selfies

Release 2.0.0

Mario Krenn

Oct 21, 2021

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 Tutorial

1.1.1 The Basics

We begin by importing `selfies`.

[1]:
```
import selfies as sf
```

First, let's try translating between SMILES and SELFIES - as an example, we will use benzaldehyde. To translate from SMILES to SELFIES, use the `selfies.encoder` function, and to translate from SMILES back to SELFIES, use the `selfies.decoder` function.

[2]:
```
original_smiles = "O=Cc1ccccc1"  # benzaldehyde

try:

    encoded_selfies = sf.encoder(original_smiles)  # SMILES  -> SELFIES
    decoded_smiles = sf.decoder(encoded_selfies)   # SELFIES -> SMILES

except sf.EncoderError as err:
    pass  # sf.encoder error...
except sf.DecoderError as err:
    pass  # sf.decoder error...
```

[3]:
```
encoded_selfies
```

[3]:
```
'[O][=C][C][=C][C][=C][C][=C][Ring1][=Branch1]'
```

[4]:
```
decoded_smiles
```

[4]:
```
'O=CC1=CC=CC=C1'
```

Note that `original_smiles` and `decoded_smiles` are different strings, but they both represent benzaldehyde. Thus, when comparing the two SMILES strings, string equality should *not* be used. Instead, use RDKit to check whether the SMILES strings represent the same molecule.

[5]:
```
from rdkit import Chem

Chem.CanonSmiles(original_smiles) == Chem.CanonSmiles(decoded_smiles)
```

[5]:
```
True
```

1.1.2 Customizing SELFIES

The SELFIES grammar is derived dynamically from a set of semantic constraints, which assign bonding capacities to various atoms. Let's customize the semantic constraints that `selfies` operates on. By default, the following constraints are used:

[6]:
```
sf.get_preset_constraints("default")
```

[6]:
```
{'H': 1,
 'F': 1,
 'Cl': 1,
 'Br': 1,
 'I': 1,
 'O': 2,
 'O+1': 3,
 'O-1': 1,
 'N': 3,
 'N+1': 4,
 'N-1': 2,
 'C': 4,
 'C+1': 5,
 'C-1': 3,
 'P': 5,
 'P+1': 6,
 'P-1': 4,
 'S': 6,
 'S+1': 7,
 'S-1': 5,
 '?': 8}
```

These constraints map atoms (they keys) to their bonding capacities (the values). The special ? key maps to the bonding capacity for all atoms that are not explicitly listed in the constraints. For example, S and Li are constrained to a maximum of 6 and 8 bonds, respectively. Every SELFIES string can be decoded into a molecule that obeys the current constraints.

[7]:
```
sf.decoder("[Li][=C][C][S][=C][C][#S]")
```

[7]:
```
'[Li]=CCS=CC#S'
```

But suppose that we instead wanted to constrain S and Li to a maximum of 2 and 1 bond(s), respectively. To do so, we create a new set of constraints, and tell `selfies` to operate on them using `selfies.set_semantic_constraints`.

[8]:
```
new_constraints = sf.get_preset_constraints("default")
new_constraints['Li'] = 1
new_constraints['S'] = 2

sf.set_semantic_constraints(new_constraints)
```

To check that the update was succesful, we can use `selfies.get_semantic_constraints`, which returns the semantic constraints that `selfies` is currently operating on.

[9]:
```
sf.get_semantic_constraints()
```

[9]:
```
{'H': 1,
 'F': 1,
 'Cl': 1,
 'Br': 1,
 'I': 1,
 'O': 2,
 'O+1': 3,
 'O-1': 1,
 'N': 3,
 'N+1': 4,
 'N-1': 2,
 'C': 4,
 'C+1': 5,
 'C-1': 3,
 'P': 5,
 'P+1': 6,
 'P-1': 4,
 'S': 2,
 'S+1': 7,
 'S-1': 5,
 '?': 8,
 'Li': 1}
```

Our previous SELFIES string is now decoded like so. Notice that the specified bonding capacities are met, with every S and Li making only 2 and 1 bonds, respectively.

[10]:
```
sf.decoder("[Li][=C][C][S][=C][C][#S]")
```

[10]:
```
'[Li]CCSCC=S'
```

Finally, to revert back to the default constraints, simply call:

[11]:
```
sf.set_semantic_constraints()
```

Please refer to the API reference for more details and more preset constraints.

1.1.3 SELFIES in Practice

Let's use a simple example to show how `selfies` can be used in practice, as well as highlight some convenient utility functions from the library. We start with a toy dataset of SMILES strings. As before, we can use `selfies.encoder` to convert the dataset into SELFIES form.

[12]:
```
smiles_dataset = ["COC", "FCF", "O=O", "O=Cc1ccccc1"]
selfies_dataset = list(map(sf.encoder, smiles_dataset))

selfies_dataset
```

[12]:
```
['[C][O][C]',
 '[F][C][F]',
```

(continues on next page)

(continued from previous page)

```
 '[O][=O]',
 '[O][=C][C][=C][C][=C][C][=C][Ring1][=Branch1]']
```

The function `selfies.len_selfies` computes the symbol length of a SELFIES string. We can use it to find the maximum symbol length of the SELFIES strings in the dataset.

[13]:
```
max_len = max(sf.len_selfies(s) for s in selfies_dataset)
max_len
```

[13]:
```
10
```

To extract the SELFIES symbols that form the dataset, use `selfies.get_alphabet_from_selfies`. Here, we add `[nop]` to the alphabet, which is a special padding character that `selfies` recognizes.

[14]:
```
alphabet = sf.get_alphabet_from_selfies(selfies_dataset)
alphabet.add("[nop]")

alphabet = list(sorted(alphabet))
alphabet
```

[14]:
```
['[=Branch1]', '[=C]', '[=O]', '[C]', '[F]', '[O]', '[Ring1]', '[nop]']
```

Then, create a mapping between the alphabet SELFIES symbols and indices.

[15]:
```
vocab_stoi = {symbol: idx for idx, symbol in enumerate(alphabet)}
vocab_itos = {idx: symbol for symbol, idx in vocab_stoi.items()}

vocab_stoi
```

[15]:
```
{'[=Branch1]': 0,
 '[=C]': 1,
 '[=O]': 2,
 '[C]': 3,
 '[F]': 4,
 '[O]': 5,
 '[Ring1]': 6,
 '[nop]': 7}
```

SELFIES provides some convenience methods to convert between SELFIES strings and label (integer) and one-hot encodings. Using the first entry of the dataset (dimethyl ether) as an example:

[16]:
```
dimethyl_ether = selfies_dataset[0]
label, one_hot = sf.selfies_to_encoding(dimethyl_ether, vocab_stoi, pad_to_len=max_len)
```

[17]:
```
label
```

[17]:
```
[3, 5, 3, 7, 7, 7, 7, 7, 7, 7]
```

[18]:
```
one_hot
```

[18]:
```
[[0, 0, 0, 1, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 1, 0, 0],
 [0, 0, 0, 1, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0, 1],
 [0, 0, 0, 0, 0, 0, 0, 1],
```

(continues on next page)

(continued from previous page)

```
 [0, 0, 0, 0, 0, 0, 0, 1],
 [0, 0, 0, 0, 0, 0, 0, 1],
 [0, 0, 0, 0, 0, 0, 0, 1],
 [0, 0, 0, 0, 0, 0, 0, 1],
 [0, 0, 0, 0, 0, 0, 0, 1]]
```

[21]:
```
dimethyl_ether = sf.encoding_to_selfies(one_hot, vocab_itos, enc_type="one_hot")
dimethyl_ether
```

[21]:
```
'[C][O][C][nop][nop][nop][nop][nop][nop][nop]'
```

[22]:
```
sf.decoder(dimethyl_ether)  # sf.decoder ignores [nop]
```

[22]:
```
'COC'
```

If different encoding strategies are desired, `selfies.split_selfies` can be used to tokenize a SELFIES string into its individual symbols.

[24]:
```
list(sf.split_selfies("[C][O][C]"))
```

[24]:
```
['[C]', '[O]', '[C]']
```

Please refer to the API reference for more details and utility functions.

1.2 API Reference

1.2.1 Core Functions

`selfies.`**`encoder`**(*smiles*, *strict=True*)

Translates a SMILES string into its corresponding SELFIES string.

This translation is deterministic and does not depend on the current semantic constraints. Additionally, it preserves the atom order of the input SMILES string; thus, one could generate randomized SELFIES strings by generating randomized SMILES strings, and then translating them.

By nature of SELFIES, it is impossible to represent molecules that violate the current semantic constraints as SELFIES strings. Thus, we provide the `strict` flag to guard against such cases. If `strict=True`, then this function will raise a *`selfies.EncoderError`* if the input SMILES string represents a molecule that violates the semantic constraints. If `strict=False`, then this function will not raise any error; however, calling *`selfies.decoder()`* on a SELFIES string generated this way will *not* be guaranteed to recover a SMILES string representing the original molecule.

Parameters

- **`smiles`** (`str`) – the SMILES string to be translated. It is recommended to use RDKit to check that the strings passed into this function are valid SMILES strings.
- **`strict`** (`bool`) – if `True`, this function will check that the input SMILES string obeys the semantic constraints. Defaults to `True`.

Return type `str`

Returns a SELFIES string translated from the input SMILES string.

Raises *`EncoderError`* – if the input SMILES string is invalid, cannot be kekulized, or violates the semantic constraints with `strict=True`.

Example

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

Note: This function does not currently support SMILES with:

- The wildcard symbol `*`.
- The quadruple bond symbol `$`.
- Chirality specifications other than `@` and `@@`.
- Ring bonds across a dot symbol (e.g. `c1cc([O-].[Na+])ccc1`) or ring bonds between atoms that are over 4000 atoms apart.

Although SELFIES does not have aromatic symbols, this function *does* support aromatic SMILES strings by internally kekulizing them before translation.

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

Translates a SELFIES string into its corresponding SMILES string.

This translation is deterministic but depends on the current semantic constraints. The output SMILES string is guaranteed to be syntatically correct and guaranteed to represent a molecule that obeys the semantic constraints.

Parameters

- **`selfies`** (`str`) – the SELFIES string to be translated.
- **`compatible`** (`bool`) – if `True`, this function will accept SELFIES strings containing depreciated symbols from previous releases. However, this function may behave differently than in previous major relases, and should not be treated as backard compatible. Defaults to `False`.

Return type `str`

Returns a SMILES string derived from the input SELFIES string.

Raises ***`DecoderError`*** – if the input SELFIES string is malformed.

Example

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

1.2.2 Customization Functions

The SELFIES grammar is derived dynamically from a set of semantic constraints, which assign bonding capacities to various atoms. By default, `selfies` operates under the following constraints:

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

The +1 and -1 charged versions of O, N, C, S, and P are also constrained, where a +1 increases the bonding capacity of the neutral atom by 1, and a -1 decreases the bonding capacity of the neutral atom by 1. For example, N+1 has a bonding capacity of $3 + 1 = 4$, and N-1 has a bonding capacity of $3 - 1 = 2$. The charged versions B+1 and B-1 are constrained to a capacity of 2 and 4 bonds, respectively.

However, the default constraints are inadequate for SMILES strings 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 strings *cannot* be represented by SELFIES strings 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 strings 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_preset_constraints`**`(`*name*`)`

Returns the preset semantic constraints with the given name.

Besides the aforementioned default constraints, `selfies` offers other preset constraints for convenience; namely, constraints that enforce the octet rule and constraints that accommodate hypervalent molecules.

The differences between these constraints can be summarized as follows:

	Cl, Br, I	N	P	P+1	P-1	S	S+1	S-1
`default`	1	3	5	6	4	6	7	5
`octet_rule`	1	3	3	4	2	2	3	1
`hypervalent`	7	5	5	6	4	6	7	5

Parameters **`name`** (`str`) – the preset name: `default` or `octet_rule` or `hypervalent`.

Return type `Dict[str, int]`

Returns the preset constraints with the specified name, represented as a dictionary which maps atoms (the keys) to their bonding capacities (the values).

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

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

Return type `Dict[str, int]`

Returns the current semantic constraints, represented as a dictionary which maps atoms (the keys) to their bonding capacities (the values).

`selfies.`**`set_semantic_constraints`**`(`*bond_constraints='default'*`)`

Updates the semantic constraints that `selfies` operates on.

If the input is a string, the new constraints are taken to be the preset named `bond_constraints` (see *selfies.get_preset_constraints()*).

Otherwise, the input is a dictionary representing the new constraints. This dictionary maps atoms (the keys) to non-negative bonding capacities (the values); the atoms are specified by strings of the form `E` or `E+C` or `E-C`, where `E` is an element symbol and `C` is a positive integer. For example, one may have:

- `bond_constraints["I-1"] = 0`
- `bond_constraints["C"] = 4`

This dictionary must also contain the special `?` key, which indicates the bond capacities of all atoms that are not explicitly listed in the dictionary.

Parameters **bond_constraints** (`Union[str, Dict[str, int]]`) – the name of a preset, or a dictionary representing the new semantic constraints.

Return type `None`

Returns `None`.

1.2.3 Utility Functions

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

Returns the number of symbols in a given SELFIES string.

Parameters **selfies** (`str`) – a SELFIES string.

Return type `int`

Returns the symbol length of the SELFIES string.

Example

```
>>> import selfies as sf
>>> sf.len_selfies("[C][=C][F].[C]")
5
```

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

Tokenizes a SELFIES string into its individual symbols.

Parameters **selfies** (`str`) – a SELFIES string.

Return type `Iterator[str]`

Returns the symbols of the SELFIES string one-by-one with order preserved.

Example

```
>>> import selfies as sf
>>> list(sf.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 strings.

The returned alphabet is the set of all symbols that appear in the SELFIES strings from the input iterable, minus the dot `.` symbol.

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

Return type `Set[str]`

Returns an alphabet of SELFIES symbols, built from the input iterable.

Example

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

selfies.**get_semantic_robust_alphabet**()

Returns a subset of all SELFIES symbols that are constrained by `selfies` under the current semantic constraints.

Return type `Set[str]`

Returns a subset of all SELFIES symbols that are semantically constrained.

selfies.**selfies_to_encoding**(*selfies*, *vocab_stoi*, *pad_to_len=- 1*, *enc_type='both'*)

Converts a SELFIES string into its label (integer) and/or one-hot encoding.

A label encoded output will be a list of shape `(L,)` and a one-hot encoded output will be a 2D list of shape `(L, len(vocab_stoi))`, where `L` is the symbol length of the SELFIES string. Optionally, the SELFIES string can be padded before it is encoded.

Parameters

- **selfies** (`str`) – the SELFIES string to be encoded.
- **vocab_stoi** (`Dict[str, int]`) – a dictionary that maps SELFIES symbols to indices, which must be non-negative and contiguous, starting from 0. If the SELFIES string is to be padded, then the special padding symbol `[nop]` must also be a key in this dictionary.
- **pad_to_len** (`int`) – the length that the SELFIES string string is padded to. If this value is less than or equal to the symbol length of the SELFIES string, then no padding is added. Defaults to `-1`.
- **enc_type** (`str`) – the type of encoding of the output: `label` or `one_hot` or `both`. If this value is `both`, then a tuple of the label and one-hot encodings is returned. Defaults to `both`.

Return type `Union[List[int], List[List[int]], Tuple[List[int], List[List[int]]]]`

Returns the label encoded and/or one-hot encoded SELFIES string.

Example

```
>>> import selfies as sf
>>> sf.selfies_to_encoding("[C][F]", {"[C]": 0, "[F]": 1})
([0, 1], [[1, 0], [0, 1]])
```

selfies.**encoding_to_selfies**(*encoding*, *vocab_itos*, *enc_type*)

Converts a label (integer) or one-hot encoding into a SELFIES string.

If the input is label encoded, then a list of shape `(L,)` is expected; and if the input is one-hot encoded, then a 2D list of shape `(L, len(vocab_itos))` is expected.

Parameters

- **encoding** (`Union[List[int], List[List[int]]]`) – a label or one-hot encoding.
- **vocab_itos** (`Dict[int, str]`) – a dictionary that maps indices to SELFIES symbols. The indices of this dictionary must be non-negative and contiguous, starting from 0.
- **enc_type** (`str`) – the type of encoding of the input: `label` or `one_hot`.

Return type str

Returns the SELFIES string represented by the input encoding.

Example

```
>>> import selfies as sf
>>> one_hot = [[0, 1, 0], [0, 0, 1], [1, 0, 0]]
>>> vocab_itos = {0: "[nop]", 1: "[C]", 2: "[F]"}
>>> sf.encoding_to_selfies(one_hot, vocab_itos, enc_type="one_hot")
'[C][F][nop]'
```

selfies.**batch_selfies_to_flat_hot**(*selfies_batch*, *vocab_stoi*, *pad_to_len=-1*)

Converts a list of SELFIES strings into its list of flattened one-hot encodings.

Each SELFIES string in the input list is one-hot encoded (and then flattened) using *selfies.selfies_to_encoding()*, with vocab_stoi and pad_to_len being passed in as arguments.

Parameters

- **selfies_batch** (List[str]) – the list of SELFIES strings to be encoded.
- **vocab_stoi** (Dict[str, int]) – a dictionary that maps SELFIES symbols to indices.
- **pad_to_len** (int) – the length that each SELFIES string in the input list is padded to. Defaults to -1.

Return type List[List[int]]

Returns the flattened one-hot encodings of the input list.

Example

```
>>> import selfies as sf
>>> batch = ["[C]", "[C][C]"]
>>> vocab_stoi = {"[nop]": 0, "[C]": 1}
>>> sf.batch_selfies_to_flat_hot(batch, vocab_stoi, 2)
[[0, 1, 1, 0], [0, 1, 0, 1]]
```

selfies.**batch_flat_hot_to_selfies**(*one_hot_batch*, *vocab_itos*)

Converts a list of flattened one-hot encodings into a list of SELFIES strings.

Each encoding in the input list is unflattened and then decoded using *selfies.encoding_to_selfies()*, with vocab_itos being passed in as an argument.

Parameters

- **one_hot_batch** (List[List[int]]) – a list of flattened one-hot encodings. Each encoding must be a list of length divisible by len(vocab_itos).
- **vocab_itos** (Dict[int, str]) – a dictionary that maps indices to SELFIES symbols.

Return type List[str]

Returns the list of SELFIES strings represented by the input encodings.

Example

```
>>> import selfies as sf
>>> batch = [[0, 1, 1, 0], [0, 1, 0, 1]]
>>> vocab_itos = {0: "[nop]", 1: "[C]"}
>>> sf.batch_flat_hot_to_selfies(batch, vocab_itos)
['[C][nop]', '[C][C]']
```

1.2.4 Exceptions

exception selfies.**EncoderError**
Exception raised by *selfies.encoder()*.

exception selfies.**DecoderError**
Exception raised by *selfies.decoder()*.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

INDEX