BDS 761 – Practice Exam I

Name: ____

Time: 120 minutes

- If I cannot find or understand your writing then I cannot give you credit.
- If you do not show your work you may not get full credit.
- If you get the correct answer but add incorrect information you may not get full credit.
- *If your answer is too vague, or not the best answer, you also may not get full credit.*
- If a question asks for pseudocode, this means a simplified programming language you can invent yourself with an obvious syntax
- **CLOSED BOOK. CLOSED NOTES. NO WRITTEN SUPPORT MATERIAL ALLOWED. NO COMPUTERS. NO CELL/SMART PHONES.**

1. (5 points) BONUS.

- Write your full name on this page plus at the top of every scratch/loose page you use (if you remove the staple your name must be on every separate page; it is ok to remove the staple).
- Do not write on the backs of any pages.
- Do not fold any pages.

2. (5 points) What are the two alternative types of string matching and how do they differ? Both functionally as well as in terms of pros and cons.

Exact matching

- Pros: fast, easy to code (string1==string2)
- Cons: mismatch caused by spelling errors, capitalization, spelling differences (color vs colour)

Inexact matching: using regex etc

- Pros: can handle spelling errors etc (with appropriate regex design)
- Cons: can be slow, can get very complex and hard to maintain

Also remember string distance metrics we learned

  - Hamming – only counts substitutions
  - edit distance a.k.a. Levenshtein) – counts substitutions, insertions, deletions

To perform exact matching with edit distance we check for distance = 0 (same for either distance metric)

3. (15 points) Describe how you could perform a comparison between two documents that is lower (better) when they contain similar words, and higher when they don't. Give steps with as much detail as possible using code or pseudocode. We would like singular, plural, and other versions of a word to be treated as the same word.

   Assuming we have two input strings, doc1 and doc2 containing the text.

```
# Step1: split into words using tokenization. Could also use doc1.split(), but must handle punctuation

List1 = tokenize(doc1) # result is a list of words and punctuation tokens.
List2 = tokenize(doc2)

# Step2: extract word roots using lemmatization

List1 = lemmatize(List1) # assume it works on list, else use loop over each words
List2 = lemmatize(List2)

# Step3: remove repeats so they don't bias result

List1 = set(List1)
List2 = set(List2)

# Step 4: count matches

count=0
For word in List1:
    if word in List2:
            count=count+1
```

4. (5 points) Explain the difference between dynamic programming and memoization. Which approach requires you to rewrite the algorithm? Suppose you have a slow function that can be accelerated by these methods. In what situations would you *not* want to use the accelerated version, even though it could provide a speed benefit?

   With dynamic programming, we rewrite the algorithm to build up the function outputs from the bottom.

   With memorization, we use the original function and check each function call against prior calls to avoid repeated calculations. We don't need to rewrite the algorithm.

   As these methods trade speed for memory, we may not want to use them if there is insufficient memory available.

5. (3 points) Give the mathematical equation for a dot product for vectors with N elements each

   (3 points) How many computations will this require?

   The order of N. Or to be exact, N multiplications and N additions

(10 points) Suppose we know the vectors each have at most *k* elements which are nonzero. We further know the indices of these nonzero elements along with their values. Describe an efficient program for computing the dot product much faster than the above, by using this information. How many computations will it take?

Assume we have sparse vectors such as x = [0,0,0,0,0,0,12,0,0,0,0,5,0,0,0,0…]

This can be described efficiently with a pair of lists such as: x_indices = [6,11,…] and x_nzvals = [12,5,…] where the first is index locations and the second is nonzero values. We might store this same info in a dict such as x_sparse = {6:12,11:5,…}, where the key gives the index of the nonzero value

To take the inner product assume we have two vectors described with dicts x_sparse1 and x_sparse2

```
result = 0
for key in x_sparse1: # loops over keys, indices of nonzero values
    if key in x_sparse2: # same key found in other vector also
        result=result+x_sparse1[key]*x_sparse2[key]
```

since there are k nonzero values, this will only take *k* multiplies and additions. Note for a very sparse vector *k*<<*N* so this could be much faster than *N* multiplies and additions.

6. (3 points) Give the mathematical equation for matrix-vector multiplication

7. (5 points) Give the mathematical equation for vector-matrix multiplication, i.e., for multiplying a matrix by a vector from the left side instead of the right side.

8. (4 points) Give the general mathematical equation for matrix-matrix multiplication

9. (10 points) Describe a way to store a m by n matrix in python. Give code for accessing a specific matrix element A_{ij} from this data structure.

10. (10 points) Give code for a function that takes a single input *N* and returns a *N* by *N* identity matrix.

```
# using numpy
I = numpy.eye(N)

# using numpy but not eye function…
I = numpy.zeros((N,N))

For k in range(0,N):
    I[k,k] = 1

# without numpy is messy due to variable N requirement
# to keep things simple, let's make a zeros matrix and reuse above loop

def make_row_of_zeros(N):
    row = []
    for k in range(0,N):
        row.append(0)
    return row

def make_square_zeros_matrix(N):
    matrix = []
    for k in range(0,N):
        matrix.append(make_row_of_zeros(N))
    return matrix

# now we can use prior code to loop over zeros matrix and make diagonal ones

I = make_square_zeros_matrix(N):

for k in range(0,N):
    I[k,k] = 1
```

11. (5 points) Give two reasons why a python set is not a good choice for describing vectors. What built-in data structure is best?

Reasons: elements must be unique, order may not be preserved

Best choice: List

12. (10 points) Given two vectors with *N* elements each, we could use a dot product or a Euclidean distance to compare their similarity. Explain mathematically why these two approaches are closely related. What will be the differences between these two comparison values?

    The Euclidean distance can be expressed in terms of dot products, such as

    ||x|| = sqrt(dot(x,x))

    ||x-y|| = sqrt(dot(x,x)+dot(y,y)-2*dot(x,y))

    The dot product of x and y is one of the terms. As the dot product gets larger this term becomes more negative, and so the distance gets smaller. Also they differ due to the square root and dot(x,x)+dot(y,y) terms.

    Note that if our interest is in matching the vectors we probably don't care about the size of dot(x,x) or dot(y,y), so just using dot(x,y) may make more sense.

13. (15 points) List (and describe each mathematically) all the possibly ways you can think of to "multiply" two vectors with *N* elements each (including with broadcasting) using numpy. Including different order and possible transposes.

Given column vectors x and y,

1. Inner product x.T@y = y.T@x
2. Outer product x@y.T
3. Outer product y@x.T = transpose of #2
4. Elementwise or Hadamard product x*y

Note
* for this special case (two vectors) the outer products equal the broadcast products, i.e. x@y.T = x*y.T
* so the Hadamard product x*y = np.multiply(x,y)
* The matrix product can be variously written as e.g., x.T@y = np.matmul(x.T,y) = np.dot(x.T,y)