Report: Text Normalization System (Cardinal Numbers)

Author: CHELEN TATAP BILL CHRISTIAN
Date: November 2025
Language: English (0-1000)

1. Executive Summary

This project implements a text normalization system based on **Weighted Finite State Transducers (WFST)** using the `pynini` library. The primary goal is to convert raw text containing numeric digits (e.g., *"I have 123 apples"*) into their standard written form (e.g., *"I have one hundred and twenty three apples"*).

The solution focuses on English cardinal numbers ranging from **0 to 1000**, including negative numbers and specific digit sequences starting with zero (e.g., *"004"*). The system achieves a Word Error Rate (WER) of **0.00%** on the target subset.

2. Methodology

The core of this solution uses a "Grammar-based" approach rather than simple regular expressions. We construct a graph of grammatical rules where the input is a string of digits, and the output is the text representation.

2.1 Bottom-Up Construction

We built the grammar using a modular approach:

1. **Atomic Units:** Mappings for digits (0-9) and teens (10-19).
2. **Composite Numbers (20-99):** A rule that combines "Tens" (twenty, thirty...) with "Digits".
3. **Hundreds (100-999):** A rule that enforces the specific English syntax found in the dataset (e.g., usage of "hundred **and**").

2.2 The Logic of Weights (Solving Ambiguity)

A critical challenge in FSTs is ambiguity. For an input like `123`, a naive FST might see two valid paths:

- **Path A (Incorrect):** `1` -> "one", `2` -> "two", `3` -> "three". (Three separate operations).
- **Path B (Correct):** `123` -> "one hundred and twenty three". (One complex operation).

To ensure the system selects Path B, we implemented **Weighting**:

- **Penalties:** We assigned a weight of **1.0** to the single digit rule (`1-9`).
- **Preferences:** We assigned a weight of **0.0** to complex rules (Hundreds, Teens, Composites).
- **Shortest Path:** Using `pynini.shortestpath`, the system calculates the "cost" of translation. Path A costs 3.0, while Path B costs 0.0. The system deterministically chooses the lowest cost.

2.3 Context Dependent Rewrite

The final grammar is wrapped in a `pynini.cdrewrite` function. This allows the Finite State Transducer to scan entire sentences (e.g., *"Temperature is -5 degrees"*), identify the numbers, and transform them without altering the surrounding text.

3. Code Structure

The project consists of the following files:

- **`normalizer.py`**: The main source code. It contains the `EnglishNumberNormalizer` class. When executed, it compiles the grammar and exports it to a binary file.
- **`evaluate.py`**: The testing script. It reads the dataset (`test_en.txt`), filters for numbers within the 0-1000 scope, and calculates the WER using the `jiwer` library.
- **`normalizer.far`**: The compiled Finite State Archive.

4. Instructions for Reproduction

4.1 Requirements

Install the necessary Python libraries:

codeBash

```
pip install pynini==2.1.7  jiwer==4.0.0
```

(Note: Pynini requires a Linux, macOS, or WSL environment).

4.2 How to Generate the FAR File

Run the main script to build the FST and export the binary archive:

codeBash

```
python normalizer.py
```

Output: This will generate a file named `normalizer.far` in the current directory.

4.3 How to Use the FAR File

The FAR file allows the grammar to be used without recompiling the Python code. Here is how to load and use it programmatically:

codePython

```python
import pynini

# 1. Load the FAR file
far = pynini.Far("normalizer.far", mode="r")
# 2. Extract the FST (Key: 'NORMALIZE')
fst = far["NORMALIZE"]

# 3. Apply to text
def normalize(text):
    # Escape text to FST
    input_lattice = pynini.escape(text)
    # Compose: Input @ Grammar
    composed = input_lattice @ fst
    # Get shortest path (best result)
    return pynini.shortestpath(composed).optimize().string()


print(normalize("123"))
```

4.4 How to Evaluate

Run the evaluation script to check the WER:

codeBash

```bash
python evaluate.py
```

5. Results

- **Compilation Time:** < 1 second.
- **Execution Speed:** Milliseconds per sentence (optimized binary graph).
- **WER (Subset 0-1000):** 0.00% (Perfect match on valid inputs).

6. More Details

For more technical details about the project , please refer to the **readme** file.