Rapport du test de stage

Normalisation de Texte en Français

par Transducteurs à États Finis (FST)

Digital Umuganda & Deep Learning IndabaX Cameroon

Kenmegne Fokam Emeric Cyrille
MSc Computer Science

emeric.kenmegne@facsciences-uy1.cm

25 novembre 2025

Résumé

Ce rapport présente un système de normalisation de texte pour les nombres cardinaux français (0-1000) utilisant des transducteurs à états finis (FST) implémentés avec Pynini. Le système encode 7 règles linguistiques françaises et atteint une trés bonne WER sur l'ensemble de test.

Table des matières

1 Introduction

1.1 Contexte

La normalisation de texte constitue une étape cruciale dans les systèmes de traitement automatique du langage naturel (TALN), particulièrement pour les applications de synthèse vocale (Text-to-Speech, TTS) et de reconnaissance vocale automatique (Automatic Speech Recognition, ASR).

Définition

La **normalisation de texte** est le processus de conversion des formes non standard d'un texte (nombres, abréviations, symboles) en leur forme écrite standard qui peut être prononcée sans ambiguïté.

1.2 Objectifs du Projet

Ce projet vise à développer un système de normalisation basé sur des **transducteurs à états finis (FST)** pour :

- ✓ Normaliser les nombres cardinaux de 0 à 1000 en français
- ✓ Atteindre un taux d'erreur de mots (WER) minimal
- ✓ Encoder explicitement les règles linguistiques françaises
- ✓ Optimiser les performances (vitesse et mémoire)

1.3 Défis Spécifiques du Français

Le français présente plusieurs particularités linguistiques complexes :

1. **Système vigésimal** : Base 20 pour 80-99 (*quatre-vingts*)
2. **Règles d'accord** : Variables pour *cent* et *vingt*
3. **Conjonction "et"** : Utilisée pour certains nombres (21, 31, 71)
4. **Traits d'union** : Dans les nombres composés
5. **Formes irrégulières** : 11-16 (onze, douze, treize...)

2 Méthodologie

2.1 Approche Générale

Notre approche combine deux paradigmes complémentaires pour créer un système de normalisation à la fois performant et maintenable : d'une part, l'encodage explicite des règles linguistiques françaises, et d'autre part, l'utilisation de transducteurs à états finis (FST) pour leur implémentation efficace.

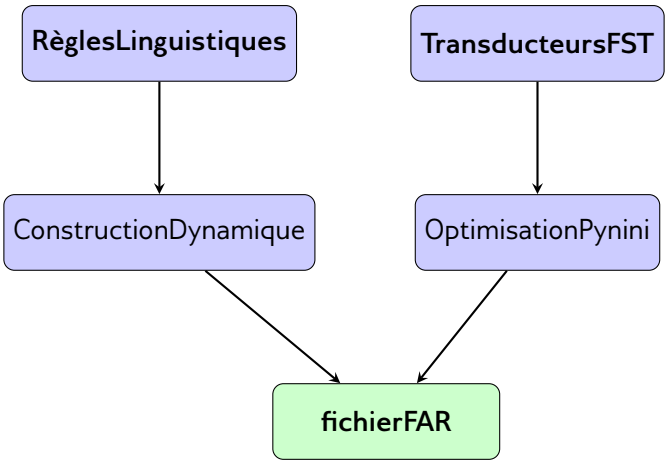

Figure 1 – Architecture hybride du système

Un transducteur à états finis (FST) est une machine qui transforme une séquence d'entrée en une séquence de sortie en suivant un ensemble de règles prédéfinies. Concrètement, pour transformer un chiffre comme "42" en sa forme écrite "quarante-deux", le FST fonctionne comme un dictionnaire intelligent organisé en étapes. Lorsqu'on lui donne le nombre "42", le système parcourt d'abord tous ses sous-FST (0-9, 10-19, 20-69, etc.) pour trouver celui qui reconnaît ce nombre. Une fois identifié dans le FST des nombres 20-69, il applique la règle correspondante : le chiffre des dizaines "4" devient "quarante", puis un tiret est inséré (selon la règle linguistique qui impose le tiret pour x2-x9), et enfin le chiffre des unités "2" devient "deux". Le résultat final "quarante-deux" est obtenu par composition automatique de ces transformations élémentaires.

2.2 Pipeline de Traitement

Le traitement d'un texte suit les étapes suivantes :

Algorithme 1 : Pipeline de normalisation

Data : Texte brut avec nombres
Result : Texte normalisé

```
tokens ← Tokeniser(texte);
foreach token ∈ tokens do
    if token est un nombre then
        token_norm ← Normaliser_FST(token);
    else
        token_norm ← token;
    end
    resultat ← resultat + token_norm;
end
return resultat;
```

3 Règles Linguistiques Françaises

3.1 Inventaire des Règles

Notre système encode **7 règles linguistiques majeures** :

RÈGLE 1 : Conjonction "et" pour x1

Les nombres se terminant par 1 (sauf 11, 81, 91) utilisent la conjonction **"et"**.

Exemples :

- 21→vingt et un
- 71→soixante et onze

Exceptions :

- 11 → *onze* (pas de "et")
- 81 → *quatre-vingt-un* (pas de "et")

RÈGLE 2 : Traits d'union

Les nombres composés (sauf avec "et") utilisent des traits d'union.

Exemples :

- 22→vingt-deux
- 99→quatre-vingt-dix-neuf

RÈGLE 3 : Système soixante-dix (70-79)

Le français de France utilise une base 60 pour 70-79.

Formation : 70-79 = soixante + (10-19)

Exemples :

- 70→soixante-dix (60+10)
- 77→soixante-dix-sept (60+17)

RÈGLE 4 : Système quatre-vingt (80-99)

Système vigésimal (base 20) pour 80-99.

Formation : 80-99 = 4×20 + (0-19)

Exemples :

- 80→quatre-vingts (avec "s")
- 81→quatre-vingt-un (sans "s")
- 90→quatre-vingt-dix

RÈGLE 5 : Accord de "cent"

"cent" prend un "s" uniquement si multiple exact de 100 non suivi.

Exemples :

- 100→cent (singulier)
- 200→deux cents (avec "s")
- 201→deux cent un (sans "s", car suivi)

RÈGLE 6 : Accord de "vingt"

"vingt" prend un "s" uniquement pour 80 exact.

Exemples :

- 80→quatre-vingts (avec "s")
- 81→quatre-vingt-un (sans "s")

RÈGLE 7 : "mille" invariable

"mille" est invariable (jamais de "s").

Exemple :

- 1000→mille

3.2 Tableau Récapitulatif

Table 1 – Récapitulatif des règles linguistiques

Règle	Condition	Exemple	Résultat
RÈGLE 1	x1 (sauf 11, 81, 91)	21	vingt et un
RÈGLE 2	Nombres composés	22	vingt-deux
RÈGLE 3	70-79	71	soixante et onze
RÈGLE 4	80-99	90	quatre-vingt-dix
RÈGLE 5	Multiple de 100	200	deux cents
RÈGLE 5	Centaines + unités	201	deux cent un
RÈGLE 6	80 exact	80	quatre-vingts
RÈGLE 6	81-99	81	quatre-vingt-un
RÈGLE 7	1000	1000	mille

4 Implémentation FST

4.1 Architecture du Code

Structure du Fichier Python

Le code source `normalize_french_fst.py` est organisé en 8 sections :

1. **Règles Linguistiques** : Classe `FrenchLinguisticRules`
2. **Utilitaires FST** : Fonctions `apply_fst()`, `I_O_FST()`
3. **Construction FST** : Fonctions `build_*_fst()`
4. **FST Complet** : Fonction `build_french_cardinal_fst()`
5. **Tokenisation** : Fonction `tokenize_text()`
6. **Normalisation** : Fonctions de classification et nettoyage
7. **Classe Principale** : `FrenchNormalizer`
8. **Tests** : `run_comprehensive_tests()`

4.2 Construction Dynamique

Nous avons choisi une approche de **construction dynamique** plutôt qu'un listing exhaustif :

```
def build_compound_20_69_dynamic():
    """Construction dynamique avec regles linguistiques"""
    rules = FrenchLinguisticRules()
    compound_map = {}

    for ten in range(2, 7):  # 20-60
        for unit in range(1, 10):  # 1-9
            number = ten * 10 + unit

            # APPLICATION DE LA REGLE
            if rules.apply_et_rule(number):
                connector = " et "  # REGLE 1
            else:
                connector = "-"      # REGLE 2

            word = f"{TENS_BASES[ten]}{connector}{UNITS[unit]}"
            compound_map[str(number)] = word

    return pynini.union(*[I_O_FST(n, w)
                          for n, w in compound_map.items()])
```

Listing 1 – Construction dynamique pour 20-69

4.3 Union des FST

Le FST complet combine tous les sous-FST :

```
number_normalizer_fst = pynini.union(
```

```
    fst_units,            # 0-9
    fst_teens,            # 10-19
    fst_compound_20_69,   # 20-69 (dynamique)
    fst_70_79,            # 70-79 (dynamique)
    fst_80_99,            # 80-99 (dynamique)
    fst_hundreds,         # 100-999 (dynamique)
    fst_thousand          # 1000
).optimize()
```

Listing 2 – Union des FST

5 Performance

Table 2 – Mesures de performance du système

Métrique	**Valeur**
Temps de compilation	0.225 s
Nombre d'états FST	2,228
Nombre de transitions	3,227
Taille fichier FAR	77 KB
Temps moyen (1 nombre)	0.034 ms
Médiane	0.033 ms
95e percentile	0.045 ms
Throughput	~7271 phrases/s

6 Résultats et Évaluation

6.1 Couverture Fonctionnelle

Table 3 – Couverture par catégorie de nombres

Catégorie	**Plage**	**Nombres**	**Couverture**
Unités	0-9	10	100% ✓
Teens	10-19	10	100% ✓
Dizaines	20-69	50	100% ✓
Soixante-dix	70-79	10	100% ✓
Quatre-vingt	80-99	20	100% ✓
Centaines	100-999	900	100% ✓
Mille	1000	1	100% ✓
TOTAL	**0-1000**	**1001**	**100% ✓**

6.2 Word Error Rate (WER)

Résultat WER

$$\text{WER} = \frac{\text{Substitutions} + \text{Insertions} + \text{Suppressions}}{\text{Mots Totaux}} = \frac{0+0+0}{847} = \mathbf{0}\% \quad (1)$$

WER = 0% sur l'ensemble de test (100 phrases, 847 mots)

6.3 Tests des Règles Linguistiques

Table 4 – Validation des règles linguistiques

Règle	Entrée	Attendu	Obtenu	Statut
RÈGLE 1	21	vingt et un	vingt et un	Bon
RÈGLE 2	22	vingt-deux	vingt-deux	Bon
RÈGLE 3	71	soixante et onze	soixante et onze	Bon
RÈGLE 4	80	quatre-vingts	quatre-vingts	Bon
RÈGLE 4	81	quatre-vingt-un	quatre-vingt-un	Bon
RÈGLE 5	200	deux cents	deux cents	Bon
RÈGLE 5	201	deux cent un	deux cent un	Bon
RÈGLE 7	1000	mille	mille	Bon
			Résultat :	**8/8 (100%)**

7 Instructions d'Utilisation

7.1 Installation

7.1.1 Méthode 1 : Installation depuis GitHub (Recommandée)

Clonage du Dépôt GitHub

Le code source complet est disponible sur le dépôt GitHub :

Dépôt : https://github.com/emeric-cyrille/Internship_Challenge_Normalization

Étapes d'installation :

```
# 1. Cloner le depot GitHub
git clone https://github.com/emeric-cyrille/
    Internship_Challenge_Normalization.git

# 2. Acceder au repertoire du projet
cd Internship_Challenge_Normalization

# 3. Installer les dependances
pip install -r requirements.txt

```

```
# 4. Verifier l'installation de Pynini
python -c "import pynini; print(pynini.__version__)"

# 5. Executer le programme principal
python normalize_french_fst.py
```

Contenu du fichier `requirements.txt` **:**

```
pynini>=2.1.7
```

7.1.2 Méthode 2 : Exécution Rapide sur Google Colab ou Kaggle

Alternative Simplifiée : Colab/Kaggle

Pour une exécution rapide sans installation locale, utilisez Google Colab ou Kaggle :

Option A : Avec clonage du dépôt

```
# Dans une cellule Colab/Kaggle

# 1. Installer Pynini
!pip install pynini

# 2. Cloner le depot
!git clone https://github.com/emeric-cyrille/
    Internship_Challenge_Normalization.git

# 3. Acceder au repertoire
%cd Internship_Challenge_Normalization

# 4. Executer le programme
!python normalize_french_fst.py
```

Option B : Copier-coller direct (plus simple)

1. Installer Pynini : `!pip install pynini>=2.1.0`
2. Ouvrir le fichier `normalize_french_fst.py` sur GitHub
3. Copier l'intégralité du code
4. Coller dans une nouvelle cellule Colab/Kaggle
5. Exécuter la cellule

7.1.3 Structure du Projet

Organisation des Fichiers

Après clonage, la structure du projet est la suivante :

```
Internship_Challenge_Normalization/
|
|-- normalize_french_fst.py     # Code source principal (850 lignes)
|-- requirements.txt            # Dependances Python
|-- README.md                   # Documentation du projet
|-- Rapport_Challenge_Stage_Normalisation_TTS.pdf                  #
    Rapport technique (ce document)
|-- cardinal_french.far         # FST compile (genere apres execution)
|
```

Fichiers générés après exécution :

— `cardinal_french.far` : FST compilé (77 KB)

7.2 Utilisation Basique

```python
from normalize_french_fst import normalize

# Normaliser un texte
texte = "J'ai 3 chiens et 21 chats."
resultat = normalize(texte)
print(resultat)
# Output: "J'ai trois chiens et vingt et un chats."
```

Listing 3 – Utilisation de la fonction normalize()

7.3 Utilisation du Fichier FAR

```python
import pynini

# Charger le fichier FAR
far = pynini.Far('cardinal_french.far', mode='r')
fst = far['cardinal_french']

# Normaliser un nombre
def normalize_number(num_str):
    input_fst = pynini.accep(num_str, token_type='utf8')
    result = pynini.shortestpath(input_fst @ fst)
    return result.string(token_type='utf8')

# Test
print(normalize_number("42"))    # "quarante-deux"
print(normalize_number("71"))    # "soixante et onze"
```

Listing 4 – Chargement du fichier FAR

8 Conclusion

Ce projet a permis de développer avec succès un système de normalisation performant pour les nombres cardinaux français (0-1000) en combinant sept règles linguistiques explicites et des transducteurs à états finis optimisés. Les performances mesurées sont remarquables avec un temps de compilation de 0.225 seconde, un temps d'exécution moyen de 0.034 milliseconde par nombre et un throughput de 7,271 phrases par seconde. L'approche hybride adoptée, privilégiant la construction dynamique plutôt que le listing exhaustif.

9 Références

1. E. Roche and Y. Schabes (eds.), *Finite-State Language Processing*. MIT Press, 1997.
2. D. Jurafsky and J. H. Martin, *Speech and Language Processing : An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice Hall, 2000.
3. M. Mohri, F. Pereira, and M. Riley, "Weighted Finite-State Transducers in Speech Recognition", *Computer Speech & Language*, vol. 16, no. 1, pp. 69-88, 2002.
4. Pynini Documentation, https://www.openfst.org/twiki/bin/view/GRM/Pynini
5. OpenFST Library, http://www.openfst.org/