vgram Documentation

Aleksandr Khvorov

Jul 04, 2019

Contents

CHAPTER 1

About

Vgram is the implementation of the new method for constructing an optimal feature set from sequential data. It creates a dictionary of n-grams of variable length, based on the minimum description length principle. The method is a dictionary coder and works simultaneously as both a compression algorithm and as unsupervised feature extraction. The length of constructed v-grams is not limited by any bound and exceeds 100 characters in provided experiments. Constructed v-grams can be used for any sequential data analysis and allows transfer bag-of-word techniques to non-text data types. Extracted features generate a practical basis for text classification, that shows competitive results on standard text classification collections without using the text structure. Combining extracted character v-grams with the words from the original text we achieved substantially better classification quality than on words or v-grams alone.

See the CIKM '18 paper for details.

Igor Kuralenok, Natalia Starikova, Aleksandr Khvorov, and Julian Serdyuk. Construction of Efficient V-Gram Dictionary for Sequential Data Analysis, CIKM '18 Proceedings of the 27th ACM International Conference on Information and Knowledge Management, Pages 1343-1352

CHAPTER 2

Install

Install vgram by running:

```
pip install vgram
```

Maybe you keep some errors about not installed pybind11, but it is okay.

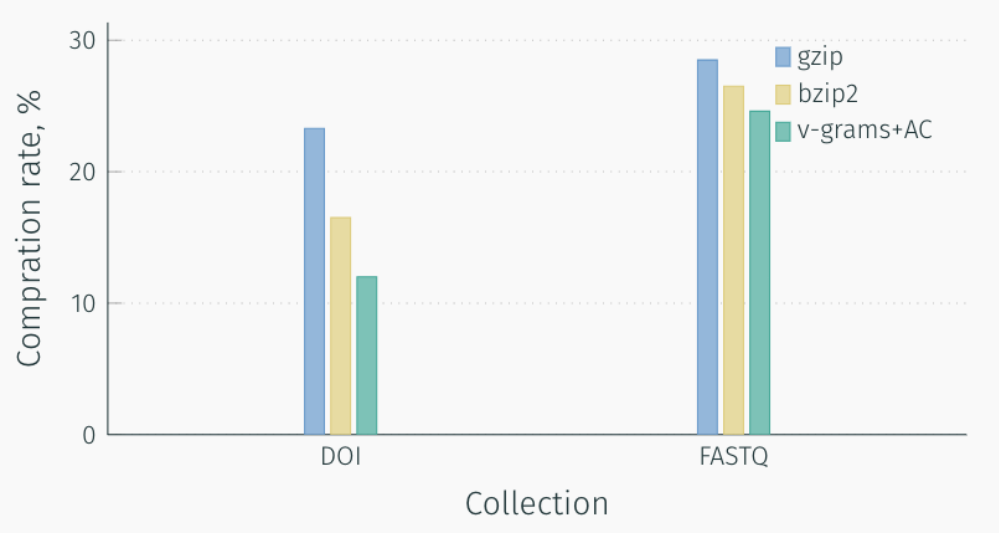
CHAPTER 3

Contents:

3.1 Use cases and theory

3.1.1 Data compression

V-Grams dictionary building based on minimum description length principle. This allows us to achieve competitive results in data compression tasks. Comparison of v-gram algorithm with other popular compression algorithms on FASTQ and DOI URLs 2013 datasets has presented below. We have used arithmetic coding (AC) as an entropy encoder.

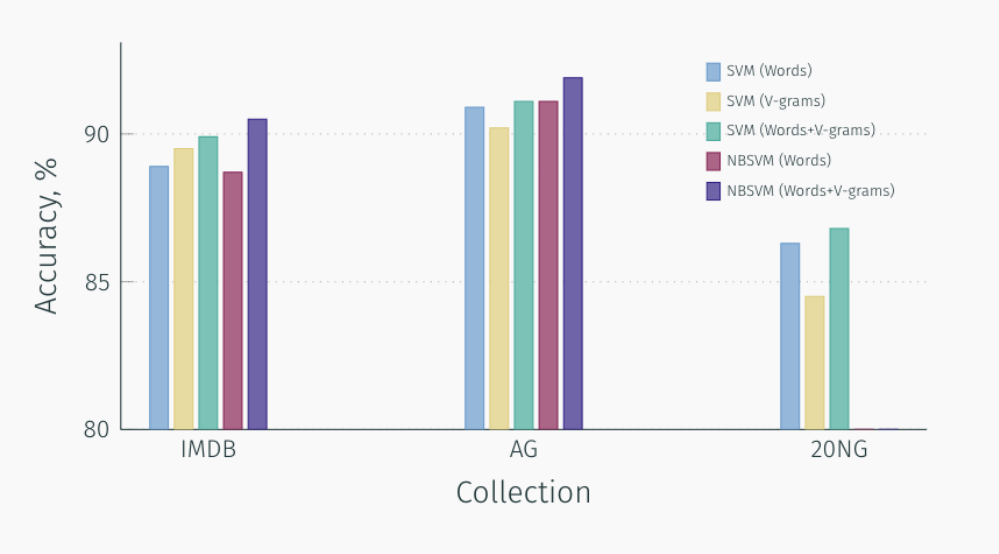


3.1.2 Text classification

Dictionary of v-grams may be used as words in bag-of-words text representation. From the practical perspective, v-grams has at least two significant advantages over words: straightforward text normalization and fixed dictionary

volume. V-gram dictionary can be much smaller than words dictionary but demonstrate the same score. At IMDb dataset classification dictionary of only 5000 vgrams reach better score then 75000 words (15 times smaller).

In our experiments, we tested v-grams on three datasets: 20 News Groups, IMDb movie reviews, AG news. Bag-of-words, bag-of-vgrams, and bag-of-(words&vgrams) were compared. A text was normalized in the following way: all non-alphanumeric symbols were removed, then the text was converted to lower case. Experiments show that even small vgram dictionary extract additional information and significantly increase the score.



Also, bag-of-words methods can be applied to non-textual data such as genome sequences or text without words splitting like text in Asian or Arabic languages.

3.1.3 Sequences analysis

V-Grams dictionary contains a lot of valuable information about the domain. The following are examples of the resulting v-gram for two datasets: 20 newsgroups and IMDb movie database.

V-grams (IMDB)	V-grams (20 Newsgroups)
j	j
ja	ja
jack	jack
jackass	jacket
jackblack	jackson
jacket	jacob
jackfrost	jadetuftsedu
jackie	jagr
jackiechan	jaguar
jacklemmon	jail
jacknicholson	jake

As you can see v-grams reflects the specifics of the data well. Also, such analysis may be useful for more difficult to interpret data as genome sequences.

3.2 V-Gram construction

3.2.1 VGram

For working with texts use `VGram`. This class implement the scikit-learn fit-transform interface and can be well embedded in the existing code.

```
class VGram(StreamVGram):
    def __init__(self, size=10000, iter_num=10, verbose=0): ..
    def fit(self, X): ..
    def transform(self, X): ..
    def save(self, filename="vgram_dict.json"): ..
    def alphabet(self): ..
    def freqs(self): ..
```

`VGram(size=10000, iter_num=10, verbose=0)` construct object, which can learn dictionary of size `size` in `iter_num` iterations.
By default CharTokenizer is used (see *Tokenizers*).
`verbose` means a level of verbose. `0` not print anything, `1` or more print some useful information about v-gram learning process.

`fit(X)` consume a list of strings and return VGram object. It makes `iter_num` iterations on all data to fit dictionary better. One iteration often is not enough.

`transform(X)` consume a list of strings. Return a 2-d list of strings, where each row is partition of original string on v-grams. It's good for pipeline where CountVectorizer follows VGram (see *Examples*).

`save(filename="", tokenizer=None)` consume filename where dictionary will be saved (see save).

`alphabet()` return a list of v-grams.

`freqs()` return a list of integers with frequencies of v-gram occurrence in data.

Load VGram from file by `loadVGram`. `loadVGram` get file name and return VGram object.

If you work with streams, use StreamVGram (see *Stream V-Gram construction*). If you work with integer sequences, use *IntVGram_*

See *Examples* for details.

3.2.2 IntVGram

For working with integers sequences use `VGram`. It's similar to *VGram* but works with integers. This class implement the scikit-learn fit-transform interface and can be well embedded in the existing code.

```
class IntVGram(StreamVGram):
    def __init__(self, size=10000, iter_num=10, verbose=0): ..
    def fit(self, X): ..
    def transform(self, X): ..
    def save(self, filename="vgram_dict.json"): ..
    def alphabet(self): ..
    def freqs(self): ..
```

`IntVGram(size=10000, iter_num=10, verbose=0)` construct object, which can learn dictionary of size `size` in `iter_num` iterations.
`verbose` means a level of verbose. `0` not print anything, `1` or more print some useful information about v-gram learning process.

`fit(X)` consume a 2-list of integers and return `IntVGram` object. It makes `iter_num` iterations on all data to fit dictionary better. One iteration often is not enough.

`transform(X)` consume a 2-d list of integers or 2-d numpy array. Return a list of strings, where each row is consist of integers separate by stace. Each integer is id of v-gram in dictionary. It's good for pipeline where CountVectorizer follows IntVGram even if you work with integers (see *Examples*). You can get integer list by apply `split()` to every element of returned list.

`save(filename="vgram_dict.json")` consume filename where dictionary will be saved (see save).

`alphabet()` return a 2-d list of integers where every row is a list of index of the corresponding v-gram.

`freqs()` return a list of integers with frequencies of v-gram occurrence in data.

Load IntVGram from file by `loadIntVGram`. `loadIntVGram` get file name and return IntVGram object.

If you work with streams, use IntStreamVGram (see *Stream V-Gram construction*).

See *Examples* for details.

3.3 Stream V-Gram construction

3.3.1 StreamVGram

For working with streaming texts use `StreamVGram`.

```
class StreamVGram:
    def __init__(self, size=10000, verbose=0): ..
    def accept(self, X): ..
    def parse(self, X): ..
    def update(self): ..
    def save(self, filename="vgram_dict.json"): ..
    def alphabet(self): ..
    def freqs(self): ..
```

`StreamVGram(size=10000, verbose=0)` construct object, which can learn dictionary of size `size` in stream mode.
By default CharTokenizer is used (see *Tokenizers*).
`verbose` means a level of verbose. `0` not print anything, `1` or more print some useful information about v-gram learning process.

`accept(x)` consume a strings and not return value.

`parse(x)` consume a strings and return v-gram representation of it.

`update()` after accepting and before parsing make update of dictionary. Otherwise, you will use unfitted dictionary.

`save(filename="vgram_dict.json")` consume filename where dictionary will be saved (see save).

`alphabet()` return a list of v-grams.

`freqs()` return a list of integers with frequencies of v-gram occurrence in data.

Load StreamVGram from file by `loadStreamVGram`. `loadStreamVGram` get file name and return StreamV-Gram object.

If your text collection is not so big and comes ones, more convinient use VGram (see *V-Gram construction*). If you work with integer sequences, use *IntStreamVGram_*

See *Examples* for details.

3.3.2 IntStreamVGram

For working with integers streaming sequences use `IntStreamVGram`. It's similar to *StreamVGram* but works with integers.

```
class IntStreamVGram(StreamVGram):
    def __init__(self, size=10000, verbose=0): ..
    def accept(self, X): ..
    def parse(self, X): ..
    def update(self): ..
    def save(self, filename="vgram_dict.json"): ..
    def alphabet(self): ..
    def freqs(self): ..
```

`IntStreamVGram(size=10000, verbose=0)` construct object, which can learn dictionary of size `size`. `verbose` means a level of verbose. `0` not print anything, `1` or more print some useful information about v-gram learning process.

`accept(x)` consume a list of integers and not return value.

`parse(x)` consume a list of integers and return integer list of v-gram's id. It's list of indices for alphabet.

`update()` after accepting and before parsing make update of dictionary. Otherwise, you will use unfitted dictionary.

`save(filename="vgram_dict.json")` consume filename where dictionary will be saved (see save).

`alphabet()` return a 2-d list of integers where every row is v-gram.

`freqs()` return a list of integers with frequencies of v-gram occurrence in data.

Load IntStreamVGram from file by `loadIntStreamVGram`. `loadIntStreamVGram` get file name and return IntStreamVGram object.

If you not work with streams, use IntVGram (see *V-Gram construction*).

See *Examples* for details.

3.4 Save dictionary

VGram builders allow save dictionary to file. It's a good way to work with v-grams because dictionary is built for a long time.

Save dictionary by `save` method and load by static methods `load[ClassName]`.

3.4.1 Format

Dictionary saved as json-formatted file:

```
{
  "alphabet": [
    {
      "freq": 1188,
      "text": "fromthe",
      "vec": [
        0, 1, 2, 3, 15, 8, 6
      ]
    },
    ..
  ],
  "coder": [0, 1, 2, 3, 18, 12, ..],
  "size": 1000,
  "min_prob": 3.7657904299967802e-06,
  "fitted": true,
  "freqs_computed": true,
}
```

Field `text` is not nessesary and provided only when you work with tests. It contains the text of v-gram in `alphabet` items. In `Int`-versions the text field will not be in the file.

After v-grams construction, you can analyze the resulting dictionary.

`alphabet` is a list of v-gram objects
`freq` is a frequency of v-gram occurrence in data.
`vec` is a vector of language alphabet symbols for v-gram presentation.
`coder` is a sequence of symbols as they occur in the data.
`size` is a size of a dictionary
`min_prob`, `fitted` and `freqs_computed` are inner model information.
`fitted` and `freqs_computed` provided only for `(Int)VGram` class.

3.5 Tokenizers

VGramBuilder accepts a 2-d array of integers, but one of the most frequent usages is text analysis. We should use tokenizers to encode data to integer arrays.

3.5.1 CharTokenizer

For experiments, we use this tokenizer. `CharTokenizer` normalize text in the following way: all non-alphanumeric symbols were removed, then the text was converted to lower case. After that text split on single chars.

In Python, it would look like this:

```
class CharTokenizer:
    def __init__(self): ..
    def fit(self, X): ..
    def transform(self, X): ..
    def decode(self, X): ..
```

`CharTokenizer` implements sklearn fit-transform interface.

`fit(X)` consume a list of strings. Other arguments will be ignored.

`transform(X)` consume a list of strings and return a 2-d list of integers. Other arguments will be ignored.

`fit_transform(X)` consume a list of strings and return a 2-d list of integers. Other arguments will be ignored. Combine fit and transform methods.

`decode(X)` consume a 2-d list of integers and return list of strings.

This is a primary example of `CharTokenizer` usage.

```
data = ["hello world", "other text", "blablabla"]
tokenizer = CharTokenizer()
transformed_data = tokenizer.fit(data).transform(data)
print(transformed_data)
```

This tokenizer gives good results in experiments but for other tasks, different tokenizer may be more useful.

3.5.2 BaseTokenizer

You can make your own tokenizer by inheritance from `BaseTokenizer`. You should only define normalize and tokenize methods for one string.

```
def BaseTokenizer():
    def __init__(self): ..
    def fit(self, X): ..
    def transform(self, X): ..
    def decode(self, X): ..
    def normalize(self, string): ..
    def tokenize(self, string): ..
```

`normalize(string)` consume string and return a normalized string. If not redefined, the same string will be returned.

`tokenize(string)` consume string and return a list of integers. If not redefined, list of characters will be returned.

Example of WordTokenizer implementation

```
class WordTokenizer(BaseTokenizer):
    def normalize(self, X):
        return [re.sub("[^ \w\d]", "", re.sub(" +", " ", x)).lower() for x in X]

    def tokenize(self, X):
        return [x.split(" ") for x in X]
```

`BaseTokenizer` implements sklearn fit-transform interface.

`fit(X)` consume a list of strings. Other arguments will be ignored.

`transform(X)` consume a list of strings and return a 2-d list of integers. Other arguments will be ignored.

`fit_transform(X)` consume a list of strings and return a 2-d list of integers. Other arguments will be ignored. Combine fit and transform methods.

`decode(X)` consume a 2-d list of integers and return list of strings.

3.6 Examples

3.6.1 Basic example

This function accept list of strings, fit v-gram dictionary and transform to v-gram representation.

```
from vgram import VGram

def split_text_on_vgram(tests):
    vgram = VGram(10000, 10)
    vgram.fit(texts)
    return vgram.transform(texts)
```

3.6.2 Save dictionary

Save fitted dictionary to file and restore them

```
from vgram import VGram, loadVGram

vgram = VGram(5000, 10)
vgram.fit(texts)
```

(continues on next page)

(continued from previous page)

```
vgram.save("my_dict.json")
vgram = loadVGram("my_dict.json")
```

3.6.3 Include in scikit-learn pipeline

VGram can be embedded to sklearn pipeline as text preprocessing.

```
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer\
from sklearn.linear_model import SGDClassifier
from sklearn.pipeline import Pipeline
from vgram import VGram, loadVGram

def classification_on_vgrams(texts, labels):
    pipeline = Pipeline([
        ("vgram", VGram(10000, 10)),
        ("vect", CountVectorizer())
        ('tfidf', TfidfTransformer()),
        ('clf', SGDClassifier())
    ])
    pipeline.fit(texts, labels)
    print("train accuracy: ", np.mean(pipeline.predict(texts) == labels))
```

3.6.4 Real example

The primary example of 20 newsgroups dataset classification

```
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer, TfidfTransformer
from sklearn.linear_model import SGDClassifier
from sklearn.datasets import fetch_20newsgroups
from sklearn.pipeline import Pipeline
from vgram import VGram

# fetch data
train, test = fetch_20newsgroups(subset='train'), fetch_20newsgroups(subset='test')
data = train.data + test.data

# make vgram pipeline and fit it
vgram = VGram(10000, 10)
# it's ok, vgb fit only once
vgram.fit(data)

# fit classifier and get score
pipeline = Pipeline([
        ("features", vgram),
        ("vect", CountVectorizer())
        ('tfidf', TfidfTransformer(sublinear_tf=True)),
        ('clf', SGDClassifier(loss='hinge', penalty='l2', alpha=1e-4, max_iter=1002))
])
pipeline.fit(train.data, train.target)
print("train accuracy: ", np.mean(pipeline.predict(train.data) == train.target))
print("test accuracy: ", np.mean(pipeline.predict(test.data) == test.target))
```

(continues on next page)

(continued from previous page)

```
# show first ten elements of constructed vgram dictionary
alpha = vgram.alphabet()
print("First 10 alphabet elements:", alpha[:10])
```

V-Gram is an unsupervised method that's why we fit v-gram to all data. Once fitted, v-gram doesn't fit again, and we could not trouble about double fitting.

In the last two lines shown how to get dictionary alphabet and print some elements.

3.6.5 Words and v-grams union

We just make the union of features.

```
from sklearn.pipeline import Pipeline, FeatureUnion

vgram = Pipeline([
        ("vgb", VGram(10000, 10)),
        ("vect", CountVectorizer())
])
vgram.fit(data)

pipeline = Pipeline([
        ("features", FeatureUnion([("vgb", vgram), ("words", CountVectorizer())])),
        ('tfidf', TfidfTransformer(sublinear_tf=True)),
        ('clf', SGDClassifier(loss='hinge', penalty='l2', alpha=1e-4, max_iter=100))
])
```

It's easy to improve your existing project by adding v-grams.

3.6.6 Build v-grams on int sequences

IntVGram helps you to work with int sequences. Standard fit-transform return list of strings for usage in sklearn text pipeline (see *IntVGram in text pipeline*). Apply *split()* to make integer lists.

```
    from vgram import IntVGram

int_seqs = [[1, 1, 1, 2], [1, 2, 1, 2]] # 2-d int list or numpy array
    vgram = IntVGram(3, 1000)
    vgram.fit(int_seqs)
    text_vgrams = vgram.transform(int_seqs) # strings like "0 1 0 1"
    int_vgrams = [s.split() for s in text_vgrams] # [[0, 0, 0, 1], [0, 1, 0, 1]]
```

3.6.7 IntVGram in text pipeline

This example is equivalent to usage of VGram but uses integer streams instead strings.

```
from sklearn.pipeline import Pipeline, FeatureUnion
from vgram import IntVGram

pipeline = Pipeline([
        ("vgram", IntVGram(10000, 10)),
        ("vect", CountVectorizer()),
```

(continues on next page)

(continued from previous page)

```
        ('tfidf', TfidfTransformer(sublinear_tf=True)),
        ('clf', SGDClassifier(loss='hinge', penalty='l2', alpha=1e-4, max_iter=100))
])
pipeline.fit(int_seqs, labels)
```

3.6.8 Save and load v-grams

Yo can fit any v-gram dictionary and save to file. Then it can be restored by load methods.

```
from vgram import VGram, loadVGram

vgram = VGram(5000, 10)
vgram.fit(data)
vgram.save("my_dict.json")
vgram2 = loadVGram("my_dict.json")
vgram2.transform(data)
alphabet = vgram2.alphabet()
```

3.6.9 Construct VGram from file

```
vgram = loadVGram("/path/to/file")
vgram_pipeline = Pipeline([
    ("vgb", vgram),
    ("vect", CountVectorizer())
])
vgram_pipeline.fit(data)
```

Note: VGram fit only once and wouldn't be fitted again. Only CountVectorizer will be fitted.

3.6.10 Saving intermediate dictionaries to file

```
vgram = Pipeline([
    ("tokenizer", CharTokenizer()),
    ("vgb", VGramBuilder(10000, 10, "/path/to/file")),
    ("vect", CountVectorizer())
])
vgram.fit(data)
```

3.6.11 StreamVGram

```
from vgram import StreamVGram

vgram = StreamVGram(5000)
for seq in seqs:  # some stream of sequences, maybe infinite
    vgram.accept(seq)
vgram.update()  # don't forget it!
parsed_seq = vgram.parse(seq)
```

3.6.12 Load StreamVGram from file

Let's read an existing dictionary from the file, fit it more and save. If you have little data, you can train a dictionary on a large dataset (e.g., all wikipedia articles) and save it. Then fit more on domain-specific data for your task and get a better result than if you fit only on this data.

```
import random
from vgram import loadStreamVGram

vgram = loadStreamVGram("common_dict.json")
n_times = 10
for iters in range(n_times):  # feed data to the model few times until convergence
    for i in range(len(little_data)):
        vgram.accept(little_data[random.randint(0, len(little_data) - 1])
vgram.update()
parsed_seq = vgram.parse(seq)
vgram.save("task_specific_dict.json")
```

3.6.13 Fine-tune StreamVGram

You can pre-train VGram, save it, load as StreamVGram and fine-tune. Unfortunately, to do the opposite will not work.

```
import random
from vgram import VGram, StreamVGram, loadStreamVGram

vgram = VGram(5000, 10)
vgram.fit(data)
vgram.save("dict.json")
stream_vgram = loadStreamVGram("dict.json")
n_times = 10
for iters in range(n_times):  # feed data to the model few times until convergence
    for i in range(len(little_data)):
        stream_vgram.accept(little_data[random.randint(0, len(little_data) - 1])
stream_vgram.update()
parsed_seq = vgram.parse(seq)
stream_vgram.save("task_specific_dict.json")
```

3.6.14 Our experiments

You can find our experiments there.

CHAPTER 4

Contribute

- Source Code
- Issue Tracker

CHAPTER 5

License

The project is licensed under the MIT license.