# APOLLO: A *Pie Forzado* Automatic Poetry Composer

Rubén Hinojosa Chapel
Barcelona, Spain
contact@hinojosachapel.com
https://linkedin.com/in/hinojosachapel

## Abstract

In the frame of the **Creative Turing Tests 2018** (http://bregman.dartmouth.edu/turingtests), organized by the Dartmouth College's Neukom Institute for Computational Science, it is presented to the LyriX challenge a ***Pie Forzado* automatic poetry composer**, able to create a poem from a word or phrase. An empty seed value is acceptable, though. A *Pie Forzado* is a verse that occurs when the author is imposed the word or verse in which the poem must end.

## Introduction

Apollo is a Language Modeling system based on n-grams and Markov Chains. The modeling logic, training and generation, are encapsulated in a POCO C# class, it does not have any dependency on an external framework. For simplicity in the distribution of the executable file, the project was developed as a WPF solution. So, in order to execute Apollo, a Windows computer is needed. On the other hand, for running the source code, you need Microsoft Visual Studio.

The system's execution involves two steps. The first one occurs on start, where the training phase takes place. The second one occurs a very little after, where Apollo becomes ready for its tasks. It is possible to overcome the first step if the system comes with pre-trained models, but because the academic, non-commercial, goal of the project, this step is included.

## Models, Training and Algorithm Overview

Apollo uses two Markov Chains, one with 2-grams and 3-grams extracted from poems, and the other one contains 3-grams extracted from open texts. The former allows to generate the actual poem, while the later allows to find the user-entered word. When the user enters a phrase, Apollo picks up the first word.

The 3-grams stored in the nodes of the Poems Graph are not adjacent, they keep a distance of 3. For instance: "due, by the grave and thee." => "due, by the", "grave and thee.", but not "by the grave" nor "the grave and". The same strategy is used with the Texts Graph.

Two auxiliary data structures are used for fast retrieval of an n-gram. That is, given a word and a graph, the retrieval algorithm returns a node that holds an n-gram that ends with that word. If an n-gram that starts with that word is found, returns a next node (actually a previous one).

The generative process predicts two or three words at once (full n-grams) with each transition. Dictionaries of nodes are used for representing the graphs. Each key is an n-gram, while the associated value holds a list of indices in the dictionary. These indices represent possible transitions from the current n-gram (node) to a next (previous) n-gram.

Nodes can be labeled as StartGram or EndGram, or not labeled at all. A node is StartGram if its n-gram starts with the first word of the trained poem or text. This is useful during the generative process because the halt condition is reached when:

1- A minimum number of verses have been generated and
2- A StartGram node is reached or a maximum number of verses have been generated.

A node is EndGram if its n-gram ends with the last word of the trained poem, or if it ends with the last word of a paragraph of the trained text. This is useful when, during the generative process, a StartGram is reached, because the algorithm can choose randomly an EndGram for moving forward. It's also useful for starting the generative process when the seed value is empty. The algorithm randomly chooses an EndGram from the Text Graph.

Poems and texts tokenization keeps punctuation marks and newline.

The generative algorithm works in backward direction, from the end to the beginning, starting with the first word the user entered so, the state machines represent transitions also in backward direction. The algorithm involves two basic steps:

1- Find, in the Texts Graph, an n-gram related to the first user-entered word and predict the next (previous) n-gram. While the first word of the predicted n-gram from the current one is not found in the Poems Graph, keep predicting and building the poem over the Texts Graph.
2- When the first word of the predicted n-gram is found in the Poems Graph, switch the model and keep predicting and building the poem until the halt condition is reached.

After the raw poem text is generated, a text format is carried out and duplicated words are eliminated.

It is possible that the algorithm keeps navigating over the Texts Graph, and that possibility is considered. Nevertheless, although possible, intuitively that probability is very low. As soon as the algorithm finds a high-frequency closed class item, like prepositions, pronouns, conjunctions, or determiners, it is highly probably to find it in the Poems Graph, and so, the algorithm can move forward to the second step.

## Datasets

Two kinds of dataset are used, one for each model. The Poems Model is trained with a number of poems created by several relevant English-language poets plus Petrarch translations. They were normalized and pre-cleaned. The Texts Model is trained with public domain texts from the Project Gutenberg (github.com/GITenberg) and other sources. They were also normalized and pre-cleaned.

Every file has a title, which is not considered during the training step. Poems are stored one per file, because the need to recognize StartGrams and EndGrams. A file containing more than one poem is not allowed. Problematic words like 'tis, 'twas, 'twere and 'twixt were replaced with their modern equivalents. Also Roman numbers and words like CHAPTER were removed from the original texts. "...." signs were converted into "..." signs. "_," and "_." were converted into "," and ".", in addition to some other text cleanings.