

Assignment 4 - Naive Machine Translation and LSH

You will now implement your first machine translation system and then you will see how locality sensitive hashing works. Let's get started by importing the required functions!

If you are running this notebook in your local computer, don't forget to download the twitter samples and stopwords from nltk.

```
nltk.download('stopwords')
nltk.download('twitter_samples')
```

Important Note on Submission to the AutoGrader

Before submitting your assignment to the AutoGrader, please make sure you are not doing the following:

1. You have not added any `extra print` statement(s) in the assignment.
2. You have not added any `extra code cell(s)` in the assignment.
3. You have not changed any of the function parameters.
4. You are not using any global variables inside your graded exercises. Unless specifically instructed to do so, please refrain from it and use the local variables instead.
5. You are not changing the assignment code where it is not required, like creating `extra` variables.

If you do any of the following, you will get something like, `Grader Error: Grader feedback not found` (or similarly unexpected) error upon submitting your assignment. Before asking for help/debugging the errors in your assignment, check for these first. If this is the case, and you don't remember the changes you have made, you can get a fresh copy of the assignment by following these [instructions](#).

Table of Contents

- [1. The Word Embeddings Data for English and French Words](#)
 - [1.1 Generate Embedding and Transform Matrices](#)
 - [Exercise 1 - get_matrices \(UNQ_C1\)](#)
- [2 - Translations](#)
 - [2.1 - Translation as Linear Transformation of Embeddings](#)
 - [Exercise 2 - compute_loss \(UNQ_C3\)](#)
 - [Exercise 3 - compute_gradient \(UNQ_C4\)](#)
 - [Exercise 4 - align_embeddings \(UNQ_C5\)](#)
 - [2.2 - Testing the Translation](#)
 - [Exercise 5 - nearest_neighbor \(UNQ_C8\)](#)
 - [Exercise 6 - test_vocabulary \(UNQ_C10\)](#)
- [3 - LSH and Document Search](#)
 - [3.1 - Getting the Document Embeddings](#)
 - [Exercise 7 - get_document_embedding \(UNQ_C12\)](#)
 - [Exercise 8 - get_document_vecs \(UNQ_C14\)](#)
 - [3.2 - Looking up the Tweets](#)
 - [3.3 - Finding the most Similar Tweets with LSH](#)
 - [3.4 - Getting the Hash Number for a Vector](#)
 - [Exercise 9 - hash_value_of_vector \(UNQ_C17\)](#)
 - [3.5 - Creating a Hash Table](#)
 - [Exercise 10 - make_hash_table \(UNQ_C19\)](#)
 - [3.6 - Creating all Hash Tables](#)
 - [Exercise 11 - approximate_knn \(UNQ_C21\)](#)

```

import pdb
import pickle
import string

import time

import nltk
import numpy as np
from nltk.corpus import stopwords, twitter_samples

from utils import (cosine_similarity, get_dict,
                   process_tweet)
from os import getcwd

```

```

# add folder, tmp2, from our Local workspace containing pre-downloaded corpora files to r
filePath = f"{getcwd()}/tmp2/"
nltk.data.path.append(filePath)

```

1. The Word Embeddings Data for English and French Words

Write a program that translates English to French.

The Data

The full dataset for English embeddings is about 3.64 gigabytes, and the French embeddings are about 629 megabytes. To prevent the Coursera workspace from crashing, we've extracted a subset of the embeddings for the words that you'll use in this assignment.

The subset of data

To do the assignment on the Coursera workspace, we'll use the subset of word embeddings.

```

en_embeddings_subset = pickle.load(open("./data/en_embeddings.p", "rb"))
fr_embeddings_subset = pickle.load(open("./data/fr_embeddings.p", "rb"))

```

Look at the data

- en_embeddings_subset: the key is an English word, and the value is a 300 dimensional array, which is the embedding for that word.

```
'the': array([ 0.08007812,  0.10498047,  0.04980469,  0.0534668 , -0.0673828
1, ....
```

- fr_embeddings_subset: the key is a French word, and the value is a 300 dimensional array, which is the embedding for that word.

```
'la': array([-6.18250e-03, -9.43867e-04, -8.82648e-03,  3.24623e-02, ...
```

Load two dictionaries mapping the English to French words

- A training dictionary
- and a testing dictionary.

```
# loading the english to french dictionaries
en_fr_train = get_dict('./data/en-fr.train.txt')
print('The length of the English to French training dictionary is', len(en_fr_train))
en_fr_test = get_dict('./data/en-fr.test.txt')
```

The length of the English to French training dictionary is 5000
The length of the English to French test dictionary is 1500

Looking at the English French dictionary

- `en_fr_train` is a dictionary where the key is the English word and the value is the French translation of that English word.

```
{'the': 'la',
 'and': 'et',
 'was': 'étais',
 'for': 'pour',
```

- `en_fr_test` is similar to `en_fr_train`, but is a test set. We won't look at it until we get to testing.

1.1 Generate Embedding and Transform Matrices

Exercise 1 - get_matrices

Translating English dictionary to French by using embeddings.

You will now implement a function `get_matrices`, which takes the loaded data and returns matrices `X` and `Y`.

Inputs:

- `en_fr` : English to French dictionary
- `en_embeddings` : English to embeddings dictionary
- `fr_embeddings` : French to embeddings dictionary

Returns:

- Matrix `X` and matrix `Y`, where each row in `X` is the word embedding for an English word, and the same row in `Y` is the word embedding for the French version of that English word.



alternate text

Figure 1

Use the `en_fr` dictionary to ensure that the i th row in the `X` matrix corresponds to the i th row in the `Y` matrix.

Instructions: Complete the function `get_matrices()`:

- Iterate over English words in `en_fr` dictionary.
- Check if the word have both English and French embedding.

Hints

```

# UNQ_C1 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
def get_matrices(en_fr, french_vecs, english_vecs):
    """
    Creates matrices of word embeddings for English and French words that are mapped to each other in en_fr.

    Inputs:
        en_fr: Dictionary mapping English words to French words.
        french_vecs: Dictionary of French word embeddings.
        english_vecs: Dictionary of English word embeddings.

    Outputs:
        X: Matrix with each row being the embedding of an English word. Shape is (number_of_english_words, dimension).
        Y: Matrix with each row being the embedding of the corresponding French word. Shape is (number_of_french_words, dimension).

    Note:
        This function does not compute or return a projection matrix.
    """
    #####
    ### START CODE HERE #####
    # X_L and Y_L are lists of the english and french word embeddings
    X_L = []
    Y_L = []

    # get the english words (the keys in the dictionary) and store in a set()
    english_set = set(english_vecs.keys())

    # get the french words (keys in the dictionary) and store in a set()
    french_set = set(french_vecs.keys())

    # Loop through all english, french word pairs in the english french dictionary
    for en_word, fr_word in en_fr.items():

        # check that the french word has an embedding and that the english word has an embedding
        if fr_word in french_set and en_word in english_set:

            # get the english embedding
            en_vec = english_vecs[en_word]

            # get the french embedding
            fr_vec = french_vecs[fr_word]

            # add the english embedding to the list
            X_L.append(en_vec)

            # add the french embedding to the list
            Y_L.append(fr_vec)

    # stack the vectors of X_L into a matrix X
    X = np.vstack(X_L)

    # stack the vectors of Y_L into a matrix Y
    Y = np.vstack(Y_L)
    ### END CODE HERE #####

```

Now we will use function `get_matrices()` to obtain sets `X_train` and `Y_train` of English and French word embeddings into the corresponding vector space models.

```

# UNQ_C2 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# You do not have to input any code in this cell, but it is relevant to grading, so please
# getting the training set:
X_train, Y_train = get_matrices(
    ... for X_train ... for embeddings subset ... for embeddings subset ...
)
# Test your function
... for test function ...

```

All tests passed

2 - Translations

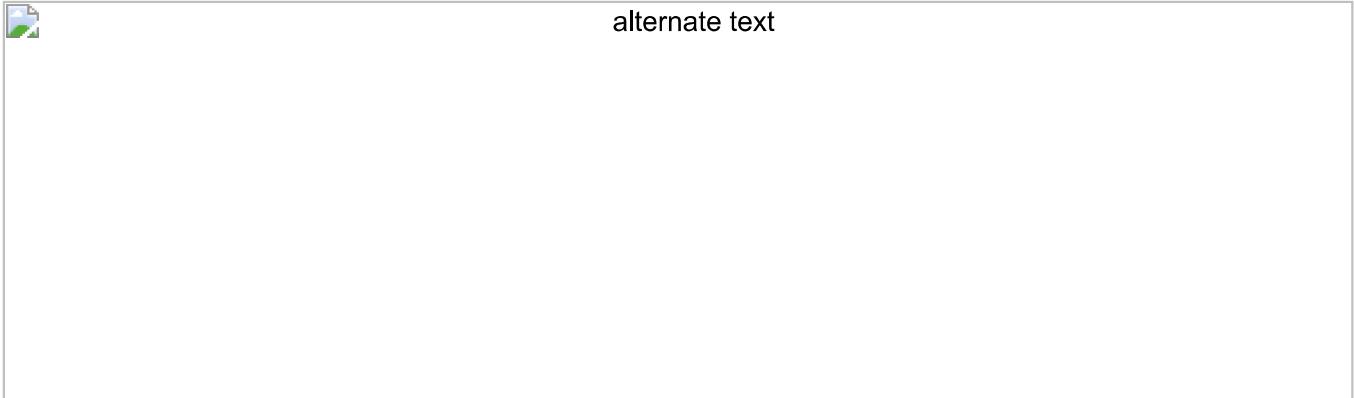


Figure 2

Write a program that translates English words to French words using word embeddings and vector space models.

2.1 - Translation as Linear Transformation of Embeddings

Given dictionaries of English and French word embeddings you will create a transformation matrix R

- Given an English word embedding, e , you can multiply eR to get a new word embedding f .
 - Both e and f are row vectors.
- You can then compute the nearest neighbors to f in the french embeddings and recommend the word that is most similar to the transformed word embedding.

Describing translation as the minimization problem

Find a matrix R that minimizes the following equation.

$$\arg \min_R \|XR - Y\|_F \quad (1)$$

Frobenius norm

The Frobenius norm of a matrix A (assuming it is of dimension m, n) is defined as the square root of the sum of the absolute squares of its elements:

$$\|A\|_F \equiv \sqrt{\sum_{i=1}^m \sum_{j=1}^n |a_{ij}|^2} \quad (2)$$

Actual loss function

In the real world applications, the Frobenius norm loss:

$$\|\mathbf{X}\mathbf{R} - \mathbf{Y}\|_F$$

is often replaced by its squared value divided by m :

$$\frac{1}{m} \|\mathbf{X}\mathbf{R} - \mathbf{Y}\|_F^2$$

where m is the number of examples (rows in \mathbf{X}).

- The same \mathbf{R} is found when using this loss function versus the original Frobenius norm.
- The reason for taking the square is that it's easier to compute the gradient of the squared Frobenius.
- The reason for dividing by m is that we're more interested in the average loss per embedding than the loss for the entire training set.
 - The loss for all training set increases with more words (training examples), so taking the average helps us to track the average loss regardless of the size of the training set.

[Optional] Detailed explanation why we use norm squared instead of the norm:

Click for optional details

Implementing translation mechanism described in this section.

Exercise 2 - compute_loss

Step 1: Computing the loss

- The loss function will be squared Frobenius norm of the difference between matrix and its approximation, divided by the number of training examples m .
- Its formula is:

$$L(\mathbf{X}, \mathbf{Y}, \mathbf{R}) = \frac{1}{m} \sum_{i=1}^m \sum_{j=1}^n (a_{ij})^2$$

where a_{ij} is value in i th row and j th column of the matrix $\mathbf{X}\mathbf{R} - \mathbf{Y}$.

Instructions: complete the `compute_loss()` function

- Compute the approximation of \mathbf{Y} by matrix multiplying \mathbf{X} and \mathbf{R}
- Compute difference $\mathbf{X}\mathbf{R} - \mathbf{Y}$
- Compute the squared Frobenius norm of the difference and divide it by m .

Hints

```

# UNQ_C3 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
def compute_loss(X, Y, R):
    ...
    Inputs:
        X: a matrix of dimension (m,n) where the rows are the English embeddings.
        Y: a matrix of dimension (m,n) where the rows correspond to the French embeddings
        R: a matrix of dimension (n,n) - transformation matrix from English to French vec
    Outputs:
        loss: scalar - the value of the loss function for given X, Y and R.
    ...
    ### START CODE HERE ###
    # m is the number of rows in X
    m = X.shape[0]

    # diff is XR - Y
    diff = np.dot(X, R) - Y

    # diff_squared is the element-wise square of the difference
    diff_squared = np.square(diff)

    # sum_diff_squared is the sum of the squared elements
    sum_diff_squared = np.sum(diff_squared)

    # Loss is the sum_diff_squared divided by the number of examples (m)
    loss = sum_diff_squared / m
    ### END CODE HERE ###

```

```

# Testing your implementation.
np.random.seed(123)
m = 10
n = 5
X = np.random.rand(m, n)
Y = np.random.rand(m, n) * .1
R = np.random.rand(n, n)

```

Expected loss for an experiment with random matrices: 8.1866

Expected output:

Expected loss for an experiment with random matrices: 8.1866

```

# Test your function
# !unitest -t compute_loss(compute_loss)
All tests passed

```

Exercise 3 - compute_gradient

Step 2: Computing the gradient of loss with respect to transform matrix R

- Calculate the gradient of the loss with respect to transform matrix R .
- The gradient is a matrix that encodes how much a small change in R affect the change in the loss function.
- The gradient gives us the direction in which we should decrease R to minimize the loss.
- m is the number of training examples (number of rows in X).
- The formula for the gradient of the loss function $L(X, Y, R)$ is:

$$\frac{d}{dR} L(X, Y, R) = \frac{d}{dR} \left(\frac{1}{m} \|XR - Y\|_F^2 \right) = \frac{2}{m} X^T (XR - Y)$$

Instructions: Complete the `compute_gradient` function below.

Hints

```
# UNQ_C4 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
def compute_gradient(X, Y, R):
    ...
    Inputs:
        X: a matrix of dimension (m,n) where the rows are the English embeddings.
        Y: a matrix of dimension (m,n) where the rows correspond to the French embeddings
        R: a matrix of dimension (n,n) - transformation matrix from English to French vec
    Outputs:
        gradient: a matrix of dimension (n,n) - gradient of the loss function L for given
    ...
    ### START CODE HERE ###
    # m is the number of rows in X
    m = X.shape[0]

    # gradient is X^T(XR - Y) * 2/m
    gradient = (2 / m) * np.dot(X.T, (np.dot(X, R) - Y))
    ### END CODE HERE ###
    ...
    return gradient
```

```
# Testing your implementation.
np.random.seed(123)
m = 10
n = 5
X = np.random.rand(m, n)
Y = np.random.rand(m, n) * .1
R = np.random.rand(n, n)
gradient = compute_gradient(X, Y, R)
print("First row of the gradient matrix: ", gradient[0])
```

First row of the gradient matrix: [1.3498175 1.11264981 0.69626762 0.98468499 1.33828969]

Expected output:

First row of the gradient matrix: [1.3498175 1.11264981 0.69626762 0.98468499 1.33828969]

```
# Test your function
# unittest test_compute_gradient(compute_gradient)
All tests passed
```

Step 3: Finding the optimal R with Gradient Descent Algorithm

Gradient Descent

Gradient descent is an iterative algorithm which is used in searching for the optimum of the function.

- Earlier, we've mentioned that the gradient of the loss with respect to the matrix encodes how much a tiny change in some coordinate of that matrix affect the change of loss function.
- Gradient descent uses that information to iteratively change matrix `R` until we reach a point where the loss is minimized.

Training with a fixed number of iterations

Most of the time we iterate for a fixed number of training steps rather than iterating until the loss falls below a threshold.

OPTIONAL: explanation for fixed number of iterations
[click here for detailed discussion](#)

Pseudocode:

1. Calculate gradient g of the loss with respect to the matrix R .
2. Update R with the formula:

$$R_{\text{new}} = R_{\text{old}} - \alpha g$$

Where α is the learning rate, which is a scalar.

Learning Rate

- The learning rate or "step size" α is a coefficient which decides how much we want to change R in each step.
- If we change R too much, we could skip the optimum by taking too large of a step.
- If we make only small changes to R , we will need many steps to reach the optimum.
- Learning rate α is used to control those changes.
- Values of α are chosen depending on the problem, and we'll use `learning_rate = 0.0003` as the default value for our algorithm.

Exercise 4 - align_embeddings

Implement `align_embeddings()`

Hints

```

# UNQ_C5 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
def align_embeddings(X, Y, train_steps=100, learning_rate=0.0003, verbose=True, compute_loss=True):
    """
    Inputs:
        X: a matrix of dimension (m,n) where the rows are the English embeddings.
        Y: a matrix of dimension (m,n) where the rows correspond to the French embeddings
        train_steps: positive int - number of iterations for gradient descent.
        learning_rate: positive float - step size for gradient descent.
    Outputs:
        R: a matrix of dimension (n,n) - the projection matrix that minimizes the Frobenius norm of X - RY
    """
    np.random.seed(129)

    # Initialize R as a square matrix with dimensions equal to the embedding size
    R = np.random.rand(X.shape[1], X.shape[1])

    for i in range(train_steps):
        if verbose and i % 25 == 0:
            print(f"loss at iteration {i} is: {compute_loss(X, Y, R):.4f}")

        ### START CODE HERE ###
        # Compute gradient using the provided function
        gradient = compute_gradient(X, Y, R)

        # Update R using gradient descent
        R -= learning_rate * gradient
        ### END CODE HERE ###

    return R

```

```

# UNQ_C6 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# You do not have to input any code in this cell, but it is relevant to grading, so please do not delete it.

# Testing your implementation.
np.random.seed(129)
m = 10
n = 5
X = np.random.rand(m, n)
Y = np.random.rand(m, n) * .1
R = align_embeddings(X, Y)

loss at iteration 0 is: 3.7242
loss at iteration 25 is: 3.6283
loss at iteration 50 is: 3.5350
loss at iteration 75 is: 3.4442

```

Expected Output:

```

loss at iteration 0 is: 3.7242
loss at iteration 25 is: 3.6283
loss at iteration 50 is: 3.5350
loss at iteration 75 is: 3.4442

```

```

# Test your function
# !/usr/bin/python3
# coding: utf-8
# This file contains unit tests for align_embeddings function.

import numpy as np
from align import align_embeddings
from align import compute_loss
from align import compute_gradient
from align import check_align_embeddings
from align import check_compute_loss
from align import check_compute_gradient
from align import check_all

# Check if align_embeddings function works correctly
def test_align_embeddings():
    X = np.random.rand(10, 5)
    Y = np.random.rand(10, 5) * .1
    R = align_embeddings(X, Y)
    assert R.shape == (5, 5), "R should be a 5x5 matrix"
    assert np.all(np.isclose(R @ Y, X)), "R should minimize the Frobenius norm of X - RY"

# Check if compute_loss function works correctly
def test_compute_loss():
    X = np.random.rand(10, 5)
    Y = np.random.rand(10, 5) * .1
    R = np.random.rand(5, 5)
    loss = compute_loss(X, Y, R)
    assert np.isclose(loss, 3.7242), "loss should be approximately 3.7242"

# Check if compute_gradient function works correctly
def test_compute_gradient():
    X = np.random.rand(10, 5)
    Y = np.random.rand(10, 5) * .1
    R = np.random.rand(5, 5)
    grad = compute_gradient(X, Y, R)
    assert grad.shape == (5, 5), "grad should be a 5x5 matrix"
    assert np.all(np.isclose(grad @ R, X - Y)), "grad should be the gradient of the loss function"

# Check if all functions work correctly
def test_all():
    X = np.random.rand(10, 5)
    Y = np.random.rand(10, 5) * .1
    R = align_embeddings(X, Y)
    loss = compute_loss(X, Y, R)
    grad = compute_gradient(X, Y, R)
    assert np.isclose(loss, 3.7242), "loss should be approximately 3.7242"
    assert np.all(np.isclose(grad @ R, X - Y)), "grad should be the gradient of the loss function"

# Run all tests
check_all()

```

Calculate Transformation matrix R

Using just the training set, find the transformation matrix \mathbf{R} by calling the function `align_embeddings()`.

NOTE: The code cell below will take a few minutes to fully execute (~3 mins)

```
# UNQ_C7 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# You do not have to input any code in this cell, but it is relevant to grading, so please
```

```
loss at iteration 0 is: 963.0146
loss at iteration 25 is: 97.8292
loss at iteration 50 is: 26.8329
loss at iteration 75 is: 9.7893
loss at iteration 100 is: 4.3776
loss at iteration 125 is: 2.3281
loss at iteration 150 is: 1.4480
loss at iteration 175 is: 1.0338
loss at iteration 200 is: 0.8251
loss at iteration 225 is: 0.7145
loss at iteration 250 is: 0.6534
loss at iteration 275 is: 0.6185
loss at iteration 300 is: 0.5981
loss at iteration 325 is: 0.5858
loss at iteration 350 is: 0.5782
loss at iteration 375 is: 0.5735
```

Expected Output

```
loss at iteration 0 is: 963.0146
loss at iteration 25 is: 97.8292
loss at iteration 50 is: 26.8329
loss at iteration 75 is: 9.7893
loss at iteration 100 is: 4.3776
loss at iteration 125 is: 2.3281
loss at iteration 150 is: 1.4480
loss at iteration 175 is: 1.0338
loss at iteration 200 is: 0.8251
loss at iteration 225 is: 0.7145
loss at iteration 250 is: 0.6534
loss at iteration 275 is: 0.6185
loss at iteration 300 is: 0.5981
loss at iteration 325 is: 0.5858
loss at iteration 350 is: 0.5782
loss at iteration 375 is: 0.5735
```

2.2 - Testing the Translation

k-Nearest Neighbors Algorithm

[k-Nearest neighbors algorithm](#)

- k-NN is a method which takes a vector as input and finds the other vectors in the dataset that are closest to it.
- The 'k' is the number of "nearest neighbors" to find (e.g. k=2 finds the closest two neighbors).

Searching for the Translation Embedding

Since we're approximating the translation function from English to French embeddings by a linear transformation matrix \mathbf{R} , most of the time we won't get the exact embedding of a French word when we transform embedding \mathbf{e} of some particular English word into the French embedding space.

- This is where k -NN becomes really useful! By using 1-NN with \mathbf{eD} as input, we can search for an

Cosine Similarity

Cosine similarity between vectors u and v calculated as the cosine of the angle between them. The formula is

$$\cos(u, v) = \frac{u \cdot v}{\|u\| \|v\|}$$

- $\cos(u, v) = 1$ when u and v lie on the same line and have the same direction.
- $\cos(u, v) = -1$ when they have exactly opposite directions.
- $\cos(u, v) = 0$ when the vectors are orthogonal (perpendicular) to each other.

Note: Distance and similarity are pretty much opposite things.

- We can obtain distance metric from cosine similarity, but the cosine similarity can't be used directly as the distance metric.
- When the cosine similarity increases (towards 1), the "distance" between the two vectors decreases (towards 0).
- We can define the cosine distance between u and v as

$$d_{\cos}(u, v) = 1 - \cos(u, v)$$

Exercise 5 - nearest_neighbor

Complete the function `nearest_neighbor()`

Inputs:

- Vector `v`,
- A set of possible nearest neighbors `candidates`
- `k` nearest neighbors to find.
- The distance metric should be based on cosine similarity.
- `cosine_similarity` function is already implemented and imported for you. Its arguments are two vectors and it returns the cosine of the angle between them.
- Iterate over rows in `candidates`, and save the result of similarities between current row and vector `v` in a python list. Take care that similarities are in the same order as row vectors of `candidates`.
- Now you can use `numpy.argsort` to sort the indices for the rows of `candidates`.

Hints

```

# UNQ_C8 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
def nearest_neighbor(v, candidates, k=1, cosine_similarity=cosine_similarity):
    """
    Input:
        - v: the vector you are going to find the nearest neighbor for
        - candidates: a set or array of candidate vectors where we will find the neighbors
        - k: number of top nearest neighbors to find
    Output:
        - k_idx: the indices of the top k closest vectors in sorted form
    """
    ### START CODE HERE ###
    similarity_l = []

    # for each candidate vector...
    for row in candidates:
        # get the cosine similarity between the input vector and the candidate vector
        cos_similarity = cosine_similarity(v, row)

        # append the similarity to the list
        similarity_l.append(cos_similarity)

    # sort the similarity list and get the indices of the sorted list (ascending order)
    sorted_ids = np.argsort(similarity_l)

    # reverse the order to get descending order (most similar first)
    sorted_ids = sorted_ids[::-1]

    # get the indices of the k most similar candidate vectors
    k_idx = sorted_ids[:k]
    ### END CODE HERE ###

```

```

# UNQ_C9 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# You do not have to input any code in this cell, but it is relevant to grading, so please

# Test your implementation:
v = np.array([1, 0, 1])
candidates = np.array([[1, 0, 5], [-2, 5, 3], [2, 0, 1], [6, -9, 5], [9, 9, 9]])
[[2 0 1]
 [1 0 5]
 [9 9 9]]

```

Expected Output:

```
[[2 0 1]
 [1 0 5]
 [9 9 9]]
```

```
# Test your function
# !/usr/bin/python
# coding: utf-8
# This file is part of the Python aiocoap library
# Copyright: 2015-2016 Carsten Schubert
# License: LGPLv3+
# URL: https://github.com/carstenschubert/python_aiocoap

import aiocoap
import aiocoap.resource as resource
import aiocoap.message as message
from aiocoap import *

# Create a resource
class Counter(resource.Resource):
    def __init__(self):
        self.value = 0

    async def render_get(self, request):
        print("GET received")
        self.value += 1
        response = message.Message(code=message.CONTENT, payload="Value: " + str(self.value))
        return response

    async def render_post(self, request):
        print("POST received")
        self.value += 1
        response = message.Message(code=message.CREATED, payload="Value: " + str(self.value))
        return response

# Create a root resource
root = resource.Site()
root.add_resource(['counter'], Counter())

# Start the server
loop = asyncio.get_event_loop()
loop.create_task(aiocoap.Context.create_server_context(root))
loop.run_forever()
```

All tests passed

Test your Translation and Compute its Accuracy

Exercise 6 - test_vocabulary

Complete the function `test_vocabulary` which takes in English embedding matrix X , French embedding matrix Y and the R matrix and returns the accuracy of translations from X to Y by R .

- Iterate over transformed English word embeddings and check if the closest French word vector belongs to French word that is the actual translation.
- Obtain an index of the closest French embedding by using `nearest_neighbor` (with argument `k=1`), and compare it to the index of the English embedding you have just transformed.
- Keep track of the number of times you get the correct translation.
- Calculate accuracy as

`#(correct_predictions)`

```
# UNQ_C10 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
def test_vocabulary(X, Y, R, nearest_neighbor=nearest_neighbor):
    ...

    Input:
        X: a matrix where the rows are the English embeddings.
        Y: a matrix where the rows correspond to the French embeddings.
        R: the transform matrix which translates word embeddings from
            English to French word vector space.

    Output:
        accuracy: for the English to French mapping
    ...

    ### START CODE HERE ###
    # The prediction is X times R (transform English embeddings)
    pred = np.dot(X, R)

    # initialize the number correct to zero
    num_correct = 0

    # Loop through each row in pred (each transformed embedding)
    for i in range(len(pred)):
        # get the index of the nearest neighbor of pred[i]; also pass in the candidates i
        pred_idx = nearest_neighbor(pred[i], Y, k=1)[0]

        # if the index of the nearest neighbor equals the row index 'i'
        if pred_idx == i:
            # increment the number correct by 1
            num_correct += 1

    # accuracy is the number correct divided by the number of rows in 'pred' (also number
    accuracy = num_correct / len(pred)
    ### END CODE HERE ###

    return accuracy
```

Let's see how is your translation mechanism working on the unseen data:

```
X_val Y_val acc matrics(on_fn test_fn embeddings subset on embeddings subset)
# UNQ_C11 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# You do not have to input any code in this cell, but it is relevant to grading, so please
acc = test_vocabulary(X_val, Y_val, R_train) # this might take a minute or two
accuracy on test set is 0.557
```

Expected Output:

0.557

You managed to translate words from one language to another language without ever seeing them with almost 56% accuracy by using some basic linear algebra and learning a mapping of words from one language to another!

```
# Test your function
# unitest(unittest.TestCase, test_vocabulary, test_vocabulary)
```

All tests passed

3 - LSH and Document Search

In this part of the assignment, you will implement a more efficient version of k-nearest neighbors using locality sensitive hashing. You will then apply this to document search.

- Process the tweets and represent each tweet as a vector (represent a document with a vector embedding).
- Use locality sensitive hashing and k nearest neighbors to find tweets that are similar to a given tweet.

```
# get the positive and negative tweets
all_positive_tweets = twitter_samples.strings('positive_tweets.json')
all_negative_tweets = twitter_samples.strings('negative_tweets.json')
```

3.1 - Getting the Document Embeddings

Bag-of-words (BOW) Document Models

Text documents are sequences of words.

- The ordering of words makes a difference. For example, sentences "Apple pie is better than pepperoni pizza." and "Pepperoni pizza is better than apple pie" have opposite meanings due to the word ordering.
- However, for some applications, ignoring the order of words can allow us to train an efficient and still effective model.
- This approach is called Bag-of-words document model.

Document Embeddings

- Document embedding is created by summing up the embeddings of all words in the document.
- If we don't know the embedding of some word, we can ignore that word.

Exercise 7 - get_document_embedding

Complete the `get_document_embedding()` function.

- The function `get_document_embedding()` encodes entire document as a "document" embedding.
- It takes in a document (as a string) and a dictionary, `en_embeddings`
- It processes the document, and looks up the corresponding embedding of each word.
- It then sums them up and returns the sum of all word vectors of that processed tweet.

Hints

```

# UNQ_C12 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
def get_document_embedding(tweet, en_embeddings, process_tweet=process_tweet):
    ...
    Input:
        - tweet: a string
        - en_embeddings: a dictionary of word embeddings
    Output:
        - doc_embedding: sum of all word embeddings in the tweet
    ...
    # Initialize a zero vector of size 300 (assuming embedding size = 300)
    doc_embedding = np.zeros(300)

    ### START CODE HERE ###
    # process the tweet into a list of words
    processed_doc = process_tweet(tweet)

    # sum the embeddings for each word in the processed tweet
    for word in processed_doc:
        if word in en_embeddings:
            doc_embedding += en_embeddings[word]
    ### END CODE HERE ###

    return doc_embedding

```

```

# UNQ_C13 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# You do not have to input any code in this cell, but it is relevant to grading, so please
# testing your function
custom_tweet = "RT @Twitter @chapagain Hello There! Have a great day. :) #good #morning"
tweet_embedding = get_document_embedding(custom_tweet, en_embeddings_subset)
array([-0.00268555, -0.15378189, -0.55761719, -0.07216644, -0.32263184])

```

Expected output:

```
array([-0.00268555, -0.15378189, -0.55761719, -0.07216644, -0.32263184])
```

```

# Test your function
#1-unit-test-test_get_document_embedding(get_document_embedding)
All tests passed

```

Exercise 8 - get_document_vecs

Store all document vectors into a dictionary

Now, let's store all the tweet embeddings into a dictionary. Implement `get_document_vecs()`

```

# UNQ_C14 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
def get_document_vecs(all_docs, en_embeddings, get_document_embedding):
    ...
    Input:
        - all_docs: list of strings - all tweets in our dataset.
        - en_embeddings: dictionary with words as the keys and their embeddings as the values
    Output:
        - document_vec_matrix: matrix of tweet embeddings.
        - ind2Doc_dict: dictionary with indices of tweets in vecs as keys and their embeddings as values
    ...

    # the dictionary's key is an index (integer) that identifies a specific tweet
    # the value is the document embedding for that document
    ind2Doc_dict = {}

    # this list will store the document vectors
    document_vec_1 = []

    for i, doc in enumerate(all_docs):

        ### START CODE HERE ####
        # get the document embedding of the tweet
        doc_embedding = get_document_embedding(doc, en_embeddings)

        # save the document embedding into the ind2Doc_dict at index i
        ind2Doc_dict[i] = doc_embedding

        # append the document embedding to the list of document vectors
        document_vec_1.append(doc_embedding)
        ### END CODE HERE ###

    # convert the list of document vectors into a 2D array (each row is a document vector)
    document_vec_matrix = np.vstack(document_vec_1)

    return document_vec_matrix, ind2Doc_dict

```

```

# UNQ_C15 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# You do not have to input any code in this cell, but it is relevant to grading, so please do it.

print(f"length of dictionary {len(ind2Tweet)}")
print(f"shape of document_vecs {document_vecs.shape}")

length of dictionary 10000
shape of document_vecs (10000, 300)

```

Expected Output

```

length of dictionary 10000
shape of document_vecs (10000, 300)

```

```

# Test your function. This cell may take some seconds to run.
# !/usr/bin/python
# coding: utf-8
# This cell contains the test cases for the function you just implemented.
# It uses the unittest module to check if your function returns the correct output.
# You can run this cell to see if your function is working correctly.
# If you get an error, make sure you have implemented the function correctly.
# If you get a warning, make sure you have followed all the instructions in the assignment.
# If you get a success message, congratulations! Your function is working correctly.
# You can now proceed to the next part of the assignment.
# If you have any questions or need help, feel free to ask in the discussion forum.
# Good luck!

```

All tests passed

3.2 - Looking up the Tweets

Now you have a vector of dimension (m,d) where m is the number of tweets (10,000) and d is the

```
my_tweet = 'i am sad'  
process_tweet(my_tweet)  
tweet_embedding = document_embedding(my_tweet, m, d)
```

```
# UNQ_C16 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)  
# You do not have to input any code in this cell, but it is relevant to grading, so please  
  
# this gives you a similar tweet as your input.  
# this implementation is vectorized...  
idx = np.argmax(cosine_similarity(document_vecs, tweet_embedding))  
print(all_tweets[idx])  
@hanbined sad pray for me :(((
```

Expected Output

```
@hanbined sad pray for me :(((
```

3.3 - Finding the most Similar Tweets with LSH

You will now implement locality sensitive hashing (LSH) to identify the most similar tweet.

- Instead of looking at all 10,000 vectors, you can just search a subset to find its nearest neighbors.

Let's say your data points are plotted like this:



Figure 3

You can divide the vector space into regions and search within one region for nearest neighbors of a given vector.

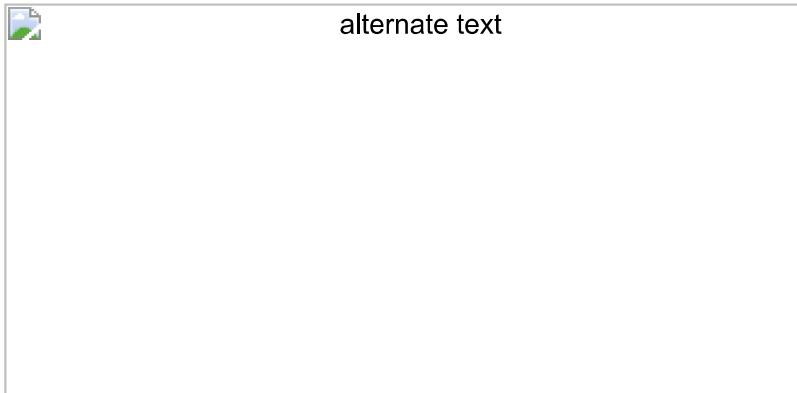


Figure 4

```

N_VECS = len(all_tweets)      # This many vectors.
N_DIMS = len(ind2Tweet[1])     # Vector dimensionality.
# Number of vectors is N_VECS and each has N_DIMS dimensions.

```

Number of vectors is 10000 and each has 300 dimensions.

Choosing the number of planes

- Each plane divides the space to 2^k parts.
- So n planes divide the space into 2^n hash buckets.
- We want to organize 10,000 document vectors into buckets so that every bucket has about ~16 vectors.
- For that we need $\frac{10000}{16} = 625$ buckets.
- We're interested in n , number of planes, so that $2^n = 625$. Now, we can calculate $n = \log_2 625 = 9.29 \approx 10$.

```

# The number of planes. We use log2(625) to have ~16 vectors/bucket.
N_PLANES = 10
# Number of times to repeat the hashing to improve the search.
N_HASHES = 25

```

3.4 - Getting the Hash Number for a Vector

For each vector, we need to get a unique number associated to that vector in order to assign it to a "hash bucket".

Hyperplanes in Vector Spaces

- In 3 -dimensional vector space, the hyperplane is a regular plane. In 2 -dimensional vector space, the hyperplane is a line.
- Generally, the hyperplane is subspace which has dimension 1 lower than the original vector space has.
- A hyperplane is uniquely defined by its normal vector.
- Normal vector n of the plane π is the vector to which all vectors in the plane π are orthogonal (perpendicular in 3 -dimensional case).

Using Hyperplanes to Split the Vector Space

We can use a hyperplane to split the vector space into 2 parts.

- All vectors whose dot product with a plane's normal vector is positive are on one side of the plane.
- All vectors whose dot product with the plane's normal vector is negative are on the other side of the plane.

Encoding Hash Buckets

- For a vector, we can take its dot product with all the planes, then encode this information to assign the vector to a single hash bucket.
- When the vector is pointing to the opposite side of the hyperplane than normal, encode it by 0.
- Otherwise, if the vector is on the same side as the normal vector, encode it by 1.
- If you calculate the dot product with each plane in the same order for every vector, you've encoded each vector's unique hash ID as a binary number, like [0, 1, 1, ..., 0].

Exercise 9 - hash_value_of_vector

We've initialized hash table `hashes` for you. It is list of `N_UNIVERSES` matrices, each describes its own hash table. Each matrix has `N_DIMS` rows and `N_PLANES` columns. Every column of that matrix is a `N_DIMS`-dimensional normal vector for each of `N_PLANES` hyperplanes which are used for creating buckets of the particular hash table.

Exercise: Your task is to complete the function `hash_value_of_vector` which places vector `v` in the correct hash bucket.

- First multiply your vector `v`, with a corresponding plane. This will give you a vector of dimension $(1, \text{N_planes})$.
- You will then convert every element in that vector to 0 or 1.
- You create a hash vector by doing the following: if the element is negative, it becomes a 0, otherwise you change it to a 1.
- You then compute the unique number for the vector by iterating over `N_PLANES`
- Then you multiply 2^i times the corresponding bit (0 or 1).
- You will then store that sum in the variable `hash_value`.

Create the sets of planes

- Create multiple (25) sets of planes (the planes that divide up the region).
- You can think of these as 25 separate ways of dividing up the vector space with a different set of planes.
- Each element of this list contains a matrix with 300 rows (the word vector have 300 dimensions), and 10 columns (there are 10 planes in each "universe").

```
np.random.seed(0)
planes_1 = [np.random.normal(size=(N_DIMS, N_PLANES))
            for _ in range(N_UNIVERSES)]
```

Hints

```

# UNQ_C17 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
def hash_value_of_vector(v, planes):
    """Create a hash for a vector; hash_id says which random hash to use.
    Input:
        - v: vector of tweet. It's dimension is (1, N_DIMS)
        - planes: matrix of dimension (N_DIMS, N_PLANES) - the set of planes that divide
    Output:
        - res: a number which is used as a hash for your vector
    """
    ### START CODE HERE ###
    # calculate the dot product between the vector and the matrix containing the planes
    dot_product = np.dot(v, planes) # shape: (1, N_PLANES)

    # get the sign of the dot product (negative -> -1, positive -> 1)
    sign_of_dot_product = np.sign(dot_product)

    # convert negative to 0, positive (and 0) to 1
    h = sign_of_dot_product >= 0 # shape: (1, N_PLANES), boolean array

    # convert boolean to integers (True -> 1, False -> 0)
    h = h.astype(int).flatten() # shape: (N_PLANES,)

    # initialize the hash value to 0
    hash_value = 0

    n_planes = len(h)
    for i in range(n_planes):
        # increment the hash value by 2^i * h_i
        hash_value += (2 ** i) * h[i]
    ### END CODE HERE ###

    # cast hash_value as an integer
    hash_value = int(hash_value)

    return hash_value

```

```

# UNQ_C18 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# You do not have to input any code in this cell, but it is relevant to grading, so please

np.random.seed(0)
idx = 0
planes = planes_l[idx] # get one 'universe' of planes to test the function
vec = np.random.rand(1, 300)
print(f" The hash value for this vector,",
      f"and the set of planes at index {idx},",
      f"-- planes_l[{idx}] --")

```

The hash value for this vector, and the set of planes at index 0, is 768

Expected Output

The hash value for this vector, and the set of planes at index 0, is 768

```

# Test your function
#1-unitest-test-hash-value-of-vector(hash_value_of_vector)

```

All tests passed

3.5 - Creating a Hash Table

Exercise 10 - make_hash_table

Given that you have a unique number for each vector (or tweet), You now want to create a hash table. You need a hash table, so that given a hash_id, you can quickly look up the corresponding vectors. This allows you to reduce your search by a significant amount of time.



alternate text

We have given you the `make_hash_table` function, which maps the tweet vectors to a bucket and stores the vector there. It returns the `hash_table` and the `id_table`. The `id_table` allows you know which

Hints

```

# UNQ_C19 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)

# This is the code used to create a hash table:
# This function is already implemented for you. Feel free to read over it.

### YOU CANNOT EDIT THIS CELL

def make_hash_table(vecs, planes, hash_value_of_vector=hash_value_of_vector):
    """
    Input:
        - vecs: list of vectors to be hashed.
        - planes: the matrix of planes in a single "universe", with shape (embedding dimension, number of planes)
    Output:
        - hash_table: dictionary - keys are hashes, values are lists of vectors (hash buckets)
        - id_table: dictionary - keys are hashes, values are list of vectors id's
                      (it's used to know which tweet corresponds to the hashed vector)
    """
    # number of planes is the number of columns in the planes matrix
    num_of_planes = planes.shape[1]

    # number of buckets is 2^(number of planes)
    # ALTERNATIVE SOLUTION COMMENT:
    # num_buckets = pow(2, num_of_planes)
    num_buckets = 2**num_of_planes

    # create the hash table as a dictionary.
    # Keys are integers (0,1,2.. number of buckets)
    # Values are empty lists
    hash_table = {i: [] for i in range(num_buckets)}

    # create the id table as a dictionary.
    # Keys are integers (0,1,2... number of buckets)
    # Values are empty lists
    id_table = {i: [] for i in range(num_buckets)}

    # for each vector in 'vecs'
    for i, v in enumerate(vecs):
        # calculate the hash value for the vector
        h = hash_value_of_vector(v, planes)

        # store the vector into hash_table at key h,
        # by appending the vector v to the list at key h
        hash_table[h].append(v) # @REPLACE None

        # store the vector's index 'i' (each document is given a unique integer 0,1,2...)
        # the key is the h, and the 'i' is appended to the list at key h
        id_table[h].append(i) # @REPLACE None

```

```
# UNQ_C20 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# You do not have to input any code in this cell, but it is relevant to grading, so please
planes = planes_l[0] # get one 'universe' of planes to test the function
tmp_hash_table, tmp_id_table = make_hash_table(document_vecs, planes)

print(f"The hash table at key 0 has {len(tmp_hash_table[0])} document vectors")
print(f"The id table at key 0 has {len(tmp_id_table[0])} document indices")
# print(f"The first 5 document indices stored at key 0 of id table are {tmp_id_table[0][0:5]}
```

The hash table at key 0 has 3 document vectors
The id table at key 0 has 3 document indices
The first 5 document indices stored at key 0 of id table are [3276, 3281, 3282]

Expected output

The hash table at key 0 has 3 document vectors
The id table at key 0 has 3 document indices
The first 5 document indices stored at key 0 of id table are [3276, 3281, 3282]

```
# Test your function
# unitest test_make_hash_table(make_hash_table)
```

All tests passed

3.6 - Creating all Hash Tables

You can now hash your vectors and store them in a hash table that would allow you to quickly look up and search for similar vectors. Run the cell below to create the hashes. By doing so, you end up having several tables which have all the vectors. Given a vector, you then identify the buckets in all the tables. You can then iterate over the buckets and consider much fewer vectors. The more tables you use, the more accurate your lookup will be, but also the longer it will take.

```

# Creating the hashtables
def create_hash_id_tables(n_universes):
    hash_tables = []
    id_tables = []
    for universe_id in range(n_universes): # there are 25 hashes
        print('working on hash universe #:', universe_id)
        planes = planes_l[universe_id]
        hash_table, id_table = make_hash_table(document_vecs, planes)
        hash_tables.append(hash_table)
        id_tables.append(id_table)

    return hash_tables, id_tables

```


-----\N_UNIVERSES\-----

```

working on hash universe #: 0
working on hash universe #: 1
working on hash universe #: 2
working on hash universe #: 3
working on hash universe #: 4
working on hash universe #: 5
working on hash universe #: 6
working on hash universe #: 7
working on hash universe #: 8
working on hash universe #: 9
working on hash universe #: 10
working on hash universe #: 11
working on hash universe #: 12
working on hash universe #: 13
working on hash universe #: 14
working on hash universe #: 15
working on hash universe #: 16
working on hash universe #: 17
working on hash universe #: 18
working on hash universe #: 19
working on hash universe #: 20
working on hash universe #: 21
working on hash universe #: 22
working on hash universe #: 23
working on hash universe #: 24

```

Approximate K-NN

Exercise 11 - approximate_knn

Implement approximate K nearest neighbors using locality sensitive hashing, to search for documents that are similar to a given document at the index `doc_id`.

Inputs

- `doc_id` is the index into the document list `all_tweets`.
- `v` is the document vector for the tweet in `all_tweets` at index `doc_id`.
- `planes_l` is the list of planes (the global variable created earlier).
- `k` is the number of nearest neighbors to search for.
- `num_universes_to_use` : to save time, we can use fewer than the total number of available universes. By default, it's set to `N_UNIVERSES`, which is \$25\$ for this assignment.
- `hash_tables` : list with hash tables for each universe.
- `id_tables` : list with id tables for each universe.

The `approximate_knn` function finds a subset of candidate vectors that are in the same "hash bucket" as the input vector 'v'. Then it performs the usual k-nearest neighbors search on this subset (instead of

Hints

```
def approximate_knn(doc_id, v, planes_l, hash_tables, id_tables, k=1, num_universes_to_use=1):
    """Search for k-NN using hashes.

    vecs_to_consider_l = list()
    ids_to_consider_l = list()
    ids_to_consider_set = set()

    for universe_id in range(num_universes_to_use):
        planes = planes_l[universe_id]
        hash_value = hash_value_of_vector(v, planes)
        hash_table = hash_tables[universe_id]
        document_vectors_l = hash_table[hash_value]
        id_table = id_tables[universe_id]
        new_ids_to_consider = id_table[hash_value]

        # Loop through the subset of document vectors to consider
        for i, new_id in enumerate(new_ids_to_consider):
            if doc_id == new_id:
                continue

            if new_id not in ids_to_consider_set:
                # Get the vector at this index
                document_vector_at_i = document_vectors_l[i]

                # Add vector to the candidate list
                vecs_to_consider_l.append(document_vector_at_i)

                # Add the corresponding ID
                ids_to_consider_l.append(new_id)

                # Add the ID to the set to prevent duplicates
                ids_to_consider_set.add(new_id)

    print("Fast considering %dvecs" % len(vecs_to_consider_l))

    vecs_to_consider_arr = np.array(vecs_to_consider_l)

    # Run nearest neighbor search on reduced candidate set
    nearest_neighbor_idx_l = nearest_neighbor(v, vecs_to_consider_arr, k=k)

    nearest_neighbor_ids = [ids_to_consider_l[idx]
                           for idx in nearest_neighbor_idx_l]
```

```
#document_vecs, ind2Tweet
doc_id = 0
doc_to_search = all_tweets[doc_id]
vec_to_search = document_vecs[doc_id]
```

```
# UNQ_C22 (UNIQUE CELL IDENTIFIER, DO NOT EDIT)
# You do not have to input any code in this cell, but it is relevant to grading, so please
```

```
# Sample
```

```
nearest_neighbor_ids = approximate_knn(
```

```
Fast considering 77 vecs
```

```
print(f"Nearest neighbors for document {doc_id}")
print(f"Document contents: {doc_to_search}")
print("")  
  
for neighbor_id in nearest_neighbor_ids:
    print(f"Nearest neighbor at document id {neighbor_id}")
    print(f"document contents: {all_tweets[neighbor_id]}")
```

```
Nearest neighbors for document 0
```

```
Document contents: #FollowFriday @France_Inte @PKuchly57 @Milipol_Paris fo  
r being top engaged members in my community this week :)
```

```
Nearest neighbor at document id 51
```

```
document contents: #FollowFriday @France_Espana @reglisse_menthe @CCI_inte  
r for being top engaged members in my community this week :)
```

```
Nearest neighbor at document id 2478
```

```
document contents: #ShareTheLove @oymgroupt @musicartisthere for being top  
HighValue members this week :) @nataliavas http://t.co/IWSDMtcayt
```

```
Nearest neighbor at document id 105
```

```
document contents: #FollowFriday @straz_das @DCarsonCPA @GH813600 for bein  
g top engaged members in my community this week :)
```

```
# Test your function
```

```
!jupyter test_approximate_knn(approximate_knn, hash_table, id_table)
```

```
Fast considering 77 vecs
```

```
Fast considering 153 vecs
```

```
All tests passed
```

4 Conclusion

Congratulations - Now you can look up vectors that are similar to the encoding of your tweet using LSH!

**Natural Language Processing
with Classification and Vector...**

- ✓ Video • 5 min
- ✓ Locality sensitive hashing
Reading • 10 min
- ✓ Multiple Planes
Video • 3 min
- ✓ Multiple Planes
Reading • 10 min
- ✓ Hash tables
Lab • 1h
- ✓ Approximate nearest neighbors
Video • 3 min
- ✓ Approximate nearest neighbors
Reading • 10 min
- ✓ Searching documents
Video • 1 min
- ✓ Searching documents
Reading • 10 min
- ✓ Week Conclusion
Video • 50 sec
- ✓ Lecture Notes W4
Reading • 1 min
- ✓ Hashing and Machine Translation
Practice Assignment • Grade: 80%
 - [IMPORTANT] Reminder about end of access
to Lab Notebooks
 - Reading • 2 min
- ✓ Word Translation
Programming Assignment • Grade: 100%
- Acknowledgements
Reading • 10 min
- Bibliography
Reading • 10 min

1/3 learning items



5

Programming Assignment: Word Translation

Passed • 110/110 points

Deadline Pass this assignment by Feb 20, 11:59 PM +06

[Launch Notebook](#)[Instructions](#) [My submissions](#)

Date	Score	Passed
▼ February 17, 2026 11:20 AM +06	110/110	Yes
Test_get_matrices	10/10	Show grader output
test_compute_loss	10/10	Show grader output
test_compute_gradient	10/10	Show grader output
test_align_embeddings	10/10	Show grader output
test_nearest_neighbor	10/10	Show grader output
unittest_test_vocabulary	10/10	Show grader output
test_get_document_embedding	10/10	Show grader output
test_get_document_vecs	10/10	Show grader output
test_hash_value_of_vector	10/10	Show grader output
test_make_hash_table	10/10	Show grader output

jupyter C1_W4_Assignment Last Checkpoint: 20 minutes ago (autosaved)

File Edit View Insert Cell Kernel Help

Trusted Python 3



Assignment 4 - Naive Machine Translation and LSH

You will now implement your first machine translation system and then you will see how locality sensitive hashing works. Let's get started by importing the required functions!

If you are running this notebook in your local computer, don't forget to download the twitter samples and stopwords from nltk.

```
nltk.download('stopwords')
nltk.download('twitter_samples')
```

Important Note on Submission to the AutoGrader

Before submitting your assignment to the AutoGrader, please make sure you are not doing the following:

1. You have not added any `extra print` statement(s) in the assignment.
2. You have not added any `extra code cell(s)` in the assignment.
3. You have not changed any of the function parameters.
4. You are not using any global variables inside your graded exercises. Unless specifically instructed to do so, please refrain from it and use the local variables instead.
5. You are not changing the assignment code where it is not required, like creating `extra` variables.

If you do any of the following, you will get something like, Grader Error: Grader feedback not found (or similarly unexpected) error upon submitting your assignment. Before asking for help/debugging the errors in your assignment, check for these first. If this is the case, and you don't remember the changes you have made, you can get a fresh copy of the assignment by following these [instructions](#).

Table of Contents

- [1_ The Word Embeddings Data for English and French Words](#)
 - [1.1 Generate Embedding and Transform Matrices](#)
 - [Exercise 1 - get_matrices \(UNQ_C1\)](#)
- [2_ Translations](#)
 - [2.1 - Translation as Linear Transformation of Embeddings](#)
 - [Exercise 2 - compute_loss \(UNQ_C3\)](#)
 - [Exercise 3 - compute_gradient \(UNQ_C4\)](#)
 - [Exercise 4 - align_embeddings \(UNQ_C5\)](#)
 - [2.2 - Testing the Translation](#)
 - [Exercise 5 - nearest_neighbor \(UNQ_C8\)](#)

Grades

Passed • Grade Received: 100%

Submissions

February 17, 2026 11:20 AM +06 (Highest Grade)

▼ [Test_get_matrices](#)

10/10 points earned

▼ [test_compute_loss](#)

10/10 points earned

▼ [test_compute_gradient](#)

10/10 points earned

▼ [test_align_embeddings](#)

10/10 points earned

▼ [test_nearest_neighbor](#)

10/10 points earned

▼ [unittest_test_vocabulary](#)

10/10 points earned