Federico Matteucci 955753

Information Retrieval Project #8 part 1

# Supervised replication of OCR errors

## Introduction

The motivation behind this paper is to develop an algorithmic procedure that is able to replicate the errors produced by an **OCR** (Optical Character Reader) on fresh documents, effectively allowing researchers to expand a ground truth of *expected-extracted* pairs of documents in a data-driven automatic way.

An analysis in this regard [1] suggests a metodology to simulate such errors making use of *extracted-expected* pairs of documents to learn the mistakes: they first aligned two strings with the Needleman-Wunsch algorithm [2], then generated a character exchange list that is finally applied to perturb fresh documents.

However, the final goal in [1] is to categorize **OCR** errors in order to develop better techniques to fix them (stemming among the others) but not to build procedures to effectively replicate observed errors on new documents.

The novelty of the approach I propose is that of using the true transcription of a document as input and perturb it in a way that is the most similar to how an **OCR** would do, translating an *expected* document into a string that is the most similar to an *extracted* one.

## Research question and methodology

It is now necessary to introduce some basic terminology: an **nchar** is a sequence of n *consecutive* characters in a string; given a scan of a textual document, the **expected** string is the true content of that document, the **extracted** string is the text read from an **OCR**. Two nchars are said to be **corresponding** in an *expected-extracted* pair if they start at the same index.

Goal of this project is to build a class of predictors, named **Perturber**, that is able to learn from a corpus of *expected-extracted* pairs and to replicate the misreadings it observes to a fresh *expected* string.

The main stages of a Perturber's life are (as for every other machine learning predictor) **Fit** and **Transform**.

Examples and pseudo-python for the algorithms can be found in the Appendix.

### Fit

The basic fitting procedure is devoted to the creation of a nested index called `perturbations` such that given an **nchar** `nch1`(in the implementation, my Perturbers are only able to learn how to misread unichars and bichars) , `perturbations[nch1]` associates to an nchar `nch2` the number of times `nch1`was misread as `nch2`.

The algorithm works as follows:

1. **Access** an *expected-extracted* pair
2. **Align** *expected* and *extracted* using Needleman-Wunsch [3]
3. **Store** corresponding nchars in `perturbations` (even when the corresponding nchars are equal)

4. **Compute** for each nchar `nch` the probability of perturbations (stored in a dictionary `p` - *probability*), that is the number of times `nch` does not correspond to itself in the corpus divided by the total number of times `nch` is observed in the *expected* corpus (this latter stores in a dictionary `s` - *support*)

Note that given the nature of `perturbations`, it suffices an nchar-to-integer index to vectorize it. Also, if `perturbations[nch]` is rescaled to have sum 1, `perturbations[nch][nch1]` is the estimated probability of `nch` being misread as `nch1`.

## Transform

The transformation procedure revolves around `perturbations`.

When a Perturber transforms a string `text`, it uses `p` and `perturbations` to:

1. **Split** `text` in a list of unichars and bichars
2. **Perturb** each nchar individually using the probability distribution given by `perturbations`
3. **Reconstruct** the perturbed string by concatenating the perturbed nchars

### Split

Splitting is done in such a way to maximize the probability of the resulting list of nchars of having at least one of the nchars perturbed. This is not a trivial problem as the number of splits grows exponentially in the length of the string (see Appendix). In order to address this issue, I developped a greedy algorithm (**Split**) to retrieve a locally optimal split, local in the sense that it scans the string from left to right and for each trichar it finds it splits it in the optimal way before proceeding to the next one.

### Hyperparameters

**Perturbers** have two hyperparameters:

- $\beta$ : the probability of entering the **Perturb** for an nchar
    - Note that even if an nchar enters the **Perturb** stage, there is no guarantee
- **num_rounds** : the number of Transform iterations the Perturber does on a text

Anyway, these parameters are not intended to be tuned in the training procedure but rather to give the user the possibility to select the number of errors wanted.

## Extensions

As such a predictor is most certain to **overfit** (as it can only perturb characters it has already seen in ways that it has already seen), some expedients are used to introduce some noise in the fitting process to reduce the overfitting.

The first one is to drop from `perturbations` any nchar that was observed less than a threshold `th` times (introducing a new hyperparameter for Perturbers). This means that words that were observed too few times are not considered in the transformation and are always perturbed to themselves.

The second one is to make use of a **clustering** algorithm on the vectorized `perturbations` and move the vectors associated to each nchar towards the cluster centroids. The dissimilarity measure used to cluster is the **Continuous Overlapping Distance** (based on the Continuous Overlapping Index defined in [4] - details in the Appendix).

Let $v$ be a vector associated to an nchar, $c$ its cluster centroid, then $v$ is replaced by

$$v' = COD(v,c) \cdot v + (1 - COD(v,c)) \cdot c \tag{1}$$

Finally, `perturbations[nchar]` is redefined based on $v'$.

## Clustering

In this paper I present 3 clustering approaches that seemed the most fit to the purpose of injecting noise in the perturbations dictionary. First of all, there are some points to make: the vectors are **sparse** (as even the most likely-to-be-misread nchars have dozens possible misreadings out of hundreds) and are therefore likely to be very distant from each other.

Hence, the nchar vectors are best clustered in an **unknown number** of **small** (possibly even two elements), **well separated** clusters with lots of **noise** vectors that do not fit inside any of the clusters. Therefore, the clustering algotirhms I chose to use are **Hierarchical** (as it allows to select the number of clusters), **DBSCAN** (as it can detect noisy points) and a **Graph Clustering** algorithm.

The fitting score of the clustering algorithms is the *silhouette score* [5], as it measures the distance of a point from its cluster and compares it with its distance with the other clusters. A clustering achieves a greater score the more separated are the points from the clusters they do not belong to.

Both canonical algorithms require an hyperparameter to be tuned, as follows:

- **Hierarchical Clustering** : tune the *number of clusters*
- **DBSCAN** :  the *minimum number of samples* that can consitute a cluster is set to $2$, the *maximum acceptable distance* is tuned

The selected hyperparameter is the greatest value to achieve a score **one standard deviation below the maximum**, as it ensures either more or bigger clusters.

### Graph Clustering

As for **Graph Clustering**, it can be considered a variation on **DBSCAN** , with the main differences being that the former is completely *deterministic* and *faster* (on the training dataset, with same hyperparameter grid and same scoring method, **DBSCAN** was tuned in $54$ seconds and **Graph Clustering** in $18$).

The hyperparameters to be tuned are $\varepsilon$, the maximum acceptable distance of a vector from its cluster (without the vector itself) and the minimum cardinality of a cluster $mincard$.

The **training procedure** of a Graph Clustering on a set of arrays is:

- Build an **undirected weighted graph**
  - Each node is a vector
  - Each edge is weighted with the distance of its two extremes
- **Delete** all edges with a weight greater than $\varepsilon$
- **Compute** the connected components of the resulting subgraph
- **Return** every connected component with more than $mincard$ nodes in it

Regarding the tuning procedure, as for **DBSCAN** $mincard$ was set to 2 and a selection was run on $\varepsilon$. To explore other possibilities, two tuning targets were considered: one based on the aforementioned silhouette score and the other aiming to maximize the number of clusters found.

## Performance evaluation

The performance of a Perturber depends on how well it can predict the misreadings the **OCR** scanner it was trained on will do on a fresh dataset. Therefore, the error rate of a Perturber is measured as the average **COD** between the *learned perturbations* and *true perturbations*.

Operatively, this requires a fresh dataset of *expected-extracted* pairs. *Expected* strings are transformed with the Perturber to get a dataset of *expected-extracted-transformed* triplets. Now it is a matter of learning what perturbations are done by the **OCR** (comparing *expected-extracted* documents) and what perturbations are done by the **Perturber** (comparing *expected-transformed* documents). This can be easily done by fitting a new Perturber on each pair. Finally, the distance between two resulting `perturbations` is the average distance (**OCD**) of their entries.

Another measure of interest is that of the expected number of mistakes that a Perturber would do. This **expected number of mistakes** is calculated assuming that all perturbations occur independently and it depends on the hyperparameters of the Perturber.

## Experimental results

### Dataset

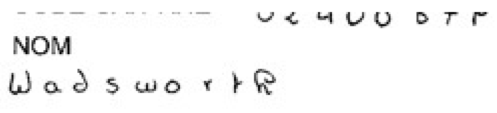
The dataset I used is the training dataset is the Landlord/handwriting recognition available on Kaggle [6], which provides thousands of photos of handwritten *capitalized* names and surnames labelled with the true name (*expected*). The images are extremely noisy and challenging for an **OCR** to correctly read. as there often appear cropped sections of other texts and the quality is not always good.

As **OCR** I used a basic pytesseract with some minor improvements in terms of recognized text post-processing (leading to *Clean interpretations*). The images where then labelled with their OCR readings.

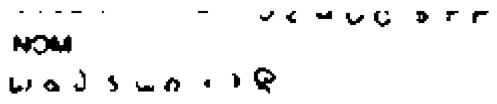
I also tested some possible image improvements (as suggested in [7]) but with little to no success.

The images below show an image, the *expected* and *extracted* intepretations with and without image post-processing.

```
Image id :  70
--------------------
```



```
Clean interpretation :  WODSWORH®
--------------------
True text :  WADSWORRE
--------------------
NW comparison :
 WODSWORH®
 WADSWORRE
--------------------
```



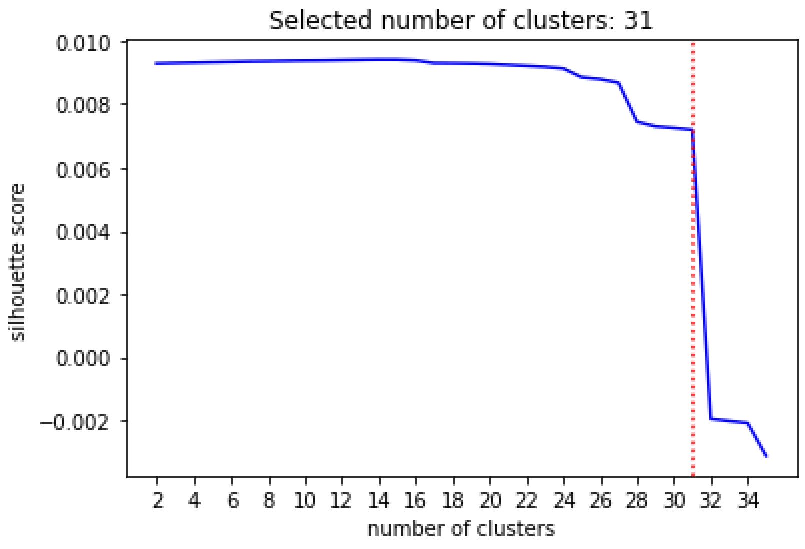
```
Clean interpretation :  SU U¥MN EEE WADLSYEA SRE
--------------------
True text :  WADSWORRE
--------------------
NW comparison :
 SU U¥MN EEE WADLSYEA SRE
 ------------WAD-S--WORRE
--------------------
```

As the reading procedure takes a long time (almost 2 seconds per image), I was able to build a dataset of about 50000 *expected-extracted* pairs stored locally on my machine.

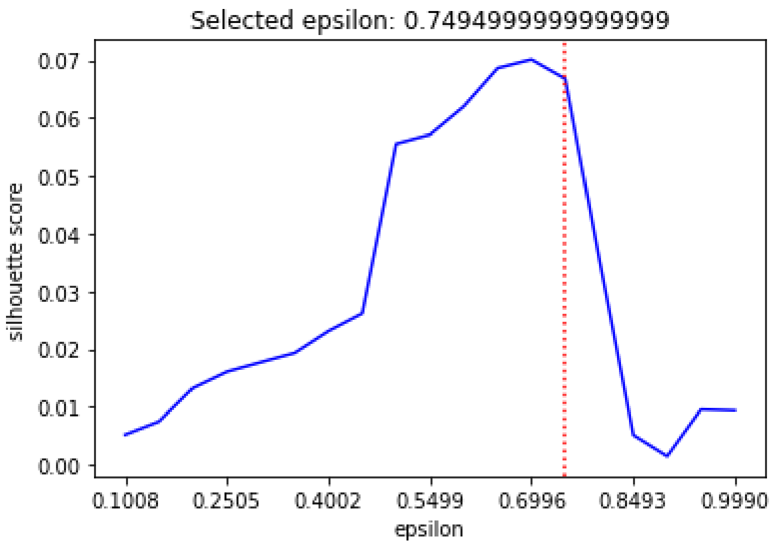
### Tuning

To give an idea of the tuning process of the hyperparameters of the clusterings, here are the results of the tuning on the entire dataset:

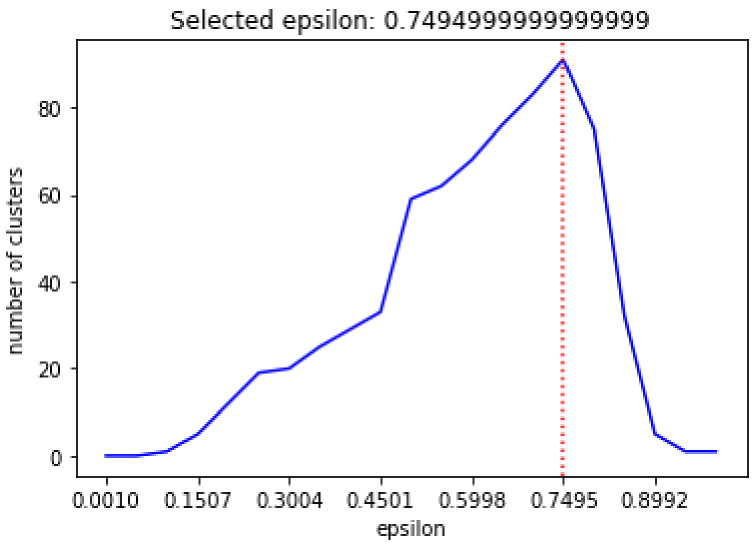
**Agglomerative Cluster:**



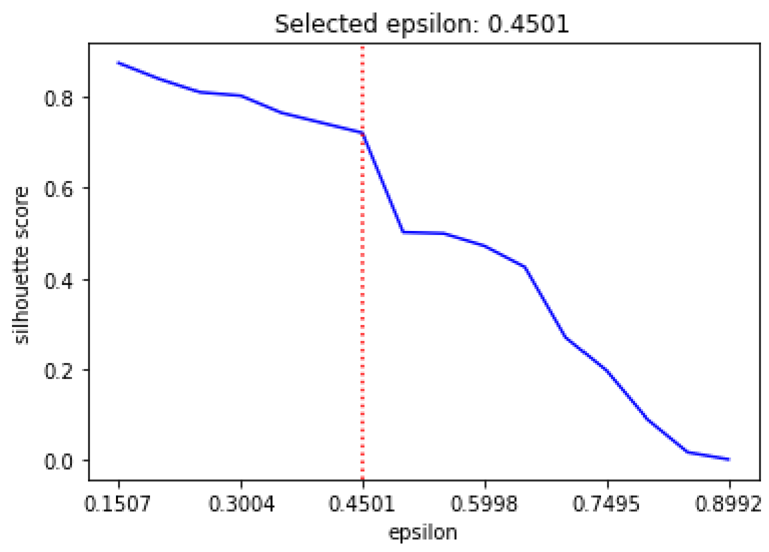
**DBSCAN Cluster:**



**GraphCluster trained to maximize the number of clusters**



**GraphCluster trained to maximize the silhouette score**



## Performance

The errors reported below are estimated with a 5-fold cross validation on the training set (80% of the original dataset, around 40000 entries).

The **RandomPerturber**, as the name suggests, maps nchars in a random one and is used as comparison.

| **Perturber** | **th=0** | **th=10** | **th=50** |
|---|---|---|---|
| **BasePerturber** | 0.4330 (*0.0081*) | 0.4270 (*0.0044*) | 0.4219 (*0.0120*) |
| **RandomPerturber** | 0.9790 (0.0008) | // | // |
| **AgglomerativePerturber** | 0.5000 (*0.0086*) | 0.4878 (*0.0120*) | 0.4887 (*0.0085*) |
| **DBSCANPerturber** | 0.4314 (*0.0088*) | 0.4225 (*0.0067*) | 0.4219 (*0.0120*) |
| **GraphPerturber trained to silhouette** | 0.4336 (*0.0063*) | 0.429 (*0.0059*) | 0.4242 (*0.0060*) |
| **GraphPerturber trained to num_clusters** | 0.4970 (*0.0218*) | 0.4682 (*0.0119*) | 0.4685 (*0.0143*) |

Average difference in expected number of mistakes per sentence done by a Perturber vs an OCR.

| Perturber | th=0 | th=10 | th=50 |
|---|---|---|---|
| **BasePerturber** | 0.2833 (*0.0281*) | 0.2856 (*0.0224*) | 0.2727 (*0.0308*) |
| **RandomPerturber** | 2.5614 (*0.0294*) | // | // |
| **AgglomerativePerturber** | 0.8719 (*0.0284*) | 0.8842 (*0.0315*) | 0.8844 (*0.0327*) |
| **DBSCANPerturber** | 0.2643 (*0.0150*) | 0.2931 (*0.0416*) | 0.2727 (*0.0308*) |
| **GraphPerturber trained to silhouette** | 0.2415 (*0.0114*) | 0.2901 (*0.0208*) | 0.2789 (*0.0357*) |
| **GraphPerturber trained to num_clusters** | 0.9114 (*0.2107*) | 0.8457 (*0.0383*) | 0.7822 (*0.0317*) |

## Concluding remarks

The results are clear: the best **Perturber** is the **BasePerturber** (with no clustering) that considers only nchars with more than 50 occurrences. They also show that Perturbers are more prone to errors than the **OCR** they were trained to replicate. This behaviour can be balanced, however, by choosing lower values of $\beta$.

Perturbers also have other possible applications that were not discussed in this paper: a *trusted* **Perturber** can be used to identify whether an **OCR** tends to make the same mistakes or if (as this is the case) it makes different mistakes every time. Perturbers can also be used in the wider context of finding patterns that arise when an object is morphed into another one.

As for future work on improving the transform stage, as right now it can only perturb already seen nchars, it would be interesting to infer the possible perturbations of an nchar base on its similarity with other ones. This would require a lot of images of characters in order to define how visually similar they are.

## References

1. Bazzo G.T., Lorentz G.A., Suarez Vargas D., Moreira V.P. (2020) Assessing the Impact of OCR Errors in Information Retrieval. In: Jose J. et al. (eds) Advances in Information Retrieval. ECIR 2020. Lecture Notes in Computer Science, vol 12036. Springer, Cham. https://doi.org/10.1007/978-3-030-45442-5_13
2. Needleman, S., Wunsch, C.: A general method applicable to the search for similarities in the amino acid sequence of two proteins. J. Mol. Biol. **48**, 443–453 (1970)
3. Slowikowski , K: A simple version of the Needleman-Wunsch algorithm in Python. https://gist.github.com/slowkow/06c6dba9180d013dfd82bec217d22eb5
4. Pastore M and Calcagnì A (2019) Measuring Distribution Similarities Between Samples: A Distribution-Free Overlapping Index. *Front. Psychol.* 10:1089. doi: 10.3389/fpsyg.2019.01089
5. Silhouette Score : https://scikit-learn.org/stable/modules/generated/sklearn.metrics.silhouette_score.html?highlight=silhouette#sklearn.metrics.silhouette_score
6. Landlord handriwriting recognition : https://www.kaggle.com/landlord/handwriting-recognition
7. Improvements to the images feeded to OCR : https://stackoverflow.com/questions/9480013/image-processing-to-improve-tesseract-ocr-accuracy

# Appendix

## Computational cost growth of the optimal splitting problem

For any string $L_n = l_1 \dots l_n$, there are exactly $F(n)$ ways to split it in unichars and bichars, where $F(n)$ is the $n$-th Fibonacci number.

*Proof*

by induction on $n$, let the thesis hold true for any string of length $< n$. Let $S(n)$ be the number of splits of $L$.

**Case** $n = 1$

$L_1 = l_1$ can only be split with $P = \{\{l1\}\}$, hence $S(L) = 1$

**Case** $n = 2$

$L_2 = l_1 l_2$ can be split with $P = \{\{l1\}, \{l2\}\}$ or $P = \{\{l1l2\}\}$, hence $S(L) = 2$

**Induction step**

Let $L_n = l_1 \dots l_{n-1} l_n$, $P$ be a partition of $L_n$ in unichars and bichars.

Then, $P$ either contains $l_n$ or it contains $l_{n-1}l_n$ (as $P$ is a partition and $l_n$ can form a bichar only with $l_{n-1}$ due to the consecutivity of bichars).

We call $P$ of the first kind if it contains $l_n$ and of the second kind if it contains $l_{n-1}l_n$.

The result derives from the following remarks:

- Any $P$ of the first kind can be bijectively mapped into a partition of $L_{n-1}$ via $P \mapsto P \setminus \{l_n\}$
- Any $P$ of the second kind can be bijectively mapped into a partition of $L_{n-2}$ via $P \mapsto P \setminus \{l_{n-1}l_n\}$

Therefore $S(n) = S(n-1) + S(n-2) = F(n-1) + F(n-2) = F(n)$

$\square$

## Fit

```
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 consecutive 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
```

## Transform

```
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 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
```

## Split

```
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

    i <- 0
    splits <- empty list

    WHILE True:
        if i >= len(text):
            break

        tri = text[i:i+3] // The trichar ti ti+1 ti+2

        spl = Optimal_trichar_split(tri, p)
        IF len(spl) == 1:
            EXTEND splits WITH spl[0]
            break
        ELSE:
            EXTEND splits WITH spl[:-1]
            i += (3-length(spl[-1]))

    return splits

End Function

Function Optimal_trichar_split(tri, p)
    ----
    Inputs:
        tri : a unichar, bichar or trichar
        p : // As in Transform
    Outputs:
        spl : the split of `tri` in unichars and bichars that has the highest
 probability of being perturbed
    ----

    probs <- empty list
    FOR all splits spl of tri // That are: [t0, t1, t2], [t0t1, t2], [t0, t1t2]
        APPEND Compute Probability_perturbation(spl, p) TO probs

    return argmax(probs) // in case of tie, returns the first split in alphabetical
// lexicographic order
End Function

```

```
Function Probability_perturbation(ls, p)
    ----
    Inputs:
        ls : a list of unichars and bichars
        p : // As in Transform, a map from nchar to probability to be perturbed
    Outputs:
        prob : the probability of at least one element of `ls`to be perturbed
    ----

    return 1 - product([1-p[nchar] for nchar in ls])
End Function
```

## Example of a Split operation

```
text = 'cane'
p = {'c' : 0.5, 'a' : 0, 'n' : 0.1, 'e' : 0.1, 'ca' : 0.2, 'an' : 0.2, 'ne' : 0.9}
splits <- empty list

// Operations performed by Split(text, p)

// 1: Consider the first trichar, split it in all the possibilities and proceed with
// the best one. Append all but the last element of the split to splits
'can'
c a n   -> 1 - (1-0.5)*(1-0)*(1-1)  = 1
c an    -> 1 - (1-0.5)*(1-0)        = 0.5
ca n    -> 1 - (1-0.2)*(1-1)        = 1

// optimal partition of 'can' : ['c', 'a', 'n']
// splits = ['c', 'a']

// 2: Consider another trichar (bichar, as the string is over) whose first element(s) // are the last of the split found
'ne'
n e     -> 1 - (1-1)*(1-0.1)        = 1
ne      -> 1 - (1-0.9)              = 0.9

// optimal partition of 'ne' : ['n', 'e']
// splits = ['c', 'a', 'n']

// 3: Repeat step 2
'e'
e       -> 1 - 0.1                  = 0.9

// splits = ['c', 'a', 'n', 'e']
```

# 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
```

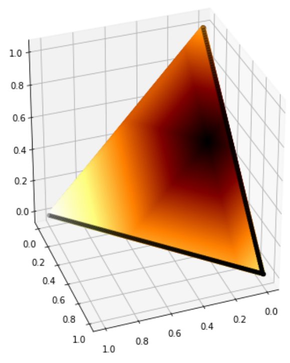
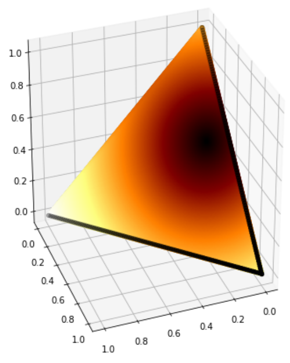
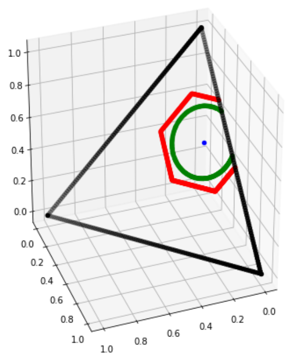
```
/*{
    'i': {' I': 1, ' i': 1, 'I': 1, 'i': 1},
    'im': {'Im': 1},
    'm': {'m': 1, 'm ': 1},
    'n': {'n': 1, 'n ': 1},
    'u': {'u': 1},
    'un': {'un': 1}
}*/
// p
// {'i': 0.75, 'im': 1.0, 'm': 0.5, 'n': 0.5, 'u': 0.0, 'un': 0.0}
// s
// {'i': 4, 'im': 1, 'm': 2, 'n': 2, 'u': 1, 'un': 1}

v = Vectorize(perturbations['i'])
// [0.  , 0.25, 0.25, 0.25, 0.  , 0.  , 0.  , 0.25, 0.  , 0.  , 0.  , 0.  ]

nch2idx['I'], nch2idx['i'], nch2idx[' I'], nch2idx[' i']
// 2, 7, 3, 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 'umido' according to the way it is split:
    ['u', 'm', 'i', 'd', 'o'] : 0.875
    ['um', 'id', 'o'] : 0
    ['um', 'i', 'd', 'o'] : 0.75
*/
```

## Continuous Overlapping Distance

The Continuous Overlapping Distance is defined over simplexes (vectors with sum 1) and is

$$COD(v, w) = 1 - \sum_t \min\{v_t, w_t\} \in [0, 1] \tag{2}$$

Chosen $v_0 = \left(\frac{1}{8}, \frac{3}{8}, \frac{1}{2}\right)$ as center:

On the left, COD distances

In the center, euclidean distances

On the right, the boundary of a neighbourhood of radius $\frac{1}{5}$ in the euclidean (green) and COD (red) metrics. Note the exagonal shape (what would it look like in 3 dimensions? The exagon has no obvious solid counterpart).