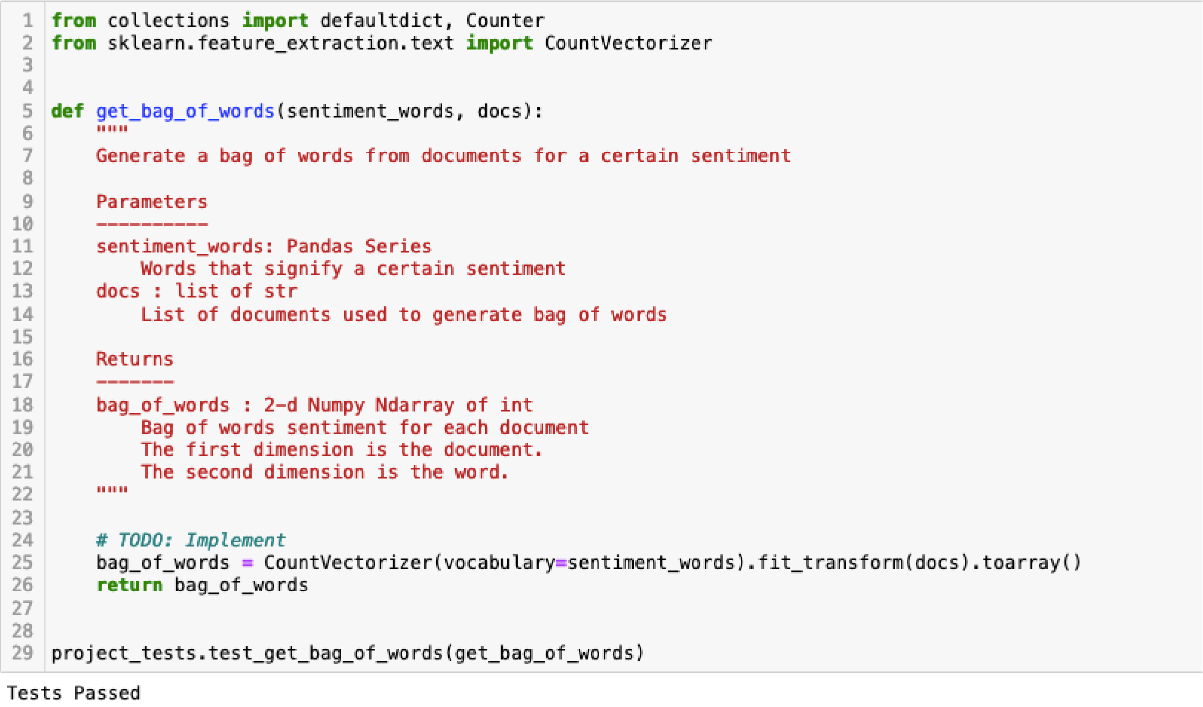
```
Tests Passed
```

Analysis on 10ks

The function `get_bag_of_words` generates a bag of words from documents.

Sentiment Bag of Words is properly created.

- You have nicely used **CountVectorizer** on the incoming **sentiment_words** and then used that to transform the incoming docs, creating a **"bag of words"**.



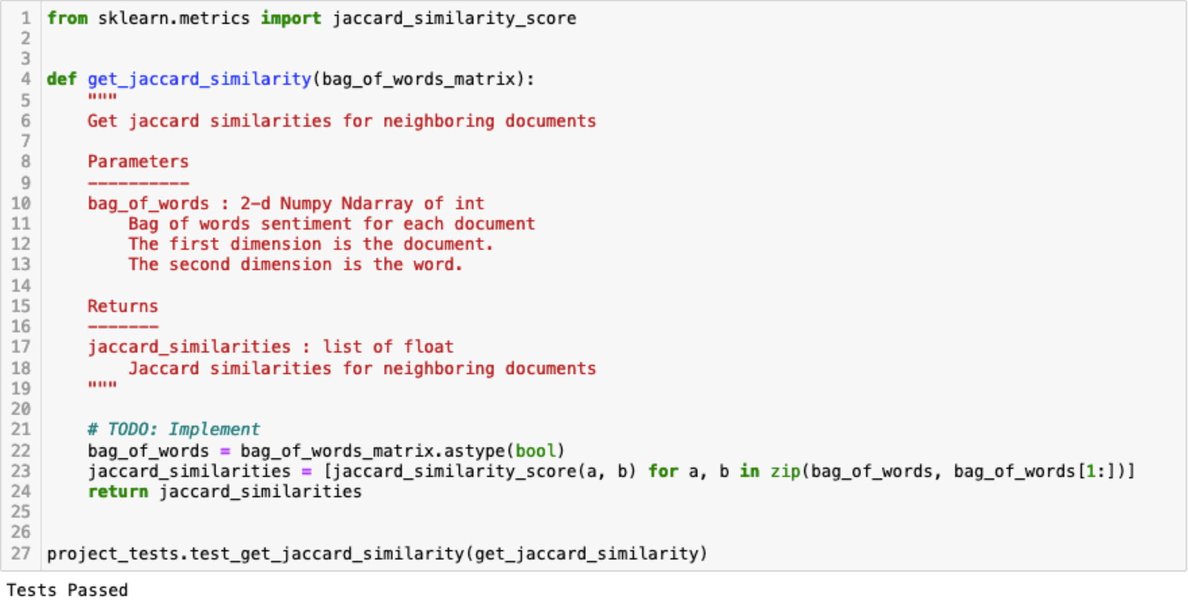
```
from collections import defaultdict, Counter
from sklearn.feature_extraction.text import CountVectorizer


def get_bag_of_words(sentiment_words, docs):
    """
    Generate a bag of words from documents for a certain sentiment

    Parameters
    ----------
    sentiment_words: Pandas Series
        Words that signify a certain sentiment
    docs : list of str
        List of documents used to generate bag of words

    Returns
    -------
    bag_of_words : 2-d Numpy Ndarray of int
        Bag of words sentiment for each document
        The first dimension is the document.
        The second dimension is the word.
    """

    # TODO: Implement
    bag_of_words = CountVectorizer(vocabulary=sentiment_words).fit_transform(docs).toarray()
    return bag_of_words


project_tests.test_get_bag_of_words(get_bag_of_words)
```

```
Tests Passed
```

The function `get_jaccard_similarity` calculates the jaccard similarities for neighboring documents.

You are correctly calculating the Jaccard Similarity on the Bag of Words

- Nice use of **list comprehension** and **zip** to calculate Jaccard similarities for neighboring documents!



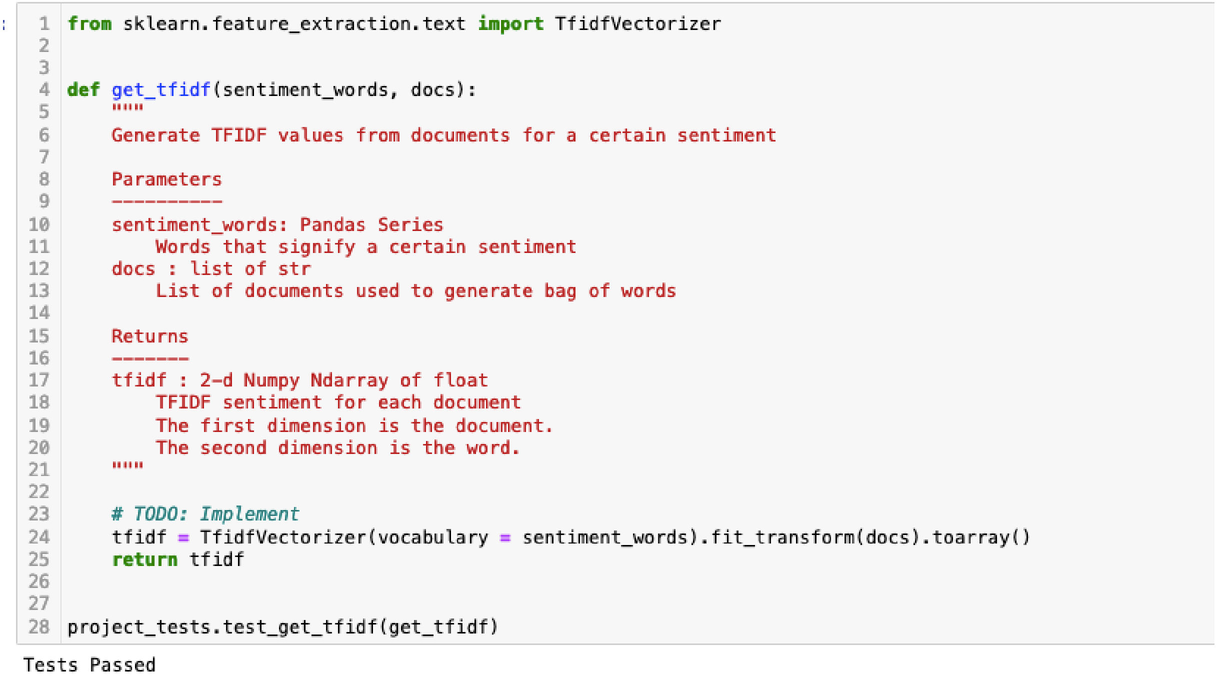
```
from sklearn.metrics import jaccard_similarity_score


def get_jaccard_similarity(bag_of_words_matrix):
    """
    Get jaccard similarities for neighboring documents

    Parameters
    ----------
    bag_of_words : 2-d Numpy Ndarray of int
        Bag of words sentiment for each document
        The first dimension is the document.
        The second dimension is the word.

    Returns
    -------
    jaccard_similarities : list of float
        Jaccard similarities for neighboring documents
    """

    # TODO: Implement
    bag_of_words = bag_of_words_matrix.astype(bool)
    jaccard_similarities = [jaccard_similarity_score(a, b) for a, b in zip(bag_of_words, bag_of_words[1:])]
    return jaccard_similarities


project_tests.test_get_jaccard_similarity(get_jaccard_similarity)
```

```
Tests Passed
```

The function `tfidf` generate TFIDF vectors for each document.

Yes!

– You used **TfidfVectorizer** (in a similar manner to how you used CountVectorizer for your bag_of_words) to generate the sentiment TFIDF from the 10-k documents using the sentiment words as the terms.

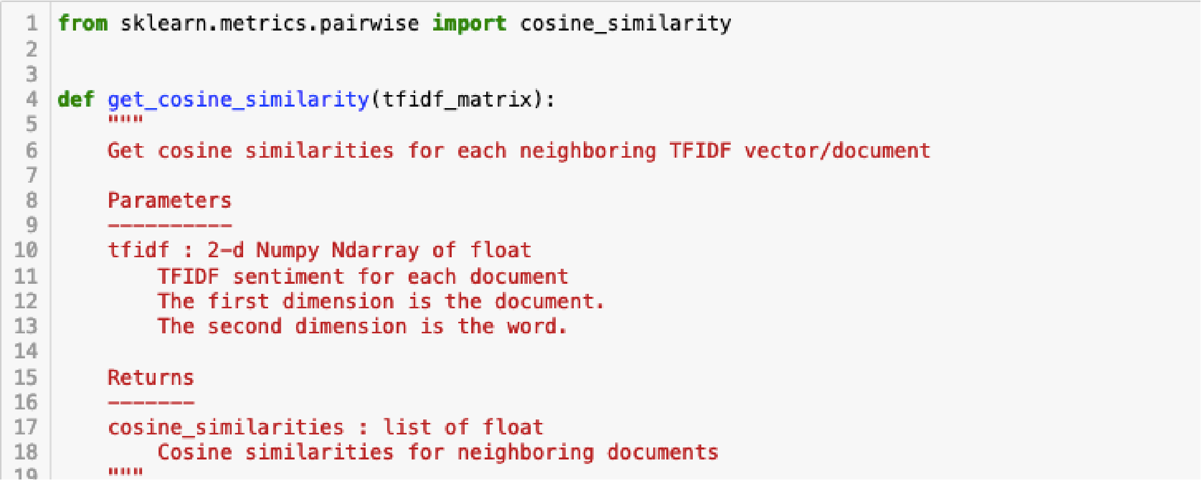


```
from sklearn.feature_extraction.text import TfidfVectorizer


def get_tfidf(sentiment_words, docs):
    """
    Generate TFIDF values from documents for a certain sentiment

    Parameters
    ----------
    sentiment_words: Pandas Series
        Words that signify a certain sentiment
    docs : list of str
        List of documents used to generate bag of words

    Returns
    -------
    tfidf : 2-d Numpy Ndarray of float
        TFIDF sentiment for each document
        The first dimension is the document.
        The second dimension is the word.
    """

    # TODO: Implement
    tfidf = TfidfVectorizer(vocabulary = sentiment_words).fit_transform(docs).toarray()
    return tfidf


project_tests.test_get_tfidf(get_tfidf)
```

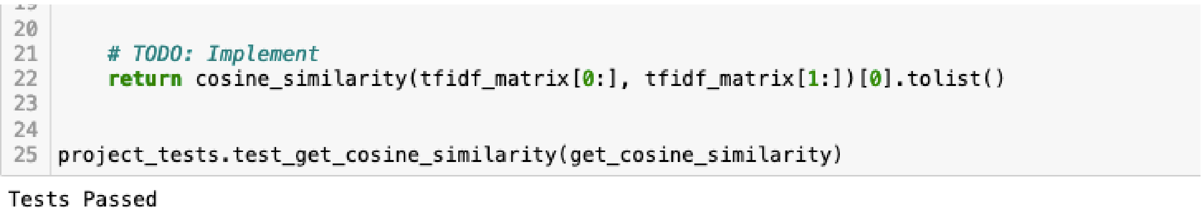
```
Tests Passed
```

The function `get_cosine_similarity` calculates the cosine similarities for each neighboring TFIDF vector/document.

Cosine Similarity is Accurately calculated using the TFIDF values!



```
from sklearn.metrics.pairwise import cosine_similarity


def get_cosine_similarity(tfidf_matrix):
    """
    Get cosine similarities for each neighboring TFIDF vector/document

    Parameters
    ----------
    tfidf : 2-d Numpy Ndarray of float
        TFIDF sentiment for each document
        The first dimension is the document.
        The second dimension is the word.

    Returns
    -------
    cosine_similarities : list of float
        Cosine similarities for neighboring documents
    """
```



```
    # TODO: Implement
    return cosine_similarity(tfidf_matrix[0:], tfidf_matrix[1:])[0].tolist()


project_tests.test_get_cosine_similarity(get_cosine_similarity)
```

```
Tests Passed
```

In case you are interested . . .

Here is a solution that uses numpy's diag function and does not require looping:

```
return list(np.diag(cosine_similarity(tfidf_matrix, tfidf_matrix), k=1))
```

DOWNLOAD PROJECT

RETURN TO PATH