Introduction

Provides an overview of the project and a short dicsussion on the pertinent literature

The project number 8 is composed of two parts. One of them is perturb appropriately a dataset of documents, the second one rebuilding original documents.

I focused on the first part.

The literature in merit [1] suggests the most common errors observed and a methodology to identify such errors centerd around the Needleman-Wunsch algorithm for identifying the best alignment of two strings. The original NW implementation was in the study of genomics and DNA sequences. [2]

This algorithm revolves around the idea of inserting characters '-' in smart places in order to obtain the best alignment of two strings.

The literature focuses more on finding what kind of errors OCR produce rather than claverly reproduce them, hence making this work innovative in the sense that it allows to reprodce errors made by an optical reader into a larger dataset. The approach relies solely on (true_text, read_text) pairs.

Research question and methodology

Provides a clear statement on the goals of the projec, an overview of the proposed approach, and a formal definition of the problem

I used a supervised method to learn what are the mistakes an OCR makes and reproduce them on any text.

These mistakes depend on

1. The OCR implementation
2. The dataset

Hence, given that Perturbers are made to replicate the mistakes they see in a ground truth, the kind of mistakes depend on the OCR thecnology and the dataset the Perturber was trained on. More, they apply a probabiistic approach (as it has no access to the original images) to replicate these.

The ground truth required is in the form (true_text, read_text). The goal is to identify the errors made and replicate them effectively on a fresh dataset. The idea is tha a well trained perturber will be able to predict the errors it will make on an entire dataset by just a part of it.

It of course requires data to work and training on one dataset can give different results.

The approach is based on the structure of nchar (mimicking ngrams) (unichars and bichars in particular). The idea is to study what an nchar is transformed to by the scanner.

This very approach can be used in translation

Fitting and Transforming

Given a list of true documents x and a matching list of read documents y, fitting goes as follows:

```
Function Fit(x, y)
    ----
    Inputs:
        x, y : lists of documents
    Outputs:
        perturbations : dictionary of dictionaries
        p : map from keys(perturbations) to interval [0,1]
```

```
        s : map from keys(perturbations) to integers
    ----

    perturbations, p, s <- empty dictionary
    FOR ALL unichar uc in true
        perturbations[uc] <- empty dictionary
    FOR ALL bichars bc in true
        perturbations[bc] <- empty dictionary

    FOR i = 0 to N
        true, scan <- Align(x[i], y[i]) // true and scan are of equal length

        FOR j = 0 to length(true)
            perturbations[true[j]][scan[j]] += 1 // add a perturbation of unichar true[j]
        FOR j = 0 to length(true)-1
            perturbations[true[j, j+1]][scan[j, j+1]] += 1 // add a perturbation of bichar true[j, j+1]

    FOR ALL nchar in keys(perturbations)
        p[nchar] = //Probability that nchar is not perturbed to itself
        s[nchar] = //Number of times nchar was observed in x

    return perturbations, p, s
End Function

Function Align(t1, t2)
    ----
    Inputs:
        t1, t2 : strings
    Outputs:
        al1, al2 : strings of equal length
    ----

     al1, al2 <- NeedlemanWunsch(t1, t2)

     return al1, al2
End Function

Function ClusterFit(x, y)
    ----
    Inputs:
        x, y : lists of strings
        Vectorize : maps a dictionary to an array
        Dictionarize : inverse of Vectorize
        nch2idx : injective map from nchars to integers
        Cluster : clustering method
    Outputs:
        perturbations: dictionary
    ----

    perturbations, _, _ <- Fit(x, y)
    D = zeros(length(keys(perturbations)), length(keys(perturbations)))

    // Build a distance matrix of keys of perturbations
    FOR (nc1, nc2) in product(keys(perturbations), keys(perturbations))
        id1, id2 <- nch2idx(nc1), nch2idx(nc2)
        v1, v2 <- Vectorize(nc1), Vectorize(nc2)
        d[id1, id2] <- Continuous_overlapping_distance(v1, v2)

    // Cluster the keys of perturbations using D
```

```
    clusters <- Cluster(D)

    // Modify perturbations
    FOR C in clusters
        centroid = Centroid(C)
        FOR nchar in C
            v <- Vectorize(C)
            s <- Sum(v) // keep track of the sum of v
            d <- Continuous_overlapping_distance(v, centroid)
            v = d*v + (1-d)*centroid // move v towards the centroid
            perturbations[nchar] = Dictionarize(v) // rebuild perturbations[nchar]
End Function

Function Continuous_overlapping_distance(v1, v2)
    ----
    Inputs:
        v1, v2 : numerical arrays of equal length
    Outputs:
        d : distance
    ----

    Rescale v1, v2 to sum 1
    mins = [min(v1[t], v2[t]) FOR t=0 to length(v1)]
    return 1 - Mean(mins)
End Function

Function Vectorize(d)
    ----
    Inputs:
        d : dictionary with string keys
        nch2idx : injective map from nchars to integers
    Outputs:
        v : array
    ----

    FOR key, value in d
        id <- nch2idx(key)
        v[id] <- value

    return v
End Function

Function Transform(text)
    ----
    Inputs:
        text : string
        perturbations, p, s : output of a call to Fit on some training data
        beta : float between 0 and 1
        num_rounds : integer >= 1
    Outputs:
        out : string
    ----

    perturbed_list <- empty list
    perturbed_text <- empty string

    FOR round in num_rounds
        splits <- Split(text, p)
        FOR nchar in splits
            APPEND Perturb(nchar, beta, perturbations) TO perturbed_list
    perturbed_text = CONCATENATE perturbed_list
```

```
End Function

Function Split(text, p)
    ----
    Inputs:
        text : string
        p : // as in Transform
    Outputs:
        splits : list of nchars
    ----

    // Splits `text` in nchars to maximize the probability of at least one
    // of these nchars to be perturbed
    // An nchar is perturbed if Perturb(nchar, 1, perturbations) != nchar

End Function

Function Perturb(nchar, beta, perturbations)
    ----
    Inputs:
        nchar, beta, perturbations : // as in Transform
    Outputs:
        pert : perturbed `nchar`
    ----

    // With probability `beta`, map `nchar` to `pert` according to the distribution
    // given by `perturbations[nchar]`
    // With probability 1-`beta`, return `nchar`

End Function
```

Toy example of Fit and Transform

`beta` is set to 1 and so is `num_rounds`

```
x1 = 'unimi'
y1 = 'un Im i'

true, scan = Align(x1, y1)
// un-im-i
// un Im i

perturbations, p, s = Fit(true, scan)
// perturbations
/*{
    'i': {' I': 1, ' i': 1, 'I': 1, 'i': 1},
    'im': {'Im': 1},
    'm': {'m': 1, 'm ': 1},
    'n': {'n': 1, 'n ': 1},
    'u': {'u': 1},
    'un': {'un': 1}
}*/
// p
// {'i': 0.75, 'im': 1.0, 'm': 0.5, 'n': 0.5, 'u': 0.0, 'un': 0.0}
// s
// {'i': 4, 'im': 1, 'm': 2, 'n': 2, 'u': 1, 'un': 1}
```

```
fresh = 'umido'

ls = Split(fresh)
// ['u', 'm', 'i', 'd', 'o']

l <- empty list
FOR _=0 to 4
    APPEND Transform(fresh) TO l
// ['umIdo', 'um  ido', 'umIdo', 'um  ido', 'um Ido']

/*
probability of perturbing fresh when split as _ is:
    ['u', 'm', 'i', 'd', 'o'] : 0.875
    ['um', 'id', 'o'] : 0
    ['um', 'i', 'd', 'o'] : 0.75
*/
```

Transforming

Perturbating a text is not simple.

Perturbating character by character can work but actually no, it does not catch some properties of 2chars

But how to choose whether to use a 1char or a 2char? Computing all of the possibilities requires F(n)computasion, where n is length of string. Instead, I used a greedy algorithm maximizing at each step the probability of having a misinterpretation.

If wrks like this on trichars:

take the first 3 characters

compute all of the possible splittings

find the easiest-to-misiinterpret split

add it to a list

then perturb everything

Transformation

Finally transform the list of splitted, tokenized (in uni-bichars) text.

Each element of the list is mapped randomly (distribution given by what the algorithm was fitted on) to another uni or bichar.

Then the results are recombined in a singel string

Metrics for evaluation

- A perturber is trained on a training set. It perturbs a test set. The transformations on the test set are compared with the ground truh ones (having two other perturbers learning the transformations from the test set)

Experimental results

Provides an overview of the dataset used for experiments, the metrics used for evaluating performances, and the experimental methodology. Presents experimental results as plots and/or tables

Concluding remarks

Provides a critical discussion on the experimental results and some ideas for future work

Ideas:

- Explore the markov chain process of perturbation
- explore the possibility of using a similarity matrix of nchars when perturbing
    - In order to perturb never seen before nchars

References

1. https://link.springer.com/chapter/10.1007/978-3-030-45442-5_13#Bib1
2. https://gist.github.com/slowkow/06c6dba9180d013dfd82bec217d22eb5