selfies

Release 1.0.0

Mario Krenn

Aug 17, 2020

CONTENTS:

SELFIES (SELF-referencIng Embedded Strings) is a 100% robust molecular string representation. A main objective is to use SELFIES as direct input into machine learning models, in particular in generative models, for the generation of outputs with guaranteed validity.

This library is intended to be light-weight and easy to use.

For explanation of the underlying principle (formal grammar) and experiments, please see the original paper.

For comments, bug reports or feature ideas, please use github issues or send an email to mario.krenn@utoronto.ca and alan@aspuru.com.

CHAPTER

ONE

INSTALLATION

Install SELFIES in the command line using pip:

```
$ pip install selfies
```

1.1 SELFIES Derivation

This section is an informal tutorial on how molecules are derived from a SELFIES. The SELFIES grammar has non-terminal symbols or states

$$X_0, \ldots, X_7, Q$$

Derivation starts with state X_0. The SELFIES is read symbol-by-symbol, with each symbol specifying a grammar rule. SELFIES derivation terminates when no non-terminal symbols remain. In each subsection, we describe a type of SELFIES symbol and the grammar rules associated with it.

1.1.1 Atomic Symbols

Atomic symbols are of the general form `[<B><A>]`, where `<B> in {'', '/', '\\', '=', '#'}` is a prefix representing a bond, and `<A>` is a SMILES symbol representing an atom or ion. If the SMILES symbol is enclosed by square brackets (e.g. `[13C]`), then the square brackets are dropped and `expl` (for "explicit brackets") is appended to obtain `<A>`. For example:

<B>	SMILES symbol	<A>	SELFIES symbol
'='	N	N	[=N]
''	[C@@H]	C@@Hexpl	[C@@Hexpl]
'/'	[O+]	O+expl	[/O+expl]

Let atomic symbol `[<B><A>]` be given, where `<B>` is a prefix representing a bond with multiplicity β and `<A>` is an atom that can make α bonds maximally. The atomic symbol maps:

$$X_i \rightarrow \begin{cases} \texttt{<B'><A>} & \alpha - \mu = 0 \\ \texttt{<B'><A>}X_{\alpha-\mu} & \alpha - \mu \neq 0 \end{cases}$$

where `<B'>` is a prefix representing a bond with multiplicity $\mu = \min(\beta, \alpha, i)$, or the empty string if $\mu = 0$. Note that non-terminal states X_i effectively restrict the subsequent bond to a multiplicity of at most i. We provide an example of the derivation of the SELFIES `[F][=C][=C][#N]`:

$$X_0 \rightarrow \texttt{F}X_1 \rightarrow \texttt{FC}X_3 \rightarrow \texttt{FC=C}X_2 \rightarrow \texttt{FC=C=N}$$

Discussion: Intuitively, the formal grammar has the following behaviour. An atomic symbol `[<B><A>]` connects atom `<A>` to the previously derived atom through bond type `<B>`. If creating this bond would violate the bond constraints of the previous or current atom, the bond multiplicity is reduced (minimally) such that all bond constraints are fulfilled.

Examples:

Example	SELFIES	SMILES
1	`[C][=C][C][#C][13Cexpl]`	`C=CC#C[13C]`
2	`[C][F][C][C][C][C]`	`CF`
3	`[C][O][=C][#O][C][F]`	`COC=O`

1.1.2 Index Symbols

The state Q is used to derive the size of branches and the location of ring bonds. After a ring or branch symbol, the subsequent one or more SELFIES symbols are used to derive an integer from Q. Note that the specific branch and ring symbol itself will specify exactly how many symbols are used in the derivation (e.g. `[Ring3]` indicates that the subsequent three symbols are used).

First, each subsequent symbol s_i is converted to an index $\text{idx}(s_i)$, according to the following assignment:

Index	Symbol	Index	Symbol
0	`[C]`	8	`[Branch2_3]`
1	`[Ring1]`	9	`[O]`
2	`[Ring2]`	10	`[N]`
3	`[Branch1_1]`	11	`[=N]`
4	`[Branch1_2]`	12	`[=C]`
5	`[Branch1_3]`	13	`[#C]`
6	`[Branch2_1]`	14	`[S]`
7	`[Branch2_2]`	15	`[P]`
All other symbols assigned index 0.			

Then Q is mapped to the hexadecimal (base 16) integer specified by the indices. For example, if three symbols s_1, s_2, s_3 are used in the derivation, then Q is mapped to:

$$Q \rightarrow (\text{idx}(s_1) \times 16^2) + (\text{idx}(s_2) \times 16) + \text{idx}(s_3)$$

For example, `[Ring3][C][Branch1_1][O]` will derive the number $(039)_{16} = 57$.

1.1.3 Branch Symbols

Branch symbols are of the general form `[Branch<L>_<M>]`, where `<L>, <M> in {1, 2, 3}`. A branch symbol specifies a branch from the main chain, analogous to the open and closed curved brackets in SMILES. In SELFIES, a branch is derived by a recursive call to the SELFIES derivation.

A Branch symbol `[Branch<L>_<M>]` maps:

$$X_i \rightarrow \begin{cases} X_i & i \leq 1 \\ B(Q, X_n)X_j & i > 1 \end{cases}$$

where $n = \min(i-1, \texttt{<M>})$ is the derivation state of a new branch, and $j = i - n$ is the new derivation state of the main chain. In the $i > 1$ case, the `<L>` subsequent symbols are used to derive an integer from the state Q. Then

$B(Q, X_n)$ takes the next $Q + 1$ symbols, and recursively derives them with initial derivation state X_n. The resulting fragment is taken to be the derived branch, and derivation proceeds with the next derivation state X_j.

Discussion: Intuitively, branch symbols are skipped for states X_{0-1} because the previous atom can make at most one bond (branches require at least two bonds to be free). It is possible that a branch is nested at the start of another branch; in SELFIES, both branches will be connected to the same main chain atom (see Example 5 below).

Examples:

Example	SELFIES	$Q+1$	SMILES
1	`[C][Branch1_1][C][F][Cl]`	1	`C(F)Cl`
2	`[C][Branch1_2][Ring2][=C][C][C][Cl]`	3	`C(=CCC)Cl`
3	`[S][Branch1_2][C][=O][Branch1_2][C][=O][Branch1_1][C][O-expl][O-expl]`	1, 1, 1	`S(=O)(=O)([O-])[O-]`
4	`[C][Branch2_1][Ring1][Branch1_2][C][C][C][C][C][C][C][C][C][C][C][C][C][C][C][C][C][C][F]`	21	`C(CC...CC)F`
	Example 4 has a single branch of 21 carbon atoms.		
5	`[C][Branch1_2][Branch1_1][Branch1_1][C][C][Cl][F]`	4, 1	`C(C)(Cl)F`

1.1.4 Ring Symbols

Ring symbols are of the general form `[Ring<L>]` or `[Expl<B>Ring<L>]`, where `<L> in {1, 2, 3}` and `<B> in {'/', '\\', '=', '#'}` is a prefix representing a bond. A ring symbol specifies a ring bond between two atoms, analogous to the ring numbering digits in SMILES.

A Ring symbol `[Ring<L>]` maps:

$$X_i \rightarrow \begin{cases} X_i & i = 0 \\ R(Q)X_i & i \neq 0 \end{cases}$$

In the $i \neq 0$ case, the `<L>` subsequent symbols are used to derive an integer from the state Q. Then $R(Q)$ connects the *current* atom to the $(Q + 1)$-th preceding atom through a single bond. More specifically, the *current* atom is the most recently derived atom within the current derivation instance (see Example 5 below). If the *current* atom is the m-th derived atom, then a bond is made between the m-th derived atom and the n-th derived atom, where $n = \max(1, m - (Q + 1))$.

The Ring symbol `[Expl<B>Ring<L>]` has an equivalent function to `[Ring<L>]`, except that it connects the current and $(Q + 1)$-th preceding atom through a bond of type `<B>`.

Discussion: In practice, ring bonds are created during a second pass, after all atoms and branches have been derived. The candidate ring bonds are temporarily stored in a queue, and then made in the order that they appear in the SELFIES. A ring bond will be made if its connected atoms can make the ring bond without violating any bond constraints. This is the only non-local rule in SELFIES, but is efficiently implemented as this number can be determined only by looking at one location.

It is also possible that the current atom is already bonded to the $(Q+1)$-th preceding atom, e.g. if $Q = 0$. In this case, the multiplicity of the existing bond is increased by the multiplicity of the ring bond candidate. Then the multiplicity of the resulting bond is reduced (minimally) such that no bond constraints are violated, and the multiplicity is at most 3 (see Example 6 below).

Examples:

Example	SELFIES	$Q+1$	SMILES
1	`[C][=C][C][=C][C][=C][Ring1][Branch1_2]`	5	`C1=CC=CC=C1`
2	`[C][C][=C][C][=C][C][Expl=Ring1][Branch1_2]`	5	`C=1C=CC=CC=1`
3	`[C][C][Expl=Ring1][C]`	1	`C#C`
4	`[C][C][C][C][C][C][C][C][C][C][C][C][C][C][C][C][C][C][C][C][C][C][Ring2][Ring1][Branch1_2]`	21	`C1CC...CC1`
	Example 4 is a single carbon ring of 22 carbon atoms.		
5	`[C][C][C][C][Branch1_1][C][C][Ring1][Ring2][C][C]`	3	`C1CCC1(C)CC`
	Note that the SMILES `CC1CC(C1)CC` is not outputted.		
6	`[C][C][C][C][Expl=Ring1][Ring2][Expl#Ring1][Ring2]`	3, 3	`C#1CCC#1`

1.1.5 Special Symbols

The following are symbols that have a special meaning for SELFIES:

Character	Description
`[epsilon]`	The `[epsilon]` symbol maps $X_0 \to X_0$ and $X_i \to \epsilon$ (the empty string) for all $i \geq 1$.
`[nop]`	The nop (no operation) symbol is always ignored and skipped over by *`selfies.decoder()`*. Thus, it can be used as a padding symbol for SELFIES.
`.`	The dot symbol is used to indicate disconnected or ionic compounds, similar to how it is used in SMILES.

1.2 SELFIES Examples

[1]:
```
import sys
sys.path.append('../')

import selfies as sf
from rdkit import Chem
```

1.2.1 Standard Usage

First let's try translating from SMILES to SELFIES, and then from SELFIES to SMILES. We will use a non-fullerene acceptor for organic solar cells as an example.

[2]:
```
smiles = "CN1C(=O)C2=C(c3cc4c(s3)-c3sc(-c5ncc(C#N)s5)cc3C43OCCO3)N(C)C(=O)" \
         "C2=C1c1cc2c(s1)-c1sc(-c3ncc(C#N)s3)cc1C21OCCO1"
encoded_selfies = sf.encoder(smiles)  # SMILES --> SEFLIES
decoded_smiles = sf.decoder(encoded_selfies)  # SELFIES --> SMILES

print(f"Original SMILES: {smiles}")
print(f"Translated SELFIES: {encoded_selfies}")
print(f"Translated SMILES: {decoded_smiles}")
```

```
Original SMILES: CN1C(=O)C2=C(c3cc4c(s3)-c3sc(-c5ncc(C
↪#N)s5)cc3C43OCCO3)N(C)C(=O)C2=C1c1cc2c(s1)-c1sc(-c3ncc(C#N)s3)cc1C21OCCO1
Translated SELFIES: [C][N][C][Branch1_2][C][=O][C][=C][Branch2_1][Ring2][Branch1_
↪3][C][=C][C][=C][Branch1_1][Ring2][S][Ring1][Branch1_1][C][S][C][Branch1_
↪1][N][C][=N][C][=C][Branch1_1][Ring1][C][#N][S][Ring1][Branch1_
↪3][=C][C][Expl=Ring1][N][C][Ring1][S][O][C][C][O][Ring1][Branch1_1][N][Branch1_
↪1][C][C][C][Branch1_
↪2][C][=O][C][Ring2][Ring1][=N][=C][Ring2][Ring1][P][C][=C][C][=C][Branch1_
↪1][Ring2][S][Ring1][Branch1_1][C][S][C][Branch1_1][N][C][=N][C][=C][Branch1_
↪1][Ring1][C][#N][S][Ring1][Branch1_
↪3][=C][C][Expl=Ring1][N][C][Ring1][S][O][C][C][O][Ring1][Branch1_1]
Translated SMILES: CN7C(=O)C6=C(C1=CC4=C(S1)C=3SC(C2=NC=C(C
↪#N)S2)=CC=3C45OCCO5)N(C)C(=O)C6=C7C8=CC%11=C(S8)C=%10SC(C9=NC=C(C#N)S9)=CC=%10C%11
↪%12OCCO%12
```

When comparing the original and decoded SMILES, do not use == equality. Use RDKit to check whether both SMILES correspond to the same molecule.

[3]:
```
print(f"== Equals: {smiles == decoded_smiles}")

# Recomended
can_smiles = Chem.CanonSmiles(smiles)
can_decoded_smiles = Chem.CanonSmiles(decoded_smiles)
print(f"RDKit Equals: {can_smiles == can_decoded_smiles}")
```

```
== Equals: False
RDKit Equals: True
```

1.2.2 Advanced Usage

Now let's try to customize the SELFIES constraints. We will first look at the default SELFIES semantic constraints.

[4]:
```
default_constraints = sf.get_semantic_constraints()
print(f"Default Constraints:\n {default_constraints}")
```

```
Default Constraints:
 {'H': 1, 'F': 1, 'Cl': 1, 'Br': 1, 'I': 1, 'O': 2, 'N': 3, 'C': 4, 'P': 5, 'S': 6, '?
↪': 8}
```

We have two compounds here, CS=CC#S and [Li]=CC in SELFIES form. Under the default SELFIES settings, they are translated like so. Note that since Li is not recognized by SELFIES, it is constrained to 8 bonds by default.

[5]:
```
c_s_compound = sf.encoder("CS=CC#S")
li_compound = sf.encoder("[Li]=CC")

print(f"CS=CC#S --> {sf.decoder(c_s_compound)}")
print(f"[Li]=CC --> {sf.decoder(li_compound)}")
```

```
CS=CC#S --> CS=CC#S
[Li]=CC --> [Li]=CC
```

We can add Li to the SELFIES constraints, and restrict it to 1 bond only. We can also restrict S to 2 bonds (instead of its default 6). After setting the new constraints, we can check to see if they were updated.

[6]:
```
new_constraints = default_constraints
new_constraints['Li'] = 1
new_constraints['S'] = 2
```

(continues on next page)

(continued from previous page)

```
sf.set_semantic_constraints(new_constraints)  # update constraints

print(f"Updated Constraints:\n {sf.get_semantic_constraints()}")
```

```
Updated Constraints:
 {'H': 1, 'F': 1, 'Cl': 1, 'Br': 1, 'I': 1, 'O': 2, 'N': 3, 'C': 4, 'P': 5, 'S': 2, '?
↪': 8, 'Li': 1}
```

Under our new settings, our previous molecules are translated like so. Notice that our new semantic constraints are met.

[7]:
```
print(f"CS=CC#S --> {sf.decoder(c_s_compound)}")
print(f"[Li]=CC --> {sf.decoder(li_compound)}")
```

```
CS=CC#S --> CSCC=S
[Li]=CC --> [Li]CC
```

To revert back to the default constraints, simply call:

[8]:
```
sf.set_semantic_constraints()
```

1.3 Code Documentation

1.3.1 Standard Functions

selfies.**encoder**(*smiles*, *print_error=False*)

Translates a SMILES into a SELFIES.

The SMILES to SELFIES translation occurs independently of the SELFIES alphabet and grammar. Thus, *selfies.encoder()* will work regardless of the alphabet and grammar rules that selfies is operating on, assuming the input is a valid SMILES. Additionally, *selfies.encoder()* preserves the atom and branch order of the input SMILES; thus, one could generate random SELFIES corresponding to the same molecule by generating random SMILES, and then translating them.

However, encoding and then decoding a SMILES may not necessarily yield the original SMILES. Reasons include:

1. SMILES with aromatic symbols are automatically Kekulized before being translated.
2. SMILES that violate the bond constraints specified by selfies will be successfully encoded by *selfies.encoder()*, but then decoded into a new molecule that satisfies the constraints.
3. The exact ring numbering order is lost in *selfies.encoder()*, and cannot be reconstructed by *selfies.decoder()*.

Finally, note that *selfies.encoder()* does **not** check if the input SMILES is valid, and should not be expected to reject invalid inputs. It is recommended to use RDKit to first verify that the SMILES are valid.

Parameters

- **smiles** (str) – The SMILES to be translated.
- **print_error** (bool) – If True, error messages will be printed to console. Defaults to False.

Return type Optional[str]

Returns the SELFIES translation of `smiles`. If an error occurs, and `smiles` cannot be translated, `None` is returned instead.

Example

```
>>> import selfies
>>> selfies.encoder('C=CF')
'[C][=C][F]'
```

Note: Currently, *`selfies.encoder()`* does not support the following types of SMILES:

- SMILES using ring numbering across a dot-bond symbol to specify bonds, e.g. `C1.C2.C12` (propane) or `c1cc([O-].[Na+])ccc1` (sodium phenoxide).
- SMILES with ring numbering between atoms that are over `16 ** 3 = 4096` atoms apart.
- SMILES using the wildcard symbol `*`.
- SMILES using chiral specifications other than `@` and `@@`.

`selfies.`**`decoder`**(*selfies*, *print_error=False*)

Translates a SELFIES into a SMILES.

The SELFIES to SMILES translation operates based on the `selfies` grammar rules, which can be configured using *`selfies.set_semantic_constraints()`*. Given the appropriate settings, the decoded SMILES will always be syntactically and semantically correct. That is, the output SMILES will satisfy the specified bond constraints. Additionally, *`selfies.decoder()`* will attempt to preserve the atom and branch order of the input SELFIES.

Parameters

- **`selfies`** (`str`) – The SELFIES to be translated.
- **`print_error`** (`bool`) – If True, error messages will be printed to console. Defaults to False.

Return type `Optional[str]`

Returns the SMILES translation of `selfies`. If an error occurs, and `selfies` cannot be translated, `None` is returned instead.

Example

```
>>> import selfies
>>> selfies.decoder('[C][=C][F]')
'C=CF'
```

`selfies.`**`len_selfies`**(*selfies*)

Computes the symbol length of a SELFIES.

The symbol length is the number of symbols that make up the SELFIES, and not the length of the string itself (i.e. `len(selfies)`).

Parameters **`selfies`** (`str`) – A SELFIES.

Return type `int`

Returns The symbol length of `selfies`.

Example

```
>>> import selfies
>>> selfies.len_selfies('[C][O][C]')
3
>>> selfies.len_selfies('[C][=C][F].[C]')
5
```

selfies.**split_selfies**(*selfies*)

Splits a SELFIES into its symbols.

Returns an iterable that yields the symbols of a SELFIES one-by-one in the order they appear in the string. SELFIES symbols are always either indicated by an open and closed square bracket, or are the `'.'` dot-bond symbol.

Parameters **selfies** (`str`) – The SELFIES to be read.

Return type `Iterable[str]`

Returns An iterable of the symbols of `selfies` in the same order they appear in the string.

Example

```
>>> import selfies
>>> list(selfies.split_selfies('[C][O][C]'))
['[C]', '[O]', '[C]']
>>> list(selfies.split_selfies('[C][=C][F].[C]'))
['[C]', '[=C]', '[F]', '.', '[C]']
```

selfies.**get_alphabet_from_selfies**(*selfies_iter*)

Constructs an alphabet from an iterable of SELFIES.

From an iterable of SELFIES, constructs the minimum-sized set of SELFIES symbols such that every SELFIES in the iterable can be constructed from symbols from that set. Then, the set is returned. Note that the symbol `'.'` will not be added as a member of the returned set, even if it appears in the input.

Parameters **selfies_iter** (`Iterable[str]`) – An iterable of SELFIES.

Return type `Set[str]`

Returns The SElFIES alphabet built from the SELFIES in `selfies_iter`.

Example

```
>>> import selfies
>>> selfies_list = ['[C][F][O]', '[C].[O]', '[F][F]']
>>> alphabet = selfies.get_alphabet_from_selfies(selfies_list)
>>> sorted(list(alphabet))
['[C]', '[F]', '[O]']
```

selfies.**get_semantic_robust_alphabet**()

Returns a subset of all symbols that are semantically constrained by `selfies`.

These semantic constraints can be configured with *`selfies.set_semantic_constraints()`*.

Return type `Set[str]`

Returns a subset of all symbols that are semantically constrained.

1.3.2 Advanced Functions

By default, `selfies` operates under the following semantic constraints

Max Bonds	Atom(s)
1	`F`, `Cl`, `Br`, `I`
2	`O`
3	`N`
4	`C`
5	`P`
6	`S`
8	All other atoms

However, the default constraints are inadequate for SMILES that violate them. For example, nitrobenzene `O=N(=O)C1=CC=CC=C1` has a nitrogen with 6 bonds and the chlorate anion `O=Cl(=O)[O-]` has a chlorine with 5 bonds - these SMILES *cannot* be represented as SELFIES under the default constraints. Additionally, users may want to specify their own custom constraints. Thus, we provide the following methods for configuring the semantic constraints of `selfies`.

Warning: SELFIES may be translated differently under different semantic constraints. Therefore, if custom semantic constraints are used, it is recommended to report them for reproducibility reasons.

`selfies.`**`get_semantic_constraints`**`()`

Returns the semantic bond constraints that `selfies` is currently operating on.

Returned is the argument of the most recent call of *`selfies.set_semantic_constraints()`*, or the default bond constraints if the function has not been called yet. Once retrieved, it is copied and then returned. See *`selfies.set_semantic_constraints()`* for further explanation.

Return type `Dict[str, int]`

Returns The bond constraints `selfies` is currently operating on.

`selfies.`**`set_semantic_constraints`**`(`*bond_constraints=None*`)`

Configures the semantic constraints of `selfies`.

The SELFIES grammar is enforced dynamically from a dictionary `bond_constraints`. The keys of the dictionary are atoms and/or ions (e.g. `I`, `Fe+2`). To denote an ion, use the format `E+C` or `E-C`, where `E` is an element and `C` is a positive integer. The corresponding value is the maximum number of bonds that atom or ion can make, between 1 and 8 inclusive. For example, one may have:

- `bond_constraints['I'] = 1`
- `bond_constraints['C'] = 4`

`selfies.decoder()` will only generate SMILES that respect the bond constraints specified by the dictionary. In the example above, both `'[C][=I]'` and `'[I][=C]'` will be translated to `'CI'` and `'IC'` respectively, because `I` has been configured to make one bond maximally.

If an atom or ion is not specified in `bond_constraints`, it will by default be constrained to 8 bonds. To change the default setting for unrecognized atoms or ions, set `bond_constraints['?']` to the desired integer (between 1 and 8 inclusive).

Parameters **`bond_constraints`** (`Optional[Dict[str, int]]`) – a dictionary representing the semantic constraints the updated SELFIES will operate upon. Defaults to `None`; in this case, a default dictionary will be used.

Return type None

Returns None.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX