SDG: A Synthetic Datastream Generator

Grzegorz Stepien
`grzegorz.stepien@rwth-aachen.de`

December 2018

Contents

1 Introduction

Today's rapid progress of data storage and processing technology opens up a wide spectrum of possibilities for gaining new insights into real world phenomena. The prevalence of sensors in devices like smartphones, cars and industry machinery as well as technologies like the Internet lead to an ever increasing amount of data being continuously produced. While entailing great potential for new insights, it is the sheer amount and heterogeneity of data which makes the task of extracting meaningful information far from trivial. Consequently, over the last decades *Data Science* emerged as an interdisciplinary field that aims at applying methods from statistics, signal processing, information theory and computer science in order to tackle various aspects of big data analysis. Areas of application range from genome research and personalized medicine [1], particle detection in collider experiments [2, 3] through to prediction of earthquakes [4], crimes [5], business bankruptcies [6], machinery failures [7] (known as *predictive maintenance*) and many others.

Time series stream analysis is a subfield of Data Science. The term "time series" refers to data which may be represented as a function of discrete points in time. The term "stream" indicates that new data is being continuously produced and needs to be analyzed in an online, real-time fashion (as opposed to, for example, historic time series which are stored as a whole). Typical sources of time series streams are *sensors.* In certain time intervals, the latter produce some real valued measurements (like temperature, pressure, acceleration, . . .) or discrete labels (like "defect", "no defect", . . .) reflecting real world phenomena. The analysis of time series streams typically serves the purpose of supporting some sort of (semi-) automated decision making and event detection & prediction.

Current efforts to implement the concept of *Industry* 4.0[8] are a very prominent example for the importance of efficient analysis of time series streams. This hoped-for "4th industrial revolution" aims at minimizing production cost while simultaneously maximizing production flexibility in order to enable higher individualization of products at lower costs. At an intra-factory level, this entails the vision of a self organizing *smart factory* that employs interconnected sensors, embedded production systems and RFID chips on each product. Communication between those components is realized by an *Internet of things*-type infrastructure. This allows to create a (semi-) bijective mapping between the physical production environment and a digital representation of the former. Systems with this type of close physical-virtual interrelation are called *cyber-physical systems.* Input from the physical side of said mapping may be continuously fed into data processing methods from fields like pattern recognition, data mining and/or machine learning. The results of the former may then be used by an automated decision making process, the output of which is fed back into the physical side in order to trigger some production reorganization/optimization. At an inter-factory level, on the other hand, a main goal of Industry 4.0 is to digitally interconnect factories and businesses in order to enable automated and more flexible supply-chains necessary for the aforementioned intra-factory flexibility.In theory, such an infrastructure both on intra- as well as inter-factory levels would allow for highly dynamical production processes with a high level of decentralized self organization and little need for human oversight.

1.1 Challenges of time series stream analysis

The need for efficient methods for the analysis of live sensor data streams is apparent. A more practical, but nonetheless important issue when developing new methods for data analysis is the need for a controlled testing environment where those methods can be evaluated against various data parameters. Ideally, such an environment should take care of: Synthetically generating multiple, inter-correlated numeric and symbolic data streams based on a number of parameters controlling pattern frequency & correlation, noise, range, dimensionality, etc.

Our *Synthetic Datastream Generator (SDG)* provides a *motif* based, highly modular framework for doing just that.

1.2 Motifs as patterns of interest

The SDG uses so called *motifs* as patterns of interest. Given a numeric sequence, a motif is a subsequence which appears at least twice in said sequence, i.e., a subsequence that is *similar* to a different subsequence with respect to some measure (like the z-normalized Euclidean distance [9] or dynamic time warping [10]). The motivation for choosing motifs as patterns of interest is the assumption that motifs do not simply occur at random, but rather that the order of their occurrence holds valuable information for the detection and prediction of certain real world events. Our core assumptions on the nature of our data may therefore be summarized as follows:

1. Some *entity* is the subject of repeated measurements resulting in a number of numerical and/or symbolic time series streams.
2. Real world *events* related to the entity of interest are reflected by the occurrence of:
   - One or more specifically shaped, finite subsequences in the numerical data streams. Such repeating subsequences are called *motifs*.
   - One or more specific labels in the symbolic data streams.
3. Motifs and labels might be distributed among several sensor data streams and might not all occur at the same time. Some of them might occur before and some after the actual event. Earthquakes, for example, are usually foreshadowed by imperceptible preliminary tremors [11]. Similarly, Machine defects might announce themselves in one form or another (increasing vibration, a specific sound, pressure drop/increase, or a specific combination such patterns).
4. Events of the same type or nature produce similar sets of Motifs with similar temporal delays between them.

1.3 Our contribution

During our research (as of December 2018), we were unable to find an existing synthetic data stream generator which would fulfill the following requirements (which are based on the observations from the last two sections):

- It should be open source.

- It should continuously generate an arbitrary number of parallel numeric and symbolic data streams.
- The numeric streams should be made up of repeating subsequences (motifs) which are connected via random inter-motif sequences.
- The functional form of both motif as well as inter-motif sequences should be customizable.
- The symbolic streams contain randomly emitted string labels from a predefined vocabulary.
- The order in which subsequences and labels appear is determined by a probability distribution which is conditioned on the combined history of previously emitted labels from all dimensions.
- The complexity and ambiguity (i.e., the entropy) of said probability distribution should be customizable by a set of meaningful parameters.
- The data elements of each stream should be emitted in an asynchronous manner (i.e., with random delay fluctuations) and annotated with a timestamp of the time the element became available (in a real world scenario, data streams from different sensors are not necessarily synchronized).
- The numeric data elements should also be annotated with a ground truth label containing information on whether they belong to a motif or to a random inter-motif sequence.

We therefore implemented such a system ourselves: A modular and highly configurable *synthetic data steam generator (SDG)* in the programming language R which fulfills all of the points listed above. Our solution employs a generalized form of random walks for both motif and inter-motif sequences (but can be easily configured for other types of sequences). The former is computed using a fixed random seed for each motif type in order to achieve repeating subsequences. SDG also entails a set of driver methods aiming at enabling an easy and convenient configuration and instantiation of SDG's main components via JSON files. Figure 1.1 contains an example plot of two numeric and two symbolic data streams generated by our data generator. Inter-motif sequences are depicted in grey and the other colors indicate repeating subsequences (i.e., subsequences of the same color have the same shape except for a small amount of random noise). Two instances of the same chain of subsequences are noticeable: `char_label_012` → `char_label_181` → `seeded_NA_rnd_walk_012` → `seeded_NA_rnd_walk_022` → `seeded_NA_rnd_walk_146` → `char_label_022`.

1.4 Structure of this work

In the following chapters, we derive how our *synthetic data stream generator (SDG)* generates its data. We systematically break down its main modules into UML 2.0 class diagrams, define which mathematical functions each module realizes and what parameters it takes. Note that even though the UML diagrams are closely guided by the actual code structure, they do not represent a complete technical documentation. Their main purpose is to facilitate an understanding of how the SDG works by providing a simplified view of our implementation.

We implemented SDG in R using the R6 class system [12]. It is available at [13].

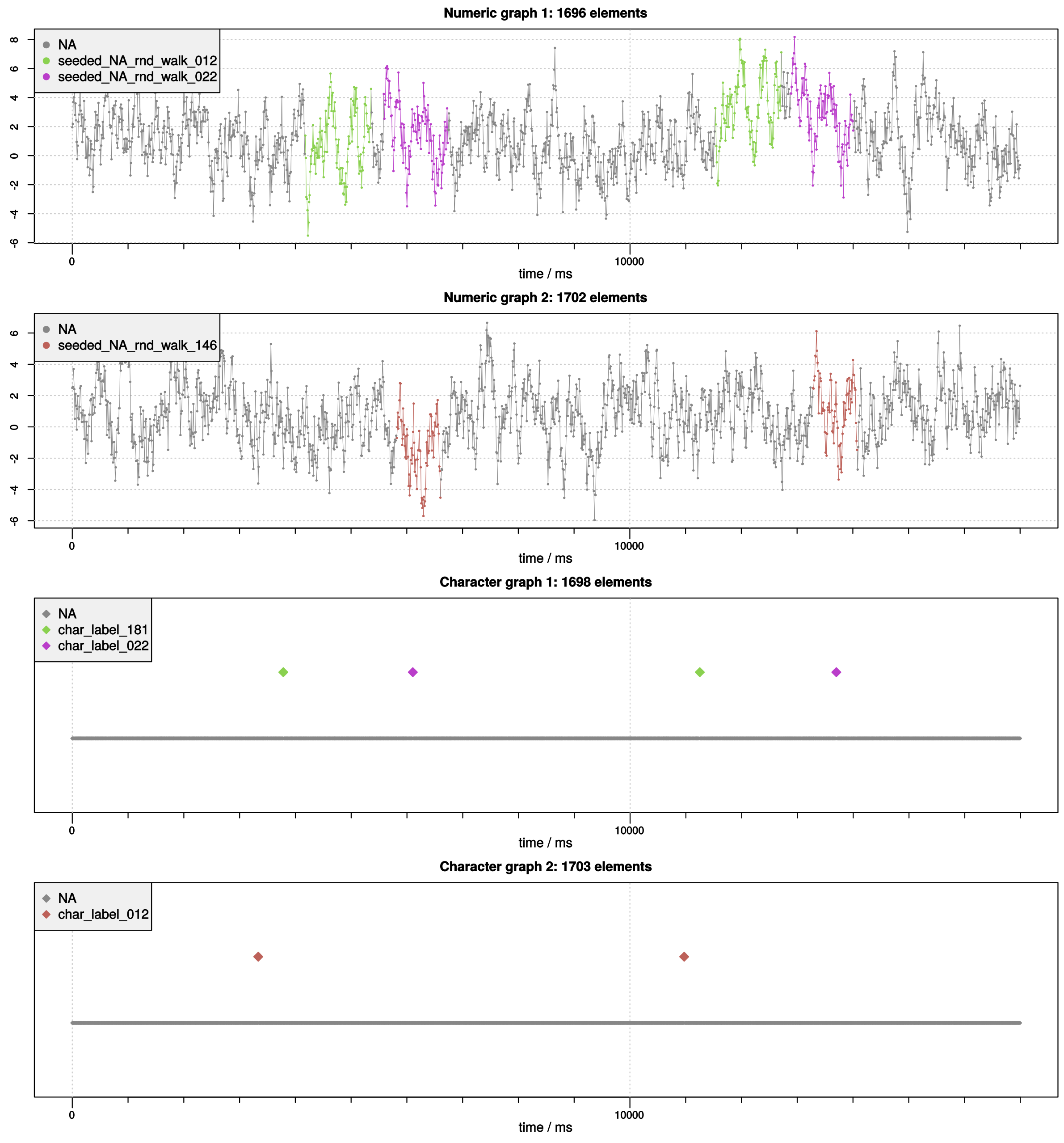


Figure 1.1: An example plot of the data generated by our synthetic data stream generator with a clearly visible repeating pattern across all data streams.

2 UML naming and color convention

Even though SDG is implemented in R, we use a JAVA like terminology throughout this chapter. Table 2.1 contains on overview of our naming convention and our diagrams use the color convention established in Table 2.2.

Notation:	Meaning:
`Abstract_*`	Represents abstract classes with at least one method that has to be overridden by subclasses. Note that the R6 class system does not syntactically support abstract classes (like, for example, JAVA does). This naming convention is therefore merely a semantic indicator on how a class is supposed to be used. Also remember that the names of abstract classes and methods are depicted in cursive in UML.
`.Classname`	Classnames with a dot prefix indicate that instances of said class are only used internally by instances of other classes and therefore should not be instantiated manually.
`TEMPLATE_PARAMETERS`	Template parameters (correspond to JAVA type parameters) are always written in upper case.
`{Constraint}`	Constraints are expressed in curly braces.
`Type<Template_Type>`	We either set template parameters via the first, JAVA like notation or by using the second, UML binding stereotype notation. We typically use the former to specify the type of member variables and the latter for the specification of a subclass type.
`≪bind T → Template_Type≫`	
Array indices start at 1.	

Table 2.1: Our UML naming convention.

Color:	Refers to:
Blue	Classes and packages related to numeric data generation and processing.
Yellow	Classes and packages related to string data generation and processing.
Green	Classes and packages handling numeric as well as string data.
Orange	Language model related classes and packages.
Grey	None of the above

Table 2.2: Our UML color convention.

3 Language model training and representation

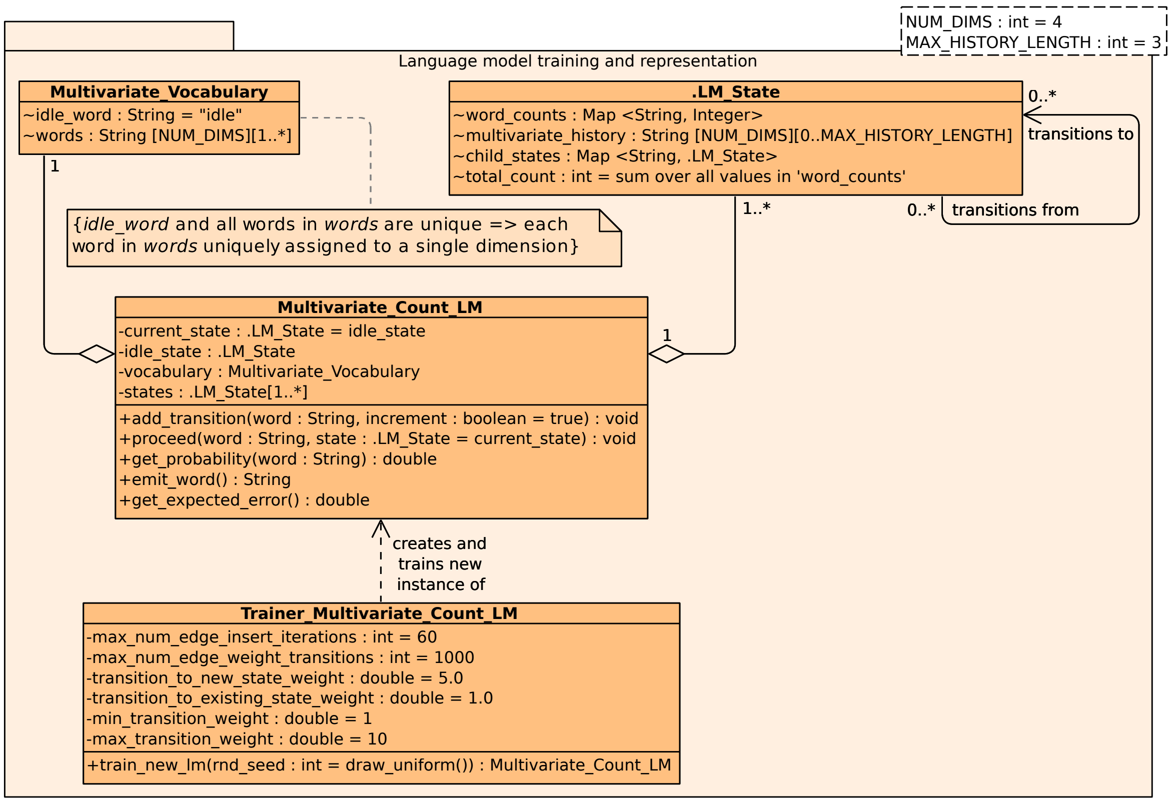

NUM_DIMS : int = 4
MAX_HISTORY_LENGTH : int = 3
Language model training and representation
Multivariate_Vocabulary
~idle_word : String = "idle"
~words : String [NUM_DIMS][1..*]
1
.LM_State
~word_counts : Map <String, Integer>
~multivariate_history : String [NUM_DIMS][0..MAX_HISTORY_LENGTH]
~child_states : Map <String, .LM_State>
~total_count : int = sum over all values in 'word_counts'
0..*
transitions to
1..*
0..*
transitions from
{idle_word and all words in words are unique => each word in words uniquely assigned to a single dimension}
Multivariate_Count_LM
-current_state : .LM_State = idle_state
-idle_state : .LM_State
-vocabulary : Multivariate_Vocabulary
-states : .LM_State[1..*]
+add_transition(word : String, increment : boolean = true) : void
+proceed(word : String, state : .LM_State = current_state) : void
+get_probability(word : String) : double
+emit_word() : String
+get_expected_error() : double
1
creates and trains new instance of
Trainer_Multivariate_Count_LM
-max_num_edge_insert_iterations : int = 60
-max_num_edge_weight_transitions : int = 1000
-transition_to_new_state_weight : double = 5.0
-transition_to_existing_state_weight : double = 1.0
-min_transition_weight : double = 1
-max_transition_weight : double = 10
+train_new_lm(rnd_seed : int = draw_uniform()) : Multivariate_Count_LM

Figure 3.1: UML 2.0 class diagram depicting language model related classes.

We begin by defining the main components of the multivariate language model (LM). An LM instance determines the order in which motifs and labels are emitted during the data generation process. Figure 3.1 contains an overview with the most important LM related classes.

3.1 Multivariate vocabulary

A `Multivariate_Vocabulary` stores a unique set of words for each dimension along with a special idle word:

Definition 3.1 (`Multivariate_Vocabulary`)**.** Let `NUM_DIMS` $\in \mathbb{N}$ refer to the number of dimensions, Ω^* be the set of all strings of finite length and `vocabulary` be a `Multivariate_Vocabulary` instance. `vocabulary` contains the following fields:

- `vocabulary.idle_word` $\in \Omega^*$: A special string used to delete the current language model history. Default value is "idle". Its exact semantics are defined in Definition 3.3.

- `vocabulary.words`: Contains the vocabularies of each dimension. Each vocabulary contains a unique and finite set of words which is not allowed to contain `vocabulary.idle_word`:

$$\begin{aligned} \texttt{vocabulary.words} &:= (W_1, \dots, W_{\texttt{NUM\_DIMS}}) \\ \forall i, j = 1, \dots, \texttt{NUM\_DIMS}\colon W_i &\subseteq \Omega^*, \\ & 1 \le |W_i| < \infty, \\ & \texttt{vocabulary.idle\_word} \notin W_i, \\ & i \neq j \Rightarrow W_i \cap W_j = \emptyset \end{aligned}$$

- Let $\mathcal{W}$ denote the union of all words in `vocabulary.words` and $\hat{\mathcal{W}} := \mathcal{W} \cup \{\texttt{vocabulary.idle\_word}\}$.
- Let $\forall w \in \mathcal{W}$, $d(w)$ denote the unique dimension this word belongs to (i.e., $d(w) = n \Leftrightarrow w \in \texttt{vocabulary.words}[n]$).

3.2 Multivariate histories as language model states

We begin by defining the *multivariate history* itself:

Definition 3.2 (Multivariate m-gram history)**.** Let $M := m - 1 := \texttt{MAX\_HISTORY\_LENGTH} \in \mathbb{N}$ be the maximal history length and, `vocabulary` be an instance of `Multivariate_Vocabulary` (Definition 3.1). We define:

- Multivariate history $\mathbf{h}$: For each dimension, the corresponding entry of $\mathbf{h}$ contains a vector of at most M words corresponding to the last words that occurred in this dimension:

$$\begin{aligned} & \mathbf{h} := (\mathbf{h}_1, \dots, \mathbf{h}_{\texttt{NUM\_DIMS}}) \\ \forall i = 1, \dots, \texttt{NUM\_DIMS}\colon & \mathbf{h}_i := (w_{1,i}, \dots, w_{N_i,i}) \\ & 0 \le |\mathbf{h}_i| := N_i \le M \\ & \forall j = 1, \dots, N_i\colon w_{j,i} \in W_i \end{aligned}$$

- Let $\mathbf{h}_\emptyset$ denote the empty history, i.e. the one where $N_i = 0$ for all dimensions i. We call this the "idle state".
- Let $\mathcal{H}$ denote the the set of all histories with maximal length M.

Next, we require a definition for *admissible transitions* between histories:

Definition 3.3 (History transitions)**.** We define:

- The transition function $t\colon \mathcal{H} \times \hat{\mathcal{W}} \to \mathcal{H}$ defines unique transitions between histories via words:

$$\begin{aligned} \forall \mathbf{h} \in \mathcal{H}, w \in \mathcal{W}\colon t(\mathbf{h}, w) &:= \mathbf{h}' = (\mathbf{h}'_1, \dots, \mathbf{h}'_{\texttt{NUM\_DIMS}}) \\ \forall i = 1, \dots, \texttt{NUM\_DIMS}\colon \mathbf{h}'_i &:= \begin{cases} \mathbf{h}_i & \text{if } i \neq d(w) \\ (w, w_{1,i}, \dots, w_{min(N_i, M-1),i}) & \text{if } i = d(w) \end{cases} \\ t(\mathbf{h}, \texttt{vocabulary.idle\_word}) &:= \mathbf{h}_\emptyset \end{aligned}$$

  For $w \in \mathcal{W}$, $\mathbf{h}'$ is therefore defined as the unique history resulting in appending w to the front of $\mathbf{h}_{d(w)}$ and removing the oldest entry if the resulting history becomes longer than the maximal history length M. The idle word, on the other hand, always result in the idle state $\mathbf{h}_\emptyset$.

- A *word chain* $\mathbf{p}_w = (w_1, \dots, w_L) \in \hat{\mathcal{W}}^L, L \ge 2$ is defined as a sequence of words corresponding to a path starting and ending at the idle state and not having the idle state as an intermediate state. Therefore, $\mathbf{p}_w = (w_1, \ldots, w_L)$ is a word chain if and only if all of the following is true:

$$\begin{aligned} L &\geq 2 \\ w_L &\in \hat{\mathcal{W}} \setminus \mathcal{W} = \{\texttt{vocabulary.idle\_word}\} \\ \forall i = 1, \ldots, L-1\colon w_i &\in \mathcal{W} \end{aligned}$$

Histories may be seen as states in an automaton with only a subset of possible transitions actually allowed. Our implementation stores all the information considering a state and its outgoing transitions in an object of type `.LM_State`:

Definition 3.4 (`.LM_State`)**.**
Let `NUM_DIMS` $\in \mathbb{N}$ refer to the number of dimensions, $M :=$ `MAX_HISTORY_LENGTH` $\in \mathbb{N}$ be the maximal history length and, `state` be an instance of `.LM_State`. `state` contains the following fields:

- `state.multivariate_histoy`: A constant field containing the history $\mathbf{h} \in \mathcal{H}$ which this `.LM_State` instance uniquely represents.
- `state.word_counts`: A map assigning the number of transitions $t(\mathbf{h}, w)$ observed during training to each word w. This field is later used to estimate the corresponding transition probability. The key set `state.word_counts.keySet()` of `state.word_counts` only contains words w for which said number is ≥ 1.
- `state.child_states`: A map assigning the corresponding subsequent history state $\mathbf{h}' = t(\mathbf{h}, w)$ to each word w in the key set of `state.word_counts`
(by definition: $w \in$ `state.word_counts.keySet()` $\Leftrightarrow$ `state.word_counts[w]` > 0).
- `total_count`: Sum over all values `state.word_counts.values()` in `state.word_counts`.

3.3 Language model

A `Multivariate_Count_LM` instance represents a probability distribution for the next word given a multivariate history over words that have been encountered so far in each dimension. We developed the following multivariate generalization of regular m-gram models (for the latter, see, for example, [14, 15]):

Definition 3.5 (Multivariate m-gram language model)**.** We define:

- Let $w^{(n)} \in \hat{\mathcal{W}}$ denote the n-th emitted word in a sequence of words and $\mathbf{h}^{(n)}$ be the resulting history (i.e., $\mathbf{h}^{(0)} := \mathbf{h}_\emptyset$ and for $m = 1, \ldots, n$: $\mathbf{h}^{(m)} := t(\mathbf{h}^{(m-1)}, w^{(m)})$)
- The language model $p\colon \hat{\mathcal{W}} \times \mathcal{H} \to [0, 1]$ is defined as the following probability distribution:

$$\forall \mathbf{h}^{(n)} \in \mathcal{H}, w \in \hat{\mathcal{W}}\colon p(w|\mathbf{h}^{(n)}) := P(w^{(n+1)} = w|\mathbf{h}^{(n)})$$

  In other words: $p(w|\mathbf{h}^{(n)})$ is the probability that the $(n+1)$-th emitted word will be w, given the current history $\mathbf{h}^{(n)}$.

Methods to represent a language model vary from simple count based models up to modern LSTM based ones [16, 17]. For our `Multivariate_Count_LM`, we chose the former due to its simplicity and fast training. The latter requires only a single run over the training corpus in order to generate the count statistics. We also postpone the application of m-gram smoothing techniques to potential future work. Our resulting `Multivariate_Count_LM` class is defined as follows:

Definition 3.6 (`Multivariate_Count_LM`)**.** Let `lm` be an instance of `Multivariate_Count_LM`. `lm` contains the following fields and methods:

- `lm.vocabulary`: The `Multivariate_Vocabulary` instance this LM is based on.
- `lm.current_state`: The current history state $\mathbf{h} \in \mathcal{H}$ this LM is currently in.
- `lm.idle_state`: A reference to the `.LM_State` instance representing $\mathbf{h}_\emptyset$.
- `lm.states`: A set of all history states encountered so far during the LM training.

- `lm.add_transition(word, increment)`: Adds a transition from the current LM state to the one reached via the provided word. Updates transition counts if `increment` is `true` (if not provided, assumed to be `true` by default):

```
1. If( word ∉ lm.current_state.child_states.keySet() ):
   1.1. Create a new instance child_state of .LM_State representing $\mathbf{h}' := t(\mathbf{h}, \texttt{word})$;
   1.2. lm.current_state.child_states.put(word, child_state);
   1.3. lm.current_state.word_counts.put(word, 0);
2. If( increment ):
   2.1. lm.current_state.word_counts[word]++;
   2.2. lm.current_state.total_counts++;
3. lm.current_state = child_state;
```

- `lm.proceed(word, state)`: Moves to the state reached via `word` from `state` (if the latter is not provided, it is set to `current_state` by default):

```
1. If(word ∉ state.child_states.keySet() ):  Error();
2. Else:  lm.current_state = lm.current_state.child_states[word]
```

- `lm.get_probability(word)`: Returns $p(\texttt{word}|\mathbf{h})$ computed as:

```
1. If( lm.current_state.child_states.size() == 0 ):
       return $\frac{1}{\texttt{total # of words in lm.vocabulary}}$;
2. Elseif( word ∉ lm.current_state.child_states.keySet() ):  return 0.0;
3. Else:  return $\frac{\texttt{lm.current\_state.word\_counts[word]}}{lm.current\_state.total\_counts}$
```

- `lm.emit_word()`: Samples the next word $w \in \mathcal{W}$ according to the current state's probability distribution and proceeds to appropriate next state:

```
1. next_word = sample(from = lm.current_state.word_counts.keySet(),
                      weights = lm.get_probability(·));
2. lm.proceed(next_word);
3. return next_word;
```

- `lm.get_expected_error()`: Returns the inherent prediction error of this language model (as defined in the following subsection's Definition 3.8).

3.3.1 Language model error

Assume we have a language `Multivariate_Count_LM` instance `lm` which produces data by randomly emitting words according to its internal probability distribution $p(w|\mathbf{h})$. Further assume that we want to train a predictor based on this data, i.e. a function $g\colon \mathcal{H} \to \hat{\mathcal{W}}, g(\mathbf{h}) = w$ which attempts to predict the next word w given the word history $\mathbf{h}$. If the data is distributed with resprect to $p(w|\mathbf{h})$ and the current history is $\mathbf{h}$, the expected next prediction error our predictor makes is:

$$\sum_{w \in \hat{\mathcal{W}} \setminus \{g(\mathbf{h})\}} p(w|\mathbf{h}) = 1 - p(g(\mathbf{h})|\mathbf{h}))$$

This term is minimized by:

$$\min_{g(\mathbf{h})} [1 - p(g(\mathbf{h})|\mathbf{h}))] = 1 - \max_{w \in \mathcal{W}} p(w|\mathbf{h})$$

This is the minimal expected error every predictor g can make in state $\mathbf{h}$. Given we knew an a priori distribution $p_a(\mathbf{h})$ over all history states (i.e., the expected probability of being in state $\mathbf{h}$ at any given time), we can define the *minimal expected language model error* inherent to `lm` as:

$$\sum_{\mathbf{h} \in \mathcal{H}} p_a(\mathbf{h})(1 - \max_{w \in \mathcal{W}} p(w|\mathbf{h})))$$

In the theory of *Markov Chains*, such a distribution $p_a(\mathbf{h})$ is called a *stationary distribution* [18] and is defined as follows:

Definition 3.7. Let `lm` be an instance of `Multivariate_Count_LM` and $p(w|\mathbf{h})$ be the probability distribution `lm` represents. We define:

- Let $N := lm.states.size()$ be the (finite) number of states in `lm` and $\mathbf{h}_{(1)}, \dots, \mathbf{h}_{(N)}$ be an arbitrary ordering of those states.
- Transition matrix $P \in [0,1]^{N \times N}$: $P := (p_{i,j})_{i,j=1,\dots,N}, p_{i,j} :=$ transition probability from state $\mathbf{h}_{(i)}$ to $\mathbf{h}_{(j)}$.
- A stationary distribution $\mathbf{p}_a := (p_a(\mathbf{h}_{(i)}))_{i=1,\dots,N}$ is defined as the distribution, for which the following holds:

$$\mathbf{p}_a = P^t \mathbf{p}_a$$

Note that given any initial state distribution $\mathbf{p}_0 = (p_0(\mathbf{h}_{(i)}))_{i=1,\dots,N}$, $\mathbf{p}_1 := P^t \mathbf{p}_0$ is the state probability distribution after one transition. The stationary distribution is therefore the distribution which is invariant to transition. It can be shown [18] that if P is of finite dimension (i.e. $N < \infty$) and irreducible (meaning that the graph induced by P is strongly connected[1], i.e., all states are reachable from all states with a non-zero probability.) then a unique stationary distribution $\mathbf{p}_a$ exists which satisfies $\mathbf{p}_a = P^t \mathbf{p}_a$. The solution is therefore the appropriately scaled, unique eigenvector of P^t to the eigenvalue 1 (which, as can be shown [18], is also the largest eigenvalue of any transition matrix P).

So as long the graph induced by our language model is strongly connected (which, as we will see in the next section, our language model trainer ensures), we may obtain our *minimal expected language model error* as:

Definition 3.8. Let `lm` be an instance of `Multivariate_Count_LM` and P be its corresponding transition matrix. Compute `lm.get_expected_error()` as follows:

```
1. If( $P$ not irreducible ): Error();
2. Else:
   2.1. Compute eigenvector $\mathbf{p}_a$ corresponding to largest eigenvalue of $P^t$ and scale it by a factor
        $c \in \mathbb{R}$ so that it contains only non-negative entries summing up to 1;
   2.2. return $\sum_{\mathbf{h} \in \mathcal{H}} p_a(\mathbf{h})(1 - \max_{w \in \mathcal{W}} p(w|\mathbf{h}))$;
```

Note: In practice, we only need to include those states in P and p_a which are actually present in `lm.states` in order to compute `lm.get_expected_error()`.

3.4 Language model trainer

Now that we have laid out the formal framework for our language model, we may proceed to its actual generation. An instance of `Trainer_Multivariate_Count_LM` generates a `Multivariate_Count_LM` instance based on a number of parameters. The latter are listed in Figure 3.1 together with their default values. Since the training is probabilistic in nature, a random seed may be provided to the `train_new_lm(rnd_seed)` method. Given the same set of parameters and the same seed, an instance of `Trainer_Multivariate_Count_LM` will always generate the same language model.

The training procedure is listed in Algorithm 3.1:

Algorithm 3.1 (`train_new_lm(rnd_seed)`).
Let `trainer` be an instance of `Trainer_Multivariate_Count_LM`.

```
trainer.train_new_lm(rnd_seed):
  1. Store current random generator state in $s$;
  2. Set random generator seed to rnd_seed;
```

[1] Our history state based language model can be represented by a graph where vertices correspond to histories $\in \mathcal{H}$, edges corresponds to transitions according to the transition function t, edge weights are set to the corresponding transition probability and zero-weight edges are excluded.

3. Create an empty `Multivariate_Count_LM` instance `lm`;
4. `edge_insertion_phase(lm); //` Algorithm 3.2
5. `edge_weight_phase(lm); //` Algorithm 3.3
6. Restore random generator state s;

It consists of two phases:

1. Edge insertion phase (Algorithm 3.2): Starting with an an empty `Multivariate_Count_LM` instance `lm` (i.e., one where `lm.states` contains only `lm.idle_state`), the trainer iteratively adds new states and state transitions into `lm`. The the number of iterations in this phase is controlled by the parameter `trainer.max_num_edge_insert_iterations` and thereby also the size of the resulting model.

   Let $W_{\mathbf{new}}$ be the set of words leading to an unseen state given `lm`'s current state. In each iteration, the trainer either proceeds to an unseen state via some word from W_{new} or returns to the idle state via the idle word. In each iteration, the former is done with a probability proportional to the current `weight_new` value and the latter with a probability proportonal to 1. `weight_new` parameter is reset to `trainer.transition_to_existing_state_weight` each time the trainer returns to the idle state. This trainer parameter therefore controls the average length of word chains (see Definition 3.3) produced by the resulting language model.

   Each time `lm` returns to the idle state, the trainer proceeds to a random state with a probability proportional to `trainer.transition_to_existing_state_weight` or remains at the idle state with a probability proportional to 1. The loop then continues as before. This procedure creates branches in the underlying language model graph and thereby makes the model more ambiguous by increasing its minimal expected error. Note that in order to keep the average word chain length constant, we need to reduce `weight_new` if we actually jumped to a random state (and restore its original value once we reach the idle state again). This is done in line 2.1.3. of Algorithm 3.2.

   If `lm` is not in the idle state after the main loop finished, the trainer adds a last transition to that state. This and the fact that the trainer makes sure that each state's transition word counts contain only 1 (the `increment` argument of `add_transition(...)` is set to `true` if and only if the corresponding edge does not exist) results in the underlying graph being strongly connected. Also note that the trainer ensures that there are no loops in the underlying graph - no idle state loops in particular.

2. Edge weight phase (Algorithm 3.3): Given the `Multivariate_Count_LM` instance `lm` from the last phase, it is guaranteed to have the following properties at this point:

   - `lm` is in the idle state.
   - The underlying graph of `lm` is strongly connected.
   - All word counts are 1, i.e.: $\forall$`state` $\in$ `lm.states`: $\forall w \in$ `state.word_counts.keySet()`: `state.word_counts`$[w] = 1$

   The edge weight phase now assigns probabilities to each transition. It does so by first uniformly drawing a random weight from the interval

   $$[\texttt{trainer.min\_transition\_weight,trainer.max\_transition\_weight}]$$

   for each combination of state and child state. The trainer then performs a random walk through the underlying language model graph and increments the corresponding count statistics after each step. In each iteration, the next state to transition to is randomly drawn from the set of child states and each child is chosen with a probability proportional to the corresponding weight generated at the beginning of the edge weight phase. The number of iterations in this phase is controlled via the `trainer.max_num_edge_weight_transitions` parameter.

Algorithm 3.2 (`edge_insertion_phase(lm)`)**.**
This is a subroutine of Algorithm 3.1.

`edge_insertion_phase(lm)`:

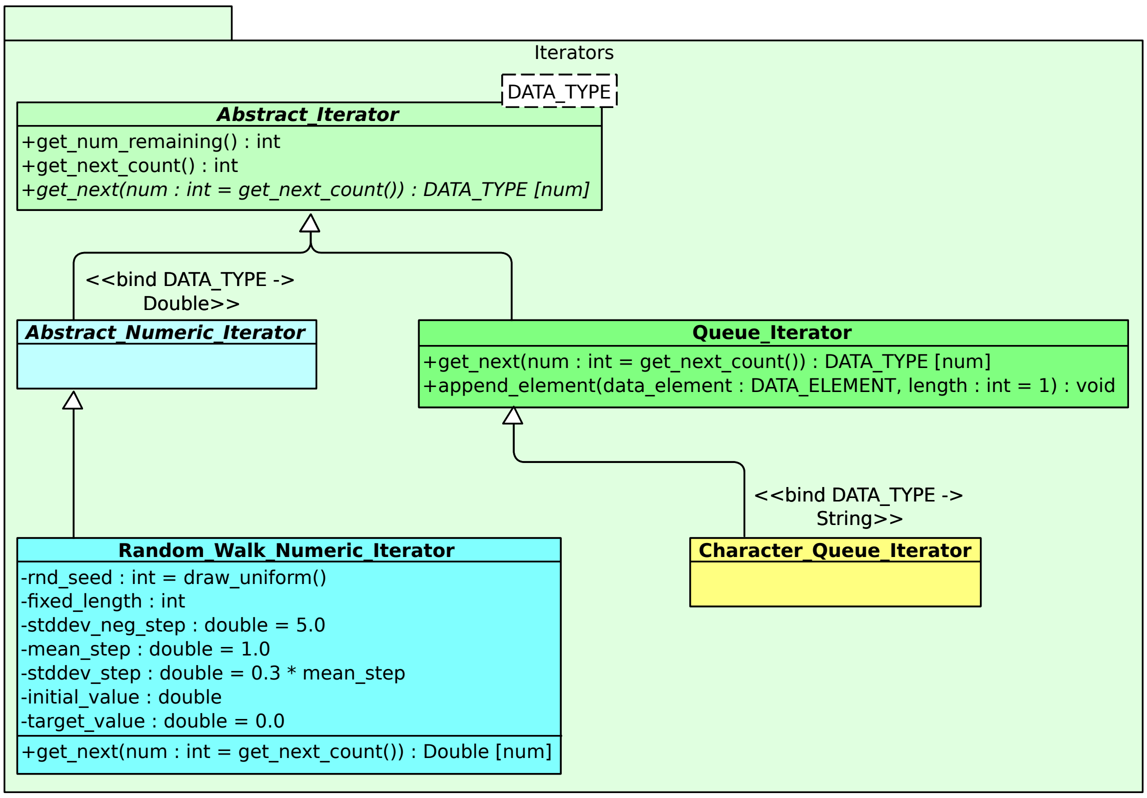
```
1. weight_new = trainer.transition_to_new_state_weight;
   weight_exist = trainer.transition_to_existing_state_weight;
2. For( it = 0; it < trainer.max_num_edge_insert_iterations; it++ ):
  2.1. If( lm.current_state == lm.idle_state &&
             sample(from = (true, false), weights = (weight_exist ,1)) ):
    2.1.1. lm.current_state = sample(from = lm.states,
                                     weights = $(1, \ldots, 1)$);
    2.1.2. Compute distance $d$ of shortest directed path
           from lm.current_state to lm.idle_state;
    2.1.3. weight_new = weight_new - min(weight_new, d);
  2.2. Let $W_{\mathtt{new}}$ be the set of words $w$ for which lm.add_transition($w$) would result in a new state
       being created;
  2.3. If( $|W_{\mathtt{new}}| == 0$ ):
    2.3.1. If( lm.current_state == lm.Idle_state ):  break;
    2.3.2. Else:  action = "idle";
  2.4. Elseif( lm.current_state == lm.idle_state):  action = "new";
  2.5. Else:  action = sample(from = ("idle", "new"),
                              weights = (1, weight_new);
  2.6. If( action == "idle"):

    2.6.1. lm.add_transition(word = lm.vocabulary.idle_word,
                        increment = lm.vocabulary.idle_word $\notin$
                                    lm.current_state.child_states.keySet());
    2.6.2. weight_new = trainer.transition_to_new_state_weight;
  2.7. Else:  lm.add_transition(sample(from = $W_{\mathtt{new}}$, weights = $(1, \ldots, 1)$));
3. If( lm.current_state $\neq$ lm.idle_state):
   lm.add_transition(word = lm.vocabulary.idle_word,
                increment = lm.vocabulary.idle_word $\notin$
                            lm.current_state.child_states.keySet());
```

Algorithm 3.3 (edge_weight_phase(lm)).
This is a subroutine of Algorithm 3.1.

```
edge_weight_phase(lm):
1. For( state $\in$ lm.states):
  1.1. For( word $\in$ state.child_states.keySet() ):
       weight(state, word) = draw_uniform(from = [trainer.min_transition_weight,
                                                  trainer.max_transition_weight]);
2. For( it = 0; it < trainer.max_num_edge_weight_transitions; it++ ):
  2.1. lm.add_transition(sample(from = lm.current_state.child_states.keySet(),
                                weights = weight(lm.current_state, $\cdot$));
```

4 Iterators



Iterators
DATA_TYPE
Abstract_Iterator
+get_num_remaining() : int
+get_next_count() : int
+get_next(num : int = get_next_count()) : DATA_TYPE [num]
<<bind DATA_TYPE -> Double>>
Abstract_Numeric_Iterator
Queue_Iterator
+get_next(num : int = get_next_count()) : DATA_TYPE [num]
+append_element(data_element : DATA_ELEMENT, length : int = 1) : void
<<bind DATA_TYPE -> String>>
Random_Walk_Numeric_Iterator
-rnd_seed : int = draw_uniform()
-fixed_length : int
-stddev_neg_step : double = 5.0
-mean_step : double = 1.0
-stddev_step : double = 0.3 * mean_step
-initial_value : double
-target_value : double = 0.0
+get_next(num : int = get_next_count()) : Double [num]
Character_Queue_Iterator

Figure 4.1: UML 2.0 class diagram depicting the basic value generating classes.

Our whole data generation process revolves around *Iterators*. The iterator concept we use is similar to JAVA and C++ iterators. Their main purpose is to iterate over some data. At the topmost level is the `Abstract_Iterator` class which serves as a common interface for all other iterators. At this level, we do not specify where the data actually originates from. An `Abstract_Iterator` subclass might read the data from some file, a remote source or, as is the case with our SDG, generate it itself. Figure 4.1 contains an overview over the most basic iterators and we now continue to specify their behavior.

`Abstract_Iterator` class is the common superclass of all iterators:

Definition 4.1 (`Abstract_Iterator`)**.** Let `it` be an instance of `Abstract_Iterator`. `it` contains the following methods:

- `DATA_TYPE`: The type of data returned by `it.get_next(num)`.
- `it.get_num_remaining()`: The number of elements this iterator still has left to iterate over. Note that this does not necessarily mean that this number of elements is immediately retrievable, only that it will at some point in the future. In order to check how many elements are *immediately* retrievable, use `get_next_count()`.
- `it.get_next_count()`: The number of currently available elements which can be safely retrieved via `it.get_next(num)`. Unless overridden, `it.get_next_count()` returns the same

value as `it.get_num_remaining()`.

- `it.get_next(num)`: Must be overridden by subclasses. May only be called with a `num` argument that does not exceed `it.get_next_count()` (otherwise the behavior is unspecified and may result in an error). Must return a collection of `num` data elements which are instances of `DATA_TYPE`.

We define `Abstract_Numeric_Iterator` to be the common class for numeric iterators. The only difference to `Abstract_Iterator` is that we set the `DATA_TYPE` template parameter to `Double`.

4.1 Random walk iterator

The SDG generates its numeric data based on random walks (although it can easily be configured to use other generating functions). Given some initial value x_0, the most basic form of random walks generates the n-th value as $x_n := x_{n-1} + z_n$ where z_n is uniformly drawn from $\{-1, 1\}$. For the variance of x_n, we have $Var(x_n) = n$. This means that the expected radius of a random walk of length N (i.e., the difference between the maximal and the minimal value occurring on the random walk), increases with $\sqrt{N}$. For our SDG, however, we prefer a random walk where the values stay within a certain range around a predefined target value - just as depicted in the example graph in Figure 1.1. We therefore define a generalized form of random walks where the probability to take a step towards the target value increases with the distance to it. Our sequences are generated by instances of the `Random_Walk_Numeric_Iterator` class: `Random_Walk_Numeric_Iterator` class is the common superclass of all iterators:

Definition 4.2 (`Random_Walk_Numeric_Iterator`).
Let `rnd_it` be an instance of `Random_Walk_Numeric_Iterator`. Let $N \in \mathbb{N}_0$ be the number of elements retrieved via `rnd_it.get_next(num)` so far and let $x_n \in \mathbb{R}$ denote the n-th retrieved value. `rnd_it` contains the following fields and methods:

- `rnd_it.rnd_seed`: Either manually or randomly (default) set during instantiation of `rnd_it`.
- `rnd_it.fixed_length`: Either set to ∞ or a value $\in \mathbb{N}$. Indicates the length of the random walk this instance generates. Note that $N \leq$ `rnd_it.fixed_length`.
- `rnd_it.get_num_remaining(): return rnd_it.fixed_length`$-N$;
- `rnd_it.get_next(num)`:

```
1. If( $num == 0$ ):  return;
2. If( $N == 0$ ):
   2.1. Set random generator seed to rnd_it.rnd_seed;
   2.2. $x_1 :=$ rnd_it.initial_value;
   2.3. $n_{\text{start}} = 2$;
3. Else:
   3.1. Restore random generator state $s$ from the end of the last call to it.get_next(num);
   3.2. $n_{\text{start}} = N + 1$;
4. For($n = n_{\text{start}}$; $n < N +$ num; $n$++):
   4.1. $p_n^- :=$ cumulative_norm($x_{n-1}$,
                          mean = rnd_it.target_value,
                          stddev = rnd_it.stddev_neg_step);
        cumulative_norm(...) returns the value of the cumulative distribution function of a
        normal distribution with the indicated mean and standard deviation. Note that $p_n^-$ is
        0.5 if $x_{n-1}$ equals the target value and converges to 1(0) as $x_{n-1}$ goes to $+\infty(-\infty)$.
   4.2. $sign_n :=$sample(from = $(0, 1)$, weights = $(1 - p_n^-, p_n^-)$);
   4.3. $\delta_n :=$draw_norm(mean = rnd_it.mean_step, stddev = rnd_it.stddev_step);
        draw_norm(...) is the step size drawn from a normal distribution with the denoted
        mean and standard deviation.
   4.4. $x_n := x_{n-1} + (-1)^{sign_n} \delta_n$;
```

```
5. Store current random generator state in $s$;
6. Restore old random generator state;
7. return $(x_{N+1}, \ldots, x_{N+\texttt{num}})$;
```

The basic random walk described earlier is actually a special case of the definition above which can be obtained by setting `rnd_it.stddev_neg_step` $= \infty$, `rnd_it.mean_step` $= 1$ and `rnd_stddev_step` $= 0$.

4.2 Queue iterator and character queue iterator

The `Queue_Iterator` is a subclass of `Abstract_Iterator` which, as the name suggests, realizes a queue:

Definition 4.3 (`Queue_Iterator`)**.**
Let `q_it` be an instance of `Queue_Iterator`. `q_it` contains the following fields and methods:

- `q_it.append_data_element(data_element, length)`:
  Adds the provided `data_element <length>` times to the back of an internal queue. If not provided, `length` is set to 1 by default. Note that the class is implemented in a way such that the `data_element` object is not copied `<length>` times but rather stored once together with a counter. The latter is initially set to `length` and then decremented each time the corresponding data element is retrieved. As soon as the counter hits zero, its associated data element is removed from the queue.
- `q_it.get_num_remaining()`: Returns the number of elements in the internal queue.
- `q_it.get_next(num)`: Removes and returns `<num>` elements from the front of the internal queue.

SDG's label streams are internally represented via the `Character_Queue_Iterator` instances. The latter is a subclass of `Queue_Iterator` which restricts the latter's `DATA_TYPE` to strings.

5 Iterator decorators

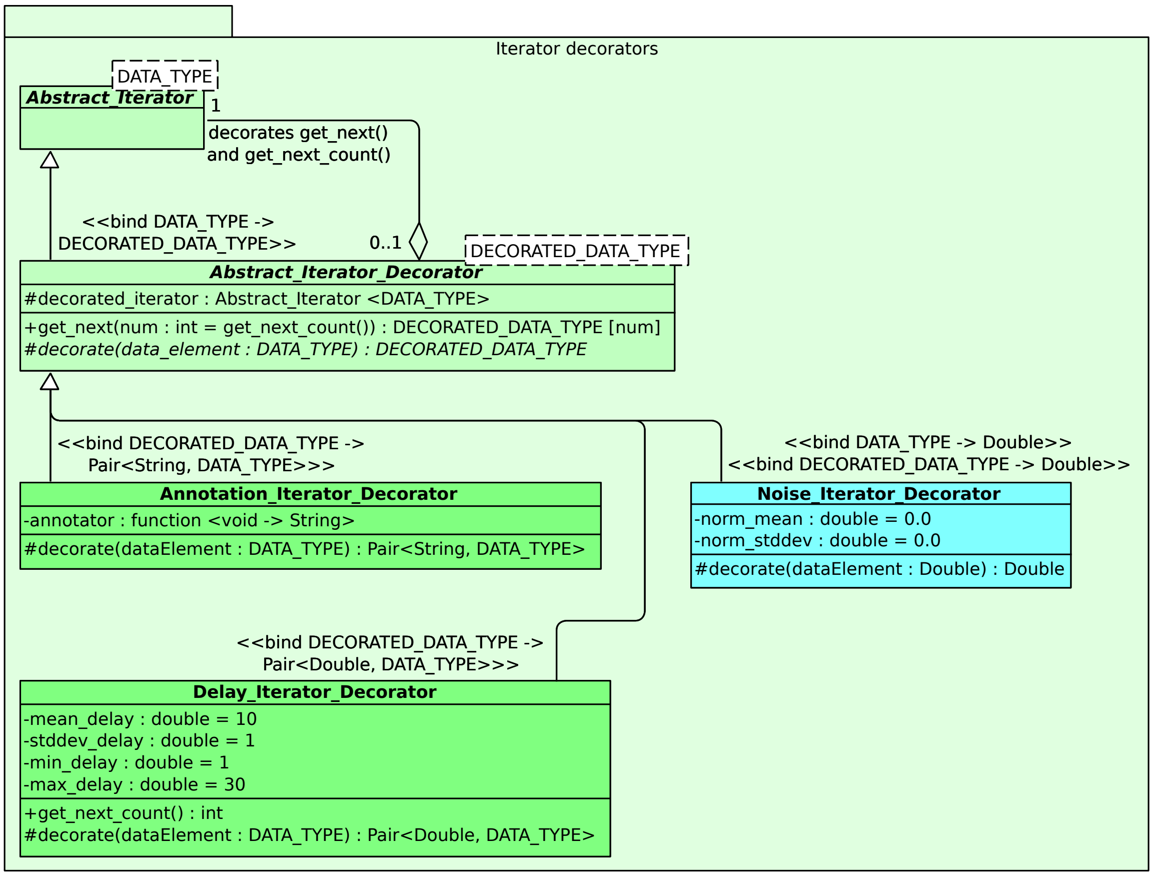

Iterator decorators
DATA_TYPE
Abstract_Iterator
1
decorates get_next() and get_next_count()
<<bind DATA_TYPE -> DECORATED_DATA_TYPE>>
0..1
DECORATED_DATA_TYPE
Abstract_Iterator_Decorator
#decorated_iterator : Abstract_Iterator <DATA_TYPE>
+get_next(num : int = get_next_count()) : DECORATED_DATA_TYPE [num]
#decorate(data_element : DATA_TYPE) : DECORATED_DATA_TYPE
<<bind DECORATED_DATA_TYPE -> Pair<String, DATA_TYPE>>>
<<bind DATA_TYPE -> Double>>
<<bind DECORATED_DATA_TYPE -> Double>>
Annotation_Iterator_Decorator
-annotator : function <void -> String>
#decorate(dataElement : DATA_TYPE) : Pair<String, DATA_TYPE>
Noise_Iterator_Decorator
-norm_mean : double = 0.0
-norm_stddev : double = 0.0
#decorate(dataElement : Double) : Double
<<bind DECORATED_DATA_TYPE -> Pair<Double, DATA_TYPE>>>
Delay_Iterator_Decorator
-mean_delay : double = 10
-stddev_delay : double = 1
-min_delay : double = 1
-max_delay : double = 30
+get_next_count() : int
#decorate(dataElement : DATA_TYPE) : Pair<Double, DATA_TYPE>

Figure 5.1: UML 2.0 class diagram depicting classes for the decoration of iterators.

SDG makes use of a special type of iterators the purpose of which is to encapsulate other iterators and change their outward behavior. This concept is known as the *decorator pattern* [19] which is why we call the common superclass of all decorators `Abstract_Iterator_Decorator`. Figure 5.1 contains an overview over the relevant decorator classes.

`Abstract_Iterator_Decorator` is itself a subclass of `Abstract_Iterator` which encapsulates an instance of `Abstract_Iterator`, retrieves the former's elements of type `DATA_TYPE` and decorates them resulting in the decorator's output data type `DECORATED_DATA_TYPE`:

Definition 5.1 (`Abstract_Decorator_Iterator`).
Let `dec_it` be an instance of `Abstract_Decorator_Iterator`. `dec_it` contains the following fields and methods:

- `dec_it.decorated_iterator`: The `Abstract_Iterator<DATA_TYPE>` instance which `d_it` decorates.
- `dec_it.get_num_remaining()`:

`return dec_it.decorated_iterator.get_num_remaining();`

- `dec_it.get_next_count()`:
  `return dec_it.decorated_iterator.get_next_count();`
- `dec_it.get_next(num)`: Performs the following operation:
  1. `undecorated_data = dec_it.decorated_iterator.get_next(num);`
  2. `For(` $n = 1$`;` $n \leq$ `undecorated_data.size();` n`++):`
     `decorated_data[n] = dec_it.decorate(undecorated_data[n]);`
  3. `return decorated_data;`
- `dec_it.decorate(data_element)`: Must be overridden by subclasses. Decorates a data element of type `DATA_TYPE` resulting in an object of type `DECORATED_DATA_ELEMENT`. The concrete decoration semantics are defined by overriding subclasses.

In the following subsections, we present the decorator implementations used by the SDG.

5.1 Annotation decorator

The `Annotation_Iterator_Decorator` employs a user provided function `annotate()` in order to decorate the data returned by the internal decorator with some string labels. As we will see later on, SDG uses an `annotate()` implementation which annotates the numeric data with ground truth labels about whether a data point is part of a motif or not:

Definition 5.2 (`Annotation_Iterator_Decorator`).
Let `a_it` be an instance of `Annotation_Iterator_Decorator`. `a_it` contains the following fields and methods:

- `DECORATED_DATA_TYPE`: Set to `Pair<String, DATA_TYPE>`, which represents a tuple the left entry of which is a string and the right entry of which is of the data type returned by the decorated iterator.
- `a_it.annotator`: A user provided parameter free method which upon each call returns some string label (its concrete implementation might possibly use some internal fields in order to determine which labels to return).
- `a_it.decorate(data_element): return Pair.of(annotate(), data_element);`

5.2 Noise decorator

The `Noise_Iterator_Decorator` is only applicable to iterators returning numeric data. As the name suggests, it adds some normally distributed noise to each data value based on the internal parameters (note that unless explicitly specified, the noise is always zero due to the default mean and standard deviation parameters - see Figure 5.1):

Definition 5.3 (`Noise_Iterator_Decorator`).
Let `n_it` be an instance of `Noise_Iterator_Decorator`. `n_it` contains the following fields and methods:

- `DATA_TYPE` and `DECORATED_DATA_TYPE`: Both set to `Double`. This decorator is only applicable to numeric iterators and its decorated values are also numeric.
- `n_it.decorate(data_element)`:
  ```
  return data_element +
          draw_norm(mean = n_it.norm_mean, stddev = n_it.norm_stddev);
  ```

5.3 Delay decorator

The `Delay_Iterator_Decorator` adds a delay to the accessibility of each data element retrieved from its internal iterator. It also annotates each data element with the time (in milliseconds) after which it became retrievable, starting at zero. In context of the SDG, this class realizes the asynchronicity demanded in Section 1.3:

Definition 5.4 (`Delay_Iterator_Decorator`).
Let `d_it` be an instance of `Delay_Iterator_Decorator`. Let $N \in \mathbb{N}_0$ be the number of elements retrieved via `d_it.get_next(num)` so far and let $x_n \in \mathbb{R}$ denote the n-th retrieved value. Furthermore, let δ_n and t_n denote the delay and the timestamp assigned to x_n and T denote the current system time. `d_it` implements `decorate(data_element)` and overrides `d_it.get_next_count()` as follows:

- `DECORATED_DATA_TYPE`: Set to `Pair<Double, DATA_TYPE>`, which represents a tuple the left entry of which is a double (containing the element's timestamp) and the right entry of which is of the data type of the elements returned by the decorated iterator.
- $\delta_n := \max\{$`d_it.min_delay`,
  $\min\{$`d_it.max_delay`,
  `draw_norm(mean = d_it.mean_delay,`
  `stddev = d_it.stddev_delay)`$\}\}$
  δ_n are independently generated delays for each data element x_n.
- $t_1 := \delta_1 + \min\{$system time of first call to `get_next(num)`,
  system time of first call to `get_next_count()`$\}$
  Elements start to become available only after the first call to either one of those methods.
- For $n > 1$: $t_n := t_{n-1} + \delta_n$
  Data element x_n becomes retrievable δ_n milliseconds after x_{n-1}.
- `d_it.get_num_remaining()`: The method ensures that only those elements are available where $t_n \leq T$, but never more than the internal decorated iterator can actually provide:
  1. $c =$ `decorated_iterator.get_next_count();`
  2. $n_{\max} = \max\limits_{\substack{n \in \{N,\dots,N+c\} \\ t_n \leq T}} n$;
  3. `return` $n_{\texttt{max}} - N$;
- `d_it.decorate(`x_n`): return Pair.of(`$t_n - t_1$`, `x_n`);`

6 Iterator concatenation

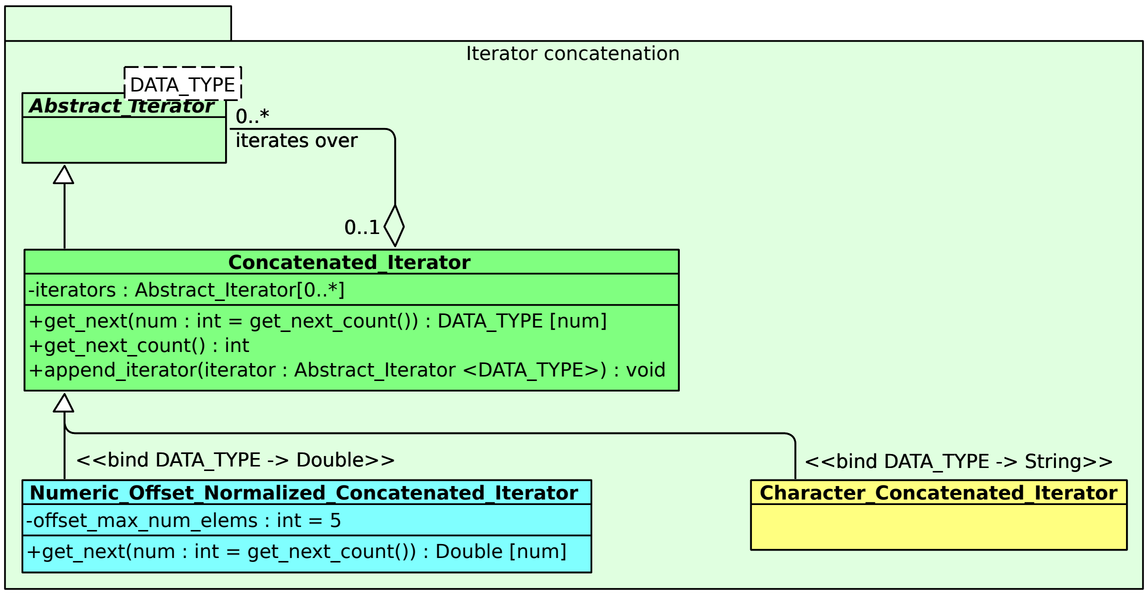

Iterator concatenation
DATA_TYPE
Abstract_Iterator
0..*
iterates over
0..1
Concatenated_Iterator
-iterators : Abstract_Iterator[0..*]
+get_next(num : int = get_next_count()) : DATA_TYPE [num]
+get_next_count() : int
+append_iterator(iterator : Abstract_Iterator <DATA_TYPE>) : void
<<bind DATA_TYPE -> Double>>
<<bind DATA_TYPE -> String>>
Numeric_Offset_Normalized_Concatenated_Iterator
-offset_max_num_elems : int = 5
+get_next(num : int = get_next_count()) : Double [num]
Character_Concatenated_Iterator

Figure 6.1: UML 2.0 class diagram depicting classes for the concatenation of iterators.

While instances of `Random_Walk_Iterator_Iterator` and `Character_Queue_Iterator` are employed by SDG to generate finite numeric and string subsequences, instances of `Concetenated_Iterator` are used for seamless iteration over the data from multiple consecutive iterators. Any number of iterators can be appended to a `Concatenated_Iterator` instance and the latter will iterate over the data from each iterator in the order the latter were added. Figure 6.1 contains an overview over the relevant concatenator classes. Note that concatenators extend `Abstract_Iterator` an therefore provide the same interface to access their internal iterators' data:

Definition 6.1 (`Concatenated_Iterator`).
Let `c_it` be an instance of `Concatenated_Iterator`. `c_it` contains the following fields and methods:

- `c_it.iterators`: An ordered collection of `Abstract_Iterator<DATA_TYPE>` instances previously added via `c_it.append_iterator(iterator)`.
- `c_it.append_iterator(iterator)`: Append `iterator` at the end of `c_it.iterators`.
- `c_it.get_num_remaining()`: `return` $\sum_{i=1}^{\texttt{c\_it.iterators.size()}}$ `c_it.iterators[i].get_num_remaining();`
- `c_it.get_next_count()`: The sum over all internal iterators' corresponding method return value up to (and including) the first where not all elements can can immediately be retrieved.

```
1. result = 0;
2. For( $n = 1$; $n \leq$ c_it.iterators.size(); $n$++ ):
   2.1. $c$ = c_it.iterators[$n$].get_next_count();
   2.2. result = result + $c$;
```

```
    2.3. If( $c <$ c_it.iterators[n].get_num_remaining() ):  break;
         We leave this loop as soon as we encounter the first iterator where not all elements
         can immediately be retrieved.
  3. return result;
```

- `c_it.get_next(num)`: Retrieve the data from internal iterators in the order the latter were appended:

```
  1. result $=$ empty_list();
  2. $n_{\text{remaining}} =$ num;
  3. While( $n_{\text{remaining}} > 0$ ):
    3.1. $n_{\text{next}} = \min\{n_{\text{remaining}},$ c_it.iterators[1].get_num_remaining()$\}$;
    3.2. result.append_to_end(c_it.iterators[1].get_next(num = $n_{\text{next}}$);
    3.3. If ( c_it.iterators[1].get_num_remaining() $== 0$ ):
             remove first element from c_it.iterators;
    3.4. $n_{\text{remaining}} = n_{\text{remaining}} - n_{\text{next}}$;
  4. return result;
```

6.1 Numeric and character iterator concatenation

In order to concatenate iterators over strings (instances of `Character_Queue_Iterator` from Section 4.2, for example), we define `Character_Concatenated_Iterator` as a subclass of `Concatenated_Iterator`. The latter merely fixed the type parameter `DATA_TYPE` to only allow strings and otherwise behaves exactly as its direct superclass.

For the concatenation of numeric iterators, however, we define a class that is able to perform some form of offset normalization. The consideration behind this is that simply concatenating numeric iterators might introduce "unnatural" jumps in the resulting sequence which correspond to switching from one iterator to the other. In order to preserve a certain smoothness in the sequence the `Numeric_Offset_Normalized_Concatenated_Iterator` defines an additive offset for each of its internal iterators. The offset is computed right before the concatenator emits the first value of a new iterator and is based on the last few values of retrieved from the last iterator:

Definition 6.2 (`Numeric_Offset_Normalized_Concatenated_Iterator`).
Let `nc_it` be an instance of `Numeric_Offset_Normalized_Concatenated_Iterator`. We denote all the iterators that will ever be appended to `nc_it` during its lifetime with $I_1, I_2, \ldots$. Furthermore, let $x_{1,n}, \ldots, x_{N_n,n} \in \mathbb{R}$ refer to all the values that iterator I_n will create[1]. Lastly, we define $o_n \in \mathbb{R}$ to be the offset used for values from iterator I_n and $\forall i = 1, \ldots, N_n : \hat{x}_{i,n} := x_{i,n} + o_n$:

- `nc_it.get_next(num)`: The iterator retrieves its values in the same order and manner as its superclass. The only difference is that it returns $\hat{x}_{i,n}$ instead of $x_{i,n}$ (as the superclass would). For all $n > 1$, the offset o_n is computed when the concatenator retrieves the first element from the internal iterator I_{n+1}. We define $o_1 := 0$ and for $n \geq 1$ as follows:
  - $m_n := \min\{N_n, \texttt{nc\_it.offset\_max\_num\_elems + 1}\}$
  - $\forall i = 1, \ldots, (N_n - 1) : \Delta_{i,n} := \hat{x}_{i+1,n} - \hat{x}_{i,n}$
  - $\mu_n := \begin{cases} \texttt{mean}(\Delta_{N_n - m_n}, \ldots, \Delta_{N_n}) & \text{if } m_n \geq 1 \\ 0 & \text{else} \end{cases}$
  - $s_n := \begin{cases} \texttt{standard\_deviation}(\Delta_{N_n - m_n}, \ldots, \Delta_{N_n}) & \text{if } m_n \geq 2 \\ 0 & \text{else} \end{cases}$

    Compute mean μ_n and standard deviation s_n of the last m_n value differences.
  - $\delta_n := \texttt{draw\_norm(mean = } \mu_n\texttt{, stddev = } s_n\texttt{)}$
  - $o_{n+1} := \hat{x}_{N_n,n} - x_{1,n+1} + \delta_n$

[1] $\Rightarrow$ before the first element is retrieved from I_n, we have I_n`.get_num_remaining()` $== N_i$.

Note that this results in the difference between the last offset normalized element emitted by iterator I_n and the first offset normalized element emitted by I_{n+1} to be δ_n. The latter is drawn from an order of magnitude similar to the step size of the last emitted elements:

$$\begin{aligned}\hat{x}_{1,n+1} - \hat{x}_{N_n,n} &= x_{1,n+1} + o_{n+1} - \hat{x}_{N_n,n} \\ &= x_{1,n+1} + \hat{x}_{N_n,n} - x_{1,n+1} + \delta_n - \hat{x}_{N_n,n} \\ &= \delta_n\end{aligned}$$

7 Iterator Factory

The `Factory` class and its subclasses are the link between words emitted by a language model and the creation of specific `Random_Walk_Numeric_Iterator` and `Character_Queue_Iterator` instances. Figure 7.1 contains an overview over the relevant concatenator classes. Their common principle is that they provide a `create_object(name, ...)` method that takes a name string and, depending on the object type, a number of additional arguments and creates and returned a new object based on those parameters. Which objects are generated by what `name` argument, how they are generated and what additional parameters they require may be specified via a JSON file. We omit a detailed documentation of the expected JSON format (examples can be found at [20]) and continue with a description of each class.

Definition 7.1 (`Factory`).
Let `fc` be an instance of `Factory`. `fc` contains the following fields and methods:

- `DATA_TYPE`: A common supertype of the objects generated by `fc.create_object(...)`.
- `fc.create_object(name, ...)`: Creates a new instance of an object associated with `name`. Depending on `name`, "`...`" might contain additional parameters.

The `Alphanumeric_Iterator_Factory` is a subclass of `Factory` which restricts the data type of the created objects to be either `Abstract_Numeric_Iterator` or `Character_Queue_Iterator`. Otherwise it behaves like `Factory`. `Synthetic_Datastream_Iterator_Factory` is the final class in this chain of inheritance and it is the one actually used by the SDG to generate subsequences. It extracts a number of special parameters from the provided JSON configuration file which are then used to create different types of SDG related data sequences:

Definition 7.2 (`Synthetic_Datastream_Iterator_Factory`).
Let `sdg_fc` be an instance of `Synthetic_Datastream_Iterator_Factory`. `sdg_fc` contains the following fields and produces the following types of data sequences:

- `NUM_NUMERIC_SEQ`: The number of regular numeric sequences (see below) `sdg_fc` generates. The factory generates this number of random seeds and names for numeric sequences. Later on, the language model trainer samples (without replacement) a predefined number of sequence names for each numeric dimension. It uses those names for the corresponding dimension's vocabulary.
- `NUM_CHAR_SEQ`: Same as `NUM_NUMERIC_SEQ` but for string sequences.
- `fc.create_object(name, ...)`: The `name` arguments and their corresponding expected additional parameters can be grouped as follows:
  1. **Regular sequences**: These are the sequences that represent repeating subsequences (motifs) in numeric streams or labels in string label steams. Regular sequences may therefore be divided into two subgroups:

     1.1. **Numeric sequences**:
     The parameters for numeric sequences are stored in the `.Numeric_Sequences_Paramers` instance `sdg_fc.numeric_seq_params`.

     Let `p := sdg_fc.numeric_seq_params` denote a shorthand for that instance and `n` $\in \{1, \ldots,$ `p.numeric_sequence_names.size()`$\}$.

     `sdg_fc.create_object(p.numeric_sequence_names[n])` is defined as:

     i. Store current random generator state in s;

     ii. Set random generator seed to `p.numeric_sequence_seeds[n]`;

     iii. `fixed_length` = max{`p.min_num_length`,
     min{`p.max_num_length`,

```
                    draw_norm(mean = p.mean_num_length,
                              stddev = p.stddev_num_length)};
```

iv. `result = new Random_Walk_Numeric_Iterator(`
`fixed_length = fixed_length,`
`initial_value = 0);`

Creates a new `Random_Walk_Numeric_Iterator` instance with the provided parameters.

v. Restore random generator state s;

vi. `return result;`

Note that by initially setting the global random generator seed to the predefined value, the random walk's length as well as its internal **`rnd_seed`** field is also deterministically drawn based on that seed. This implies that repeated calls (with the same n) to `sdg_fc.create_object(p.numeric_sequence_names[n])` result in objects generating the exact same random walk. This is, of course, exactly what we want for our motif sequences.

1.2. **String sequences**:
The parameters for string sequences are stored in the `.Character_Sequences_Paramers` instance `sdg_fc.character_seq_params`.

Let `p := sdg_fc.character_seq_params` denote a shorthand for that instance and `n` $\in \{1, \ldots,$ `p.character_sequence_names.size()`$\}$.

`sdg_fc.create_object(p.character_sequence_names[n])` is defined as:

i. `result = new Character_Queue_Iterator();`

ii. `result.append_element(p.character_sequence_names[n],`
`length = 1);`

Regular character "sequences" always have the length one as the correspond to individually emitted symbols. We use the term "sequence" here for consistency reasons.

iii. `return result;`

2. **NA sequences**: NA stands for "not available" and this class of sequences represent random sequences without any repeating structure (like the grey parts in the example stream from Figure 1.1). SDG distinguishes between three types of NA sequences:

2.1. **Inter NA sequence**: These are longer sequences serving as longer, non repeating random sequences between sequence chains corresponding to word chans (Definition 3.3). The idea behind this is that we assume that single real world events trigger sequence chains and that events themselves are separated by longer pauses. Those pauses are modeled by Inter NA sequences. Inter NA sequences are therefore created for each stream dimension each time the underlying language model returns to its idle state.

2.2. **Intra NA sequences**: These are shorter sequences serving as short, non repeating random sequences between the sequences of a single sequence chain. Their function is to separate subsequent motif sequences by a small random sequence.

2.3. **Padding NA sequences**: These are sequences inserted by the SDG in order to ensure that the order in which the first elements of non-NA sequences are emitted is consistent with the order in which the underlying language model emits the corresponding words. A padding NA sequence is created as follows (inter and intra NA sequences are created analogously - simply replace all occurrences of the term "padding" with "inter" or "intra"):

2.3.1. **Numeric padding NA sequences**: The length parameters for numeric padding NA sequences are stored in `sdg_fc.padding_NA_length_params` which is an instance of `.Padding_NA_Length_Paramers`. Its associated name is stored in the `padding_NA_rnd_walk_name` field of the `.NA_Numeric_Sequence_Paramers` instance `sdg_fc.NA_numeric_seq_params`.

We define `p := sdg_fc.padding_NA_length_params` and the corresponding creation name `name := sdg_fc.NA_numeric_seq_params.padding_NA_rnd_walk_name`. Unlike with non-NA numeric sequences, here we require an additional parameter `initial_value` for the `create_object(...)` method in order to ensure that the random walks target value is not moved by the concatenator's offset. We will elaborate more on this in Chapter 9.

`sdg_fc.create_object(name, initial_value)` is defined as:

```
i.   fixed_length = max{p.min_num_length,
                        min{p.max_num_length,
                            draw_norm(mean = p.mean_num_length,
                                      stddev = p.stddev_num_length)}};
ii.  result = new Random_Walk_Numeric_Iterator(
                                      fixed_length = fixed_length,
                                      initial_value = initial_value);
```

Creates a new `Random_Walk_Numeric_Iterator` instance with the provided parameters. Note that we do not set any random seed here. This implies that each call to `sdg_fc.create_object(name, initial_value)` results in an object generating a different random walk.

```
iii. return result;
```

2.3.2. **String padding NA sequences**: The length parameters for string padding NA sequences are stored in `sdg_fc.padding_NA_length_params` (same as for the numeric NA padding sequence) and its name in the `padding_NA_character_sequence_name` field of the `.NA_Character_Sequence_Paramers` instance which itself is stored at `sdg_fc.NA_character_seq_params`.

With `name := sdg_fc.NA_character_seq_params`
`.padding_NA_character_sequence_name`,

`sdg_fc.create_object(name)` is defined as:

```
i.   result = new Character_Queue_Iterator();
ii.  fixed_length = max{p.min_num_length,
                        min{p.max_num_length,
                            draw_norm(mean = p.mean_num_length,
                                      stddev = p.stddev_num_length)}};
```

The length is the same as for numeric padding NA sequences.

```
iii. result.append_element(sdg_fc.NA_label, length = fixed_length);
```

Note that for character streams, the `sdg_fc.NA_label` label merely serves the technical purpose of enabling the SDG to treat the numeric and character sequences in a unified manner. In practice, occurrences of these labels on SDG's symbolic streams can simply be ignored (which is exactly what our MDP does).

```
iv.  return result;
```

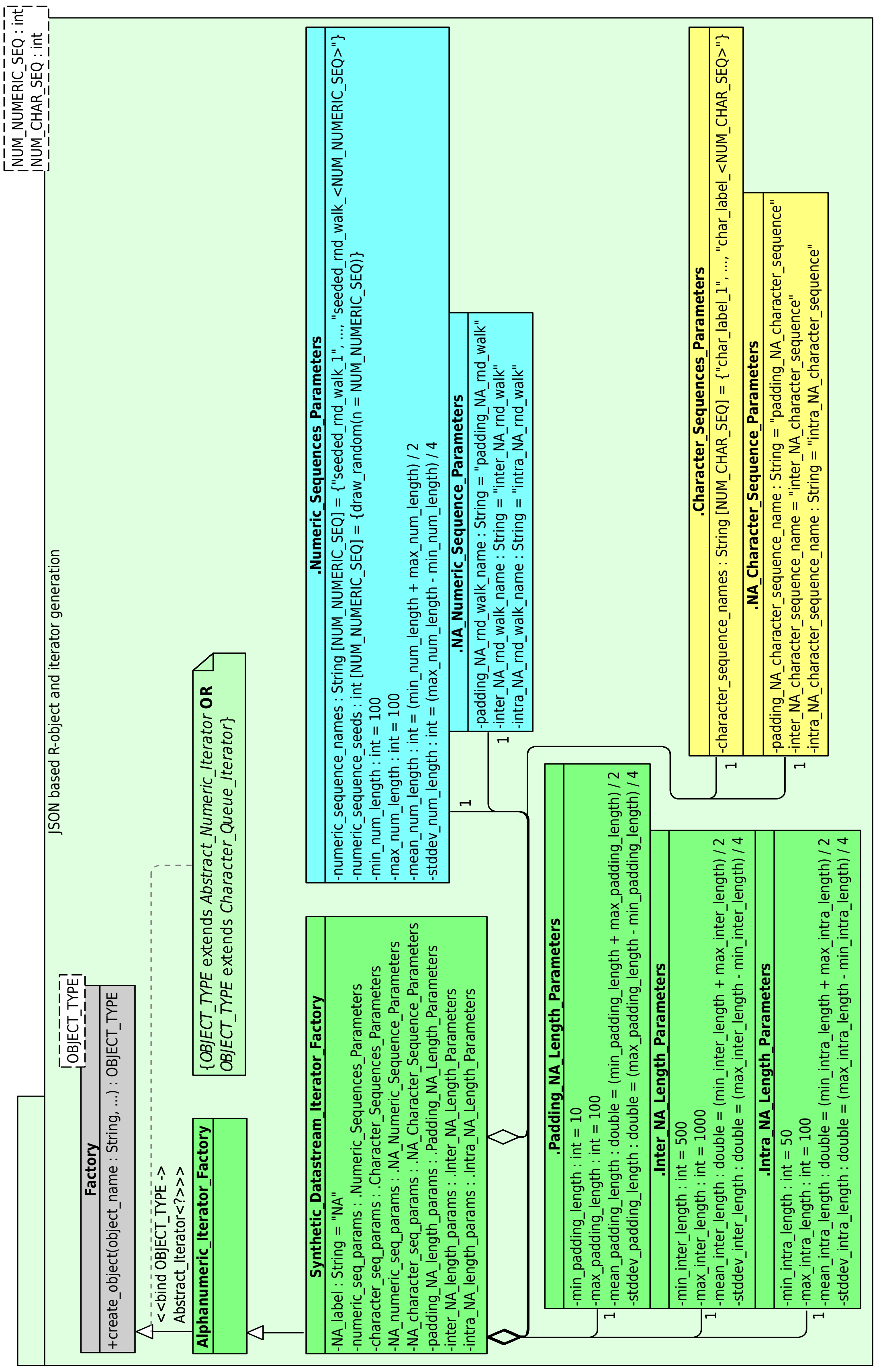

NUM_NUMERIC_SEQ : int
NUM_CHAR_SEQ : int
JSON based R-object and iterator generation
OBJECT_TYPE
Factory
+create_object(object_name : String, ...) : OBJECT_TYPE
<<bind OBJECT_TYPE -> Abstract_Iterator<?>>>
Alphanumeric_Iterator_Factory
{OBJECT_TYPE extends Abstract_Numeric_Iterator OR OBJECT_TYPE extends Character_Queue_Iterator}
Synthetic_Datastream_Iterator_Factory
-NA_label : String = "NA"
-numeric_seq_params : .Numeric_Sequences_Parameters
-character_seq_params : .Character_Sequences_Parameters
-NA_numeric_seq_params : .NA_Numeric_Sequence_Parameters
-NA_character_seq_params : .NA_Character_Sequence_Parameters
-padding_NA_length_params : .Padding_NA_Length_Parameters
-inter_NA_length_params : .Inter_NA_Length_Parameters
-intra_NA_length_params : .Intra_NA_Length_Parameters
.Numeric_Sequences_Parameters
-numeric_sequence_names : String [NUM_NUMERIC_SEQ] = {"seeded_rnd_walk_1", ..., "seeded_rnd_walk_<NUM_NUMERIC_SEQ>"}
-numeric_sequence_seeds : int [NUM_NUMERIC_SEQ] = {draw_random(n = NUM_NUMERIC_SEQ)}
-min_num_length : int = 100
-max_num_length : int = 100
-mean_num_length : int = (min_num_length + max_num_length) / 2
-stddev_num_length : int = (max_num_length - min_num_length) / 4
.NA_Numeric_Sequence_Parameters
-padding_NA_rnd_walk_name : String = "padding_NA_rnd_walk"
-inter_NA_rnd_walk_name : String = "inter_NA_rnd_walk"
-intra_NA_rnd_walk_name : String = "intra_NA_rnd_walk"
.Character_Sequences_Parameters
-character_sequence_names : String [NUM_CHAR_SEQ] = {"char_label_1", ..., "char_label_<NUM_CHAR_SEQ>"}
.NA_Character_Sequence_Parameters
-padding_NA_character_sequence_name : String = "padding_NA_character_sequence"
-inter_NA_character_sequence_name = "inter_NA_character_sequence"
-intra_NA_character_sequence_name : String = "intra_NA_character_sequence"
.Padding_NA_Length_Parameters
-min_padding_length : int = 10
-max_padding_length : int = 100
-mean_padding_length : double = (min_padding_length + max_padding_length) / 2
-stddev_padding_length : double = (max_padding_length - min_padding_length) / 4
.Inter_NA_Length_Parameters
-min_inter_length : int = 500
-max_inter_length : int = 1000
-mean_inter_length : double = (min_inter_length + max_inter_length) / 2
-stddev_inter_length : double = (max_inter_length - min_inter_length) / 4
.Intra_NA_Length_Parameters
-min_intra_length : int = 50
-max_intra_length : int = 100
-mean_intra_length : double = (min_intra_length + max_intra_length) / 2
-stddev_intra_length : double = (max_intra_length - min_intra_length) / 4
1
1
1
1
1
1
1

Figure 7.1: UML 2.0 class diagram depicting the factory classes for iterator generation.

8 SDG: Decorator hierarchy

The SDG employs the decorator hierarchy depicted in Figure 8.1. We define two helper classes called `Decorated_Numeric_Iterator` and `Decorated_Character_Iterator` which serve the purpose of encapsulating the decorator hierarchy of each numeric and each string stream. Both are themselves implementations of `Abstract_Iterator_Decorator` and are structured as follows:

Definition 8.1 (`Decorated_Numeric_Iterator` and `Decorated_Character_Iterator`).
Let `dn_it` be an instance of `Decorated_Numeric_Iterator`. `dn_it` contains the following fields and methods:

- `dn_it.decorated_iterator`: The iterator decorated by `d_it` (as defined in Definition 5.1). It is set to the `dec_num_it` instance, i.e. the topmost instance in the decoration hierarchy depicted in Figure 8.1.
- `dn_it.undecorated`: A reference to the innermost instance instance in the decoration hierarchy. Corresponds to `num_it` instance depicted in Figure 8.1 (i.e. the innermost, non-decorator iterator in the decoration hierarchy). As we will see in the next section, this instance is used to append newly created iterator instances while `dec_num_it` is used to retrieve these instances' decorated values.
- `dn_it.decorate(data_element)`: Note that by strictly applying the definition of each decoration step, the type of data `dn_it.decorated_iterator` returns is a tuple the left entry of which is the string label from the annotator and the right entry of which is itself a tuple. The latter contains the timestamp in its left entry and the actual value from the undecorated iterator in its right entry. The `dn_it.decorate(data_element)` simply transforms this into a triple containing these values in the same order.
- The `annotate()` method used in the `Annotation_Iterator_Decorator` instance: The employed `annotate()` method annotates each data element with a ground truth. It makes use of bookkeeping data structures to implement the following functionality:
  - * If the current data element has been created by an iterator corresponding to a non-NA sequence (see Definition 7.2), the annotation label is set to the `name` argument used with the factory's `create_instance(...)` method to create said iterator.
  - * Otherwise, if the current data element has been created by an iterator corresponding to an NA sequence, the annotation label is set to value of the corresponding factory's `NA_label` value.

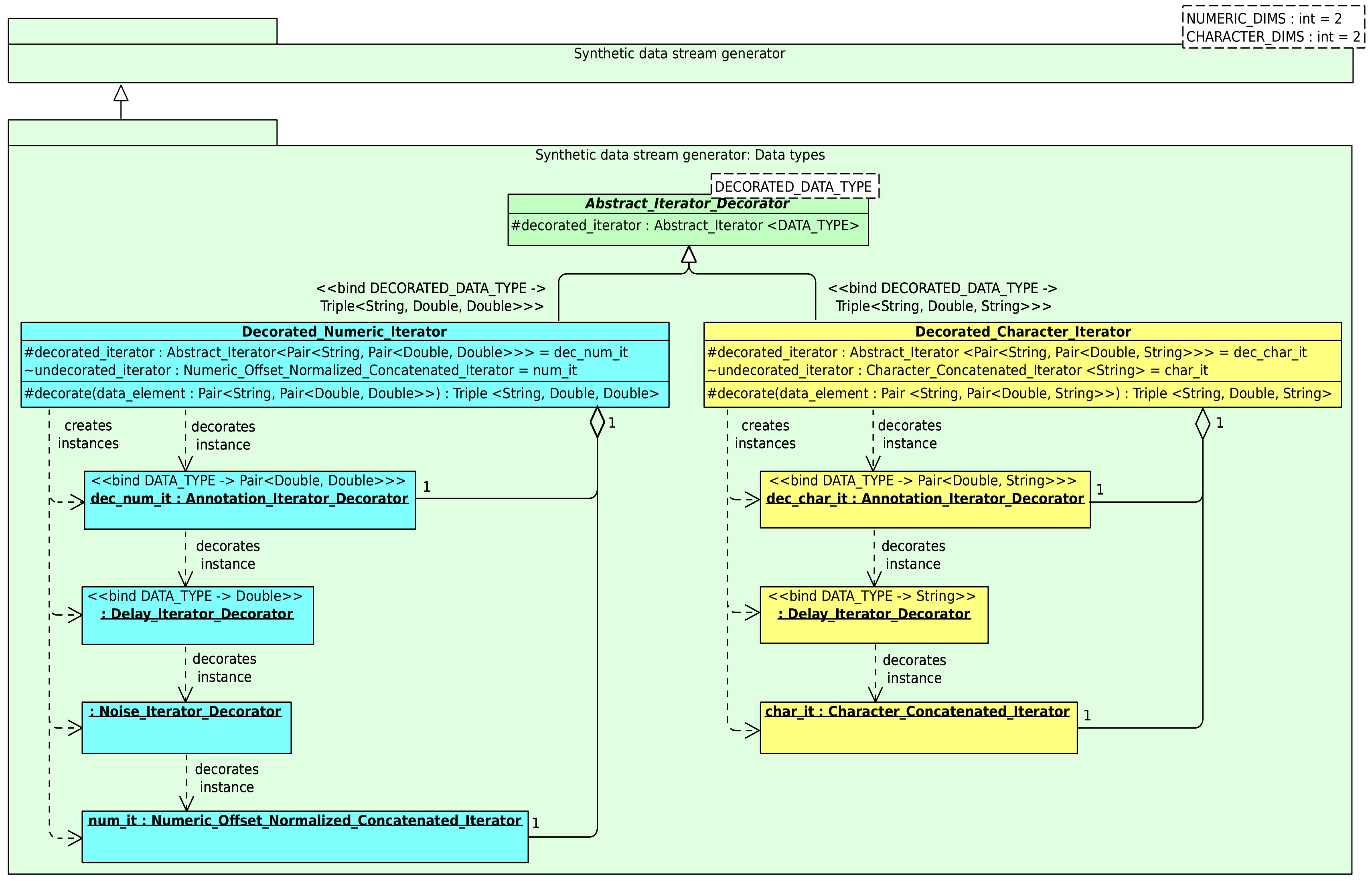

NUMERIC_DIMS : int = 2
CHARACTER_DIMS : int = 2
Synthetic data stream generator
Synthetic data stream generator: Data types
DECORATED_DATA_TYPE
Abstract_Iterator_Decorator
#decorated_iterator : Abstract_Iterator <DATA_TYPE>
<<bind DECORATED_DATA_TYPE -> Triple<String, Double, Double>>>
<<bind DECORATED_DATA_TYPE -> Triple<String, Double, String>>>
Decorated_Numeric_Iterator
#decorated_iterator : Abstract_Iterator<Pair<String, Pair<Double, Double>>> = dec_num_it
~undecorated_iterator : Numeric_Offset_Normalized_Concatenated_Iterator = num_it
#decorate(data_element : Pair<String, Pair<Double, Double>>) : Triple <String, Double, Double>
Decorated_Character_Iterator
#decorated_iterator : Abstract_Iterator <Pair<String, Pair<Double, String>>> = dec_char_it
~undecorated_iterator : Character_Concatenated_Iterator <String> = char_it
#decorate(data_element : Pair <String, Pair<Double, String>>) : Triple <String, Double, String>
creates instances
decorates instance
1
<<bind DATA_TYPE -> Pair<Double, Double>>>
dec_num_it : Annotation_Iterator_Decorator
<<bind DATA_TYPE -> Double>>
: Delay_Iterator_Decorator
: Noise_Iterator_Decorator
num_it : Numeric_Offset_Normalized_Concatenated_Iterator
<<bind DATA_TYPE -> Pair<Double, String>>>
dec_char_it : Annotation_Iterator_Decorator
<<bind DATA_TYPE -> String>>
: Delay_Iterator_Decorator
char_it : Character_Concatenated_Iterator

Figure 8.1: UML class/object 2.0 diagram depicting the decoration hierarchy of SDG's internal iterators.

9 SDG: Core

Now we have all the building blocks we need to define the behavior of the SDG. Its core class is `Synthetic_Datastream_Iterators`. The latter's overall structure is depicted in Figure 9.1 and consist of the following components:

Definition 9.1 (`Synthetic_Datastream_Iterators`).
Let `sdg` be an instance of `Synthetic_Datastream_Iterators`. `sdg` contains the following fields and methods:

- `NUMERIC_DIMS`: The number of numeric streams `sdg` generates.
- `CHARACTER_DIMS`: The number of string label streams `sdg` generates.
- `sdg.rnd_seed`: The random generator seed is set to this value during the initialization of `sdg`. Two `Synthetic_Datastream_Iterators` with the same random seed value and all other parameters matching will always produce the exact same data streams.
- `decorated_num_iterators`:
  An ordered collection of `<NUMERIC_DIMS>` instances of `Decorated_Numeric_Iterator` (Definition 8.1). The instances are initially empty, i.e., no sequences are appended to their internal undecorated concatenator during initialization.
- `sdg.decorated_char_iterators`: An ordered collection of `<CHARACTER_DIMS>` instances of initially empty `Decorated_Character_Iterator` (Definition 8.1).
- `sdg.iterator_facotry`: The `Synthetic_Datastream_Iterator_Factory` instance (see Definition 7.2) used by `sdg` to create new sequences.
- `sdg.vocabulary`: The `Multivariate_Vocabulary` instance (Definition 3.1) also used by the language model. It is created by:
  - * Randomly sampling `<NUM_WORDS_PER_DIM>` (without replacement) words from the factory's set of regular numeric sequence names for each of the `<NUMERIC_DIMS>` numeric streams this SDG instance will generate.
  - * Randomly sampling `<NUM_WORDS_PER_DIM>` (without replacement) words from the factory's set of regular string sequence names for each of the `<CHARACTER_DIMS>` string streams this SDG instance will generate.
- `sdg.language_model`: The `Multivariate_Count_LM` instance (Definition 3.6) used to determine the order in which sequences and strings are emitted in SDG's streams. That model uses `sdg.vocabulary` as its own `Multivariate_Vocabulary` instance and is either read from a JSON file or newly trained by a `Trainer_Multivariate_Count_LM` instance (Section 3.4) during initialization of the `sdg` instance.
- `sdg.get_next(dimension, num)`: Refreshes the internal fields of `sdg` and then fetches the next elements from the `<dimension>`-th data stream:

```
1. sdg.refresh();
2. If( dimension <= NUMERIC_DIMS ):
       return sdg.decorated_num_iterators[dimension].get_next(num);
3. Else:
       return sdg.decorated_char_iterators[dimension - NUMERIC_DIMS]
                                                   .get_next(num);
```

- `sdg.get_next_count(dimension)`: Refreshes the internal fields of `sdg` and returns the next count number from the `<dimension>`-th data stream:

```
1. sdg.refresh();
```

```
    2. If( dimension <= NUMERIC_DIMS ):
           return sdg.decorated_num_iterators[dimension].get_next_count();
    3. Else:
           return sdg.decorated_char_iterators[dimension - NUMERIC_DIMS]
                                                          .get_next_count();
```

- `sdg.refresh()`: See Algorithm 9.1.

The `refresh()` method is the SDG's core method. Here, the SDG draws the next word from its language model, generates the according sequence and makes sure that the sequences appear in the order their corresponding words (aka. sequence names) have been drawn. This is done based on the NA subsequence definitions from Definition 7.2. The exact procedure is listed in Algorithm 9.1:

Algorithm 9.1 (`refresh()`).
Let `sdg` be an instance of `Synthetic_Datastream_Iterators`.
Let $\forall d_x = 1, \dots,$ `NUMERIC_DIMS`, $\forall d_c = 1, \dots,$ `CHARACTER_DIMS`:

- `dec_num_it[`d_x`]` := `sdg.decorated_numeric_iterators[`d_x`]`
- `num_it[`d_x`]` := `sdg.decorated_numeric_iterators[`d_x`].undecorated_iterator`
- `dec_char_it[`d_c`]` := `sdg.decorated_character_iterators[`d_c`]`
- `char_it[`d_c`]` := `sdg.decorated_character_iterators[`d_c`].undecorated_iterator`
- `DIMS` := `NUMERIC_DIMS + CHARACTER_DIMS`

`refresh()` is defined as:

```
1. If( $\exists d_x = 1, \dots,$ NUMERIC_DIMS:  num_it[$d_x$].get_num_remaining() $> 0$ ||
       $\exists d_c = 1, \dots,$ CHARACTER_DIMS:  char_it[$d_c$].get_num_remaining() $> 0$ ):
       return;
   If at least one undecorated internal iterator still has unretrieved elements left, do nothing (even if
   the elements cannot be immediately retrieved because of the delay decorator).
2. Elseif( sdg.next_word == null ):
       sdg.next_word = sdg.language_model.emit_word();
3. NA_padding_required = false;
4. Let $t_d \in \mathbb{R}$ be the timestamp of the next element that will be returned in the $d$-th stream and
   $t_{\min} := \min_{d=1,\dots,\texttt{DIMS}} t_d$.
5. If( sdg.next_word == sdg.vocabulary.idle_word ):
   We need to add inter NA sequences to all dimensions but also make sure that we first pad each
   dimension long enough so that each inter sequence begins *after* the last non-NA sequence has
   finished (so the inter NA sequence happens roughly at the same time across all dimensions).
   5.1. Let $t_d^{\textbf{last\_end}}$ be the timestamp of the *last* element of the *last* non-NA sequence emitted in
        dimension $d$. Set $t_d^{\textbf{last\_end}} = -\infty$, if no non-NA sequence has been emitted in dimension $d$ so
        far. Let $t_{\max}^{\text{last\_end}} := \max_{d=1,\dots,\texttt{DIMS}} t_d^{\textbf{last\_end}}$.
   5.2. If( $\forall d = 1, \dots,$ DIMS : $t_d \geq t_{\max}^{\text{last\_end}}$ ):
        Add inter NA sequence to all dimensions.
     5.2.1. num_inter_NA_name = sdg.iterator_factory
                                    .NA_numeric_seq_params
                                    .inter_NA_rnd_walk_name;
     5.2.2. char_inter_NA_name = sdg.iterator_factory
                                    .NA_character_seq_params
                                    .inter_NA_character_sequence_name;
     5.2.3. For( $d_x = 1, \dots, NUMERIC\_DIMS$ ):
      5.2.3.1. Consider the terminology from the numeric offset concatenator Definition 6.2 and let
               $n$ be the index of the last iterator that num_it[$d_x$] iterated over.
```

```
We set: initial_value = $\begin{cases} \hat{x}_{N_n,n} & \text{if } n > 1 \\ 0 & \text{else} \end{cases}$
Therefore we have $x_{1,n+1} = \hat{x}_{N_n,n}$ and the offset $o_{n+1} = \hat{x}_{1,n+1} - x_{1,n+1} = \delta_n$. Since
our random walk target value is zero, our normalized target value is simply $\delta_n$, the
expected value of which is also zero. Our NA random walks will therefore stay within a
small area around zero (the order of magnitude of said area lies within the magnitude
of the variance of $\delta_n$).
        5.2.3.2. num_it[$d_x$].append_iterator(
                                   sdg.iterator_factory
                                      .create_object(num_inter_NA_name,
                                                     initial_value = initial_value));
      5.2.4. For( $d_c = 1, \ldots, CHARACTER\_DIMS$ ):
                char_it[$d_c$].append_iterator(
                                   sdg.iterator_factory
                                      .create_object(char_inter_NA_name));
      5.2.5. sdg.next_word = null;
   5.3. Else:  NA_padding_required = true;
6. Else:  // I.e., next_word is not the idle word
   6.1. Let $t_d^{\textbf{last\_begin}}$ be the timestamp of the *first* element of the *last* non-NA sequence emitted in
        dimension $d$. Set $t_d^{\textbf{last\_begin}} = -\infty$, if no non-NA sequence has been emitted in dimension
        $d$ so far. Let $t_{\max}^{\text{last\_begin}} := \max_{d=1,\ldots,\texttt{DIMS}} t_d^{\textbf{last\_begin}}$. In order to keep the streams consistent with
        the order of the words our language model emits, the first element of the next sequence
        corresponding to sdg.next_word must get a timestamp $t \geq t_{\max}^{\text{last\_begin}}$.
   6.2. If( $\forall d = 1, \ldots, \texttt{DIMS} : t_d \geq t_{\texttt{max}}^{\textbf{last\_begin}}$ ):
        Add the sequence corresponding to next_word, preceded by and an intra NA sequence (but
        only if predecessor sequence was no inter NA sequence) to its appropriate dimension. Let
        $d_w := d(\texttt{sdg.next\_word})$ be the dimension to which the next word belongs to.
      6.2.1. last_sequence_NOT_inter = last sequence in dimension $d_w$ did not
                                       correspond to an inter NA sequence;
      6.2.2. num_intra_NA_name = sdg.iterator_factory
                                    .NA_numeric_seq_params
                                    .intra_NA_rnd_walk_name;
      6.2.3. char_intra_NA_name = sdg.iterator_factory
                                     .NA_character_seq_params
                                     .intra_NA_character_sequence_name;
      6.2.4. If(last_sequence_NOT_inter && $d_w \leq$ NUMERIC_DIMS):
        6.2.4.1. Compute initial_value just as in item 5.2.3.1.;
        6.2.4.2. num_it[$d_w$].append_iterator(
                    sdg.iterator_factory
                       .create_object(num_intra_NA_name,
                                      initial_value = initial_value));
      6.2.5. If(last_sequence_NOT_inter && $d_w >$ NUMERIC_DIMS):
                char_it[$d_w$].append_iterator(
                                   sdg.iterator_factory
                                      .create_object(char_intra_NA_name));
      6.2.6. If( $d_w \leq$ NUMERIC_DIMS):
                num_it[$d_w$].append_iterator(
                                 sdg.iterator_factory
                                    .create_object(sdg.next_word)));
```

```
    6.2.7. Else:
               char_it[$d_w$].append_iterator(
                                        sdg.iterator_factory
                                           .create_object(sdg.next_word)));
    6.2.8. sdg.next_word = null;
  6.3. Else:  NA_padding_required = true;
7. If( NA_padding_required == true ):
```

This may only happen if we could not insert a new sequence corresponding to the next word (i.e., `sdg.next_word`) without potentially violating the order consistency. Let $t_{\min} := \min_{d=1,\ldots,D} t_d$ be the smallest timestamp among all stream's next timestamps. This corresponds to those streams which are the furthest behind in time. Those are the ones to which we add padding NA sequences:

```
  7.1. num_padding_NA_name = sdg.iterator_factory
                                 .NA_numeric_seq_params
                                 .padding_NA_rnd_walk_name;
  7.2. char_padding_NA_name = sdg.iterator_factory
                                  .NA_character_seq_params
                                  .padding_NA_character_sequence_name;
  7.3. For( $din1,\ldots,D|t_d == t_{\min}$):
       i. If( $d_w \leq$ NUMERIC_DIMS ):
        i.1. Compute initial_value just as in item 5.2.3.1.;
        i.2. num_it[$d_w$].append_iterator(
                   sdg.iterator_factory
                      .create_object(num_padding_NA_name,
                                     initial_value = initial_value));
      ii. Else:
               char_it[$d_w$].append_iterator(
                                        sdg.iterator_factory
                                           .create_object(char_padding_NA_name));
```

Note that `refresh()` ensures that after its execution there are always new elements to be retrieved from the internal iterators. It does so by either directly inserting a new sequence corresponding to the next word the internal language model emitted or, if that was not possible due to our ordering constraints, by inserting at least one padding NA sequence into the stream which is the furthest behind with respect to its next timestamp. This procedure also ensures that the conditions at 5.2. and 6.2. are guaranteed to be met after a finite number of `refresh()` calls. This concludes our documentation of the SDG and we continue with a documentation of our transceiver framework.

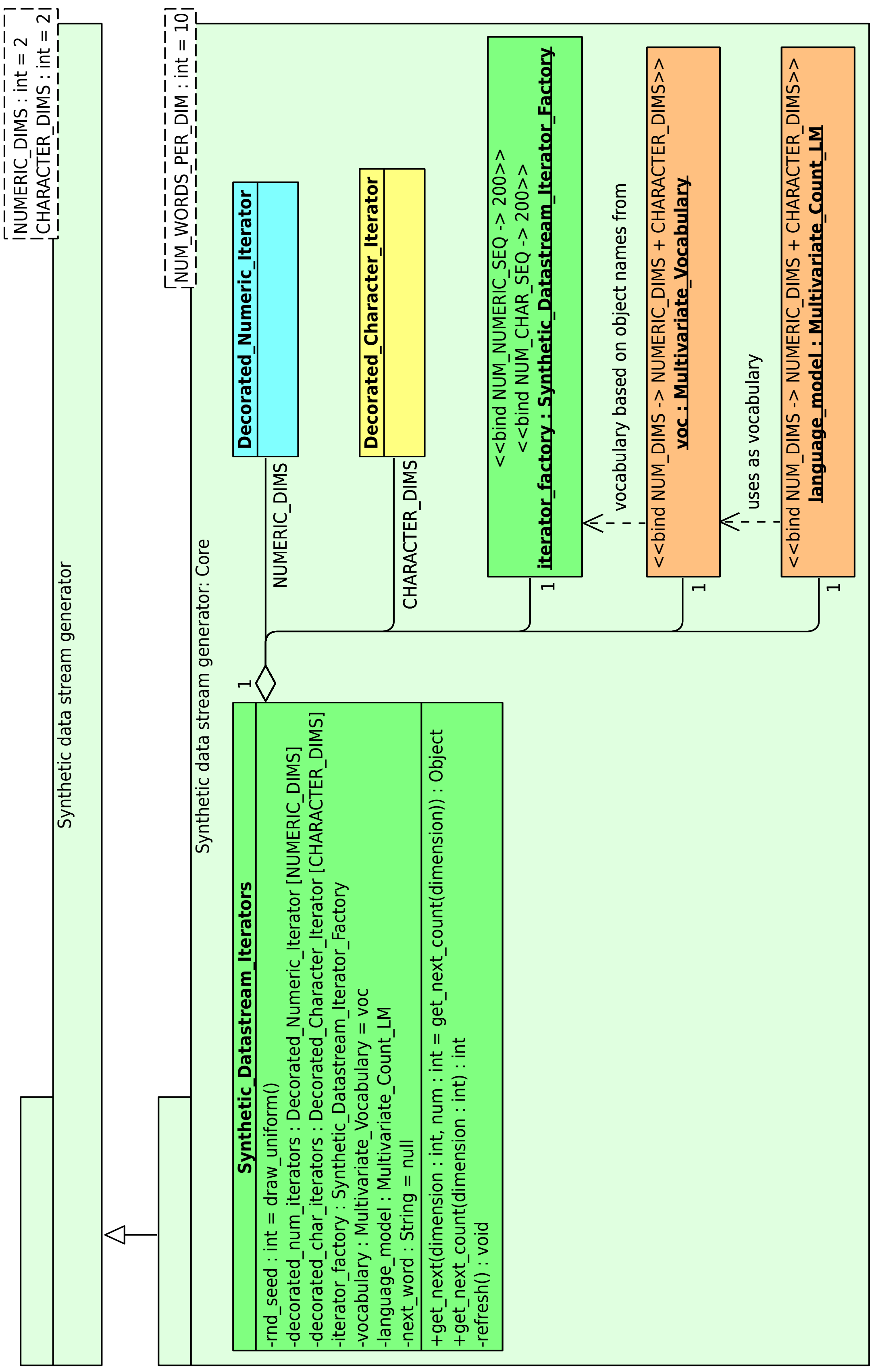

NUMERIC_DIMS : int = 2
CHARACTER_DIMS : int = 2
Synthetic data stream generator
NUM_WORDS_PER_DIM : int = 10
Synthetic data stream generator: Core
Decorated_Numeric_Iterator
Decorated_Character_Iterator
NUMERIC_DIMS
CHARACTER_DIMS
1
<<bind NUM_NUMERIC_SEQ -> 200>>
<<bind NUM_CHAR_SEQ -> 200>>
iterator_factory : Synthetic_Datastream_Iterator_Factory
vocabulary based on object names from
<<bind NUM_DIMS -> NUMERIC_DIMS + CHARACTER_DIMS>>
voc : Multivariate_Vocabulary
uses as vocabulary
<<bind NUM_DIMS -> NUMERIC_DIMS + CHARACTER_DIMS>>
language_model : Multivariate_Count_LM
1
1
1
Synthetic_Datastream_Iterators
-rnd_seed : int = draw_uniform()
-decorated_num_iterators : Decorated_Numeric_Iterator [NUMERIC_DIMS]
-decorated_char_iterators : Decorated_Character_Iterator [CHARACTER_DIMS]
-iterator_factory : Synthetic_Datastream_Iterator_Factory
-vocabulary : Multivariate_Vocabulary = voc
-language_model : Multivariate_Count_LM
-next_word : String = null
+get_next(dimension : int, num : int = get_next_count(dimension)) : Object
+get_next_count(dimension : int) : int
-refresh() : void

Figure 9.1: UML 2.0 class/object diagram depicting the SDG's overall structure.

Bibliography

[1] Graciela H. Gonzalez, Tasnia Tahsin, Britton C. Goodale, Anna C. Greene, and Casey S. Greene. Recent advances and emerging applications in text and data mining for biomedical discovery. *Briefings in Bioinformatics*, 17(1):33–42, 2016. doi: 10.1093/bib/bbv087. URL `http://dx.doi.org/10.1093/bib/bbv087`.

[2] P. Baldi, P. Sadowski, and D. Whiteson. Searching for exotic particles in high-energy physics with deep learning. *Nature Communications*, 5:4308 EP –, Jul 2014. URL `https://doi.org/10.1038/ncomms5308`. Article.

[3] A. A. Pol, G. Cerminara, C. Germain, M. Pierini, and A. Seth. Detector monitoring with artificial neural networks at the CMS experiment at the CERN Large Hadron Collider. *ArXiv e-prints*, July 2018.

[4] Q. Wang, Y. Guo, L. Yu, and P. Li. Earthquake prediction based on spatio-temporal data mining: An lstm network approach. *IEEE Transactions on Emerging Topics in Computing*, pages 1–1, 2018. ISSN 2168-6750. doi: 10.1109/TETC.2017.2699169.

[5] Devendra Kumar Tayal, Arti Jain, Surbhi Arora, Surbhi Agarwal, Tushar Gupta, and Nikhil Tyagi. Crime detection and criminal identification in india using data mining techniques. *AI & SOCIETY*, 30(1):117–127, Feb 2015. ISSN 1435-5655. doi: 10.1007/s00146-014-0539-6. URL `https://doi.org/10.1007/s00146-014-0539-6`.

[6] Félix J. López Iturriaga and Iván Pastor Sanz. Bankruptcy visualization and prediction using neural networks: A study of u.s. commercial banks. *Expert Systems with Applications*, 42 (6):2857 – 2869, 2015. ISSN 0957-4174. doi: https://doi.org/10.1016/j.eswa.2014.11.025. URL `http://www.sciencedirect.com/science/article/pii/S0957417414007118`.

[7] G. A. Susto, A. Schirru, S. Pampuri, S. McLoone, and A. Beghi. Machine learning for predictive maintenance: A multiple classifier approach. *IEEE Transactions on Industrial Informatics*, 11(3): 812–820, June 2015. ISSN 1551-3203. doi: 10.1109/TII.2014.2349359.

[8] Yang Lu. Industry 4.0: A survey on technologies, applications and open research issues. *Journal of Industrial Information Integration*, 6:1 – 10, 2017. ISSN 2452-414X. doi: https://doi.org/10.1016/j.jii.2017.04.005. URL `http://www.sciencedirect.com/science/article/pii/S2452414X17300043`.

[9] Chin-Chia Michael Yeh, Yan Zhu, Liudmila Ulanova, Nurjahan Begum, Yifei Ding, Hoang Anh Dau, Diego Furtado Silva, Abdullah Mueen, and Eamonn Keogh. Matrix profile i: All pairs similarity joins for time series: A unifying view that includes motifs, discords and shapelets. In *Data Mining (ICDM), 2016 IEEE 16th International Conference on*, pages 1317–1322. IEEE, 2016.

[10] Diego F Silva and Gustavo EAPA Batista. Speeding up all-pairwise dynamic time warping matrix calculation. In *Proceedings of the 2016 SIAM International Conference on Data Mining*, pages 837–845. SIAM, 2016.

[11] David R. Shelly. Possible deep fault slip preceding the 2004 parkfield earthquake, inferred from detailed observations of tectonic tremor. *Geophysical Research Letters*, 36(17):n/a–n/a, 2009. ISSN 1944-8007. doi: 10.1029/2009GL039589. URL `http://dx.doi.org/10.1029/2009GL039589`. L17318.

[12] R6: Encapsulated object-oriented programming for r. URL `https://github.com/r-lib/R6`. [last accessed December 16, 2018].

[13] Synthetic data steam generator. URL `https://github.com/GStepien/SDG`. [last accessed December 21, 2018].

[14] R. Kneser and H. Ney. Improved backing-off for m-gram language modeling. In *1995 International Conference on Acoustics, Speech, and Signal Processing*, volume 1, pages 181–184 vol.1, May 1995. doi: 10.1109/ICASSP.1995.479394.

[15] Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, and Phillipp Koehn. One billion word benchmark for measuring progress in statistical language modeling. *CoRR*, abs/1312.3005, 2013. URL `http://arxiv.org/abs/1312.3005`.

[16] Martin Sundermeyer, Hermann Ney, and Ralf Schlüter. From feedforward to recurrent lstm neural networks for language modeling. *IEEE/ACM Trans. Audio, Speech and Lang. Proc.*, 23(3):517–529, March 2015. ISSN 2329-9290. doi: 10.1109/TASLP.2015.2400218. URL `http://dx.doi.org/10.1109/TASLP.2015.2400218`.

[17] Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. *CoRR*, abs/1707.05589, 2017. URL `http://arxiv.org/abs/1707.05589`.

[18] D.A. Levin, Y. Peres, and E.L. Wilmer. *Markov Chains and Mixing Times*. American Mathematical Soc., 2008. ISBN 9780821886274.

[19] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1 edition, 1994. ISBN 0201633612.

[20] Transceiver framework. URL `https://github.com/GStepien/Transceiver_Framework`. [last accessed December 21, 2018].