Sally Version 1.0.0
— User Manual —

Konrad Rieck, Christian Wressnegger, and Alexander Bikadorov

May 8, 2019

Contents

1 NAME

sally - A tool for embedding strings in vector spaces

2 SYNOPSIS

sally [**options**] [**-c** *config*] *input output*

3 DESCRIPTION

sally is a small tool for mapping a set of strings to a set of vectors. This mapping is referred to as embedding and allows for applying techniques of machine learning and data mining for analysis of string data. **sally** can be applied to several types of string data, such as text documents, DNA sequences or log files, where it can handle common formats such as directories, archives and text files of string data.

The embedding of strings is carried out incrementally, where **sally** first loads a chunk of strings from *input*, computes the mapping to vectors and then writes the chunk of vectors to *output*. The configuration of this process, such as the input format, the embedding setting and the output format, are specified in the file *config* and additionally using command-line options.

```
.---------.                        .----------.
|   dir   |                        |   text   |
|   arc   |   \   .---------.   /  |  libsvm  |
|  lines  |   --  |  Sally  |  --  |  matlab  |
|   ...   |   /   '---------'   \  |   ...    |
'---------'                        '----------'
  Input           Embedding          Output
```

sally implements a standard technique for mapping strings to a vector space that can be referred to as generalized bag-of-words model. The strings are characterized by a set of features, where each feature is associated with one dimension of the vector space. The following types of features are supported by **sally**: bytes, tokens, n-grams of bytes and n-grams of tokens.

sally proceeds by counting the occurrences of the specified features in each string and generating a sparse vector of count values. Alternatively, binary or TF-IDF values can be computed and stored in the vectors. **sally** then normalizes the vector, for example using the L1 or L2 norm, and outputs it in a specified format, such as plain text, LibSVM, Cluto or Matlab format.

The processing chain of **sally** makes use of sparse data structures, such that high-dimensional vectors with millions of dimensions can be handled efficiently. All features are indexed by a hash function, where the bit width of this function can be adjusted according to the characteristics of the data. In several settings, a hash width between 22 and 26 bit already suffices to store millions of features with little loss of accuracy.

4 CONFIGURATION

The configuration of **sally** is specified in a configuration file. This file is structured into the three sections **input**, **features** and **output**, which define the parameters of the input format, the feature extraction and the output format, respectively.

All parameters of the configuration can be also be specified on the command-line. That is, if a parameter is specified in the configuration as **xx = "yy"**, it can be alternatively supplied as a command-line option by **--xx "yy"**.

If no configuration file is provided, **sally** resorts to a default configuration. This default configuration can be dumped using the command-line option **-D**.

Input formats

sally supports different formats for reading data sets of strings, which may range from plain files to directories and other structured resources. Following is a list of supported input formats.

input = {

input_format = "lines";

This parameter specifies the input format.

"dir"

The input strings are available as binary files in a directory and the name of the directory is given as *input* to **sally**. The suffixes of the files are used as labels for the extracted vectors.

"arc"

The input strings are available as binary files in a compressed archive, such as a zip or tgz archive. The name of the archive is given as *input* to **sally**. The suffixes of the files are used as labels for the extracted vectors.

"lines"

The input strings are available as lines in a text file. The name of the file is given as *input* to **sally**. The lines need to be separated by newline and may not contain the NUL character. Labels can be extracted from each line using a regular expression (see **lines_regex**).

"fasta"

The input strings are available in FASTA format. The name of the file is given as *input* to **sally**. Labels can be extracted from the description of each sequence using a regular expression (see **fasta_regex**). Comments are allowed if they are preceded by either ';' or '>'.

"stdin"

The input strings are read form standard input (stdin) as text lines. The parameter *input* is ignored. The lines need to be separated by newline and may not contain the NUL character. Labels can be extracted from each line using a regular expression (see **lines_regex**).

chunk_size = 256;

To enable an efficient processing of large data sets, **sally** processes strings in chunks. This parameter defines the number of strings in one of these chunks. Depending on the lengths of the strings, this parameter can be adjusted to balance loading and processing of data.

decode_str = false;

If this parameter is set to 1, **sally** automatically decodes strings that contain URI-encoded characters. That is, substrings of the form %XX are replaced with the byte represented by the hexadecimal number XX. This feature comes handy, if binary data is provided using the textual input format "lines". For example, HTTP requests can be stored in a single line if line-breaks are represented by "%0a%0d".

fasta_regex = " (\\+|-)?[0-9]+";

The FASTA format allows to equip each string with a short description. In several data sets this description contains a numerical label which can be useful in supervised learning tasks. The parameter is a regular expression which can be used to match numerical labels, such as +1 and -1.

lines_regex = "^(\\+|-)?[0-9]+";

If the strings are available as text lines, the parameter can be used to extract a numerical label from the start of the strings. The parameter is a regular expression matching labels, such as +1 and -1.

reverse_str = false;

If this parameter is set to 1, the bytes of all input strings will be reversed. Such reversing might help in situations where the reading direction of the input strings is unspecified.

stoptoken_file = "";

Stop tokens (irrelevant words) can be filtered from the strings by providing a file containing these tokens; one per line. Non-printable characters can be escaped using URI encoding (%XX). Stop tokens can only be filtered, if a set of delimiters is defined using **token_delim**.

};

Features

The strings loaded by **sally** are characterized by a set of features, where each feature is associated with one dimension of the vector space. Different types of features can be extracted by changing the following parameters.

features = {

ngram_len = 2;

granularity = ”bytes”;

The parameter **ngram_len** specifies the numbers of consecutive symbols that are considered as one feature, while the parameter **granularity** defines the granularity of these symbols. If the granularity is set to *bytes*, **sally** considers bytes as symbols, whereas if **granularity** is set to *tokens*, all strings (token) separated by a set of delimiters are considered as symbols. The following types of different feature types can be extracted using these parameters:

tokens

The strings are partitioned into substrings (tokens) using a set of delimiter characters. Such partitioning is typical for natural language processing, where the delimiters are usually defined as white-space and punctuation symbols. An embedding using tokens is selected by choosing *tokens* as granularity (**granularity**), defining a set of delimiter characters (**token_delim**) and setting the n-gram length to 1 (**ngram_len**).

byte n-grams

The strings are characterized by all possible byte sequences of a fixed length n (byte n-grams). These features are frequently used if no information about the structure of strings is available, such as in bioinformatics or computer security. An embedding using byte n-grams is selected by choosing *bytes* as granularity (**granularity**) and defining the n-gram length (**ngram_len**).

token n-grams

The strings are characterized by all possible token sequences of a fixed length n (token n-grams). These features require the definition of a set of delimiters and a length n. They are often used in natural language processing as a coarse way for capturing structure of text. An embedding using token n-grams is selected by choosing *tokens* as granularity (**granularity**), defining a set of delimiter characters (**token_delim**) and choosing an n-gram length (**ngram_len**).

token_delim = ” %0a%0d”;

The parameter **token_delim** defines characters for delimiting tokens in strings, for example *” %0a%0d”*. It is only considered, if the granularity is set to *tokens*, otherwise it is ignored.

ngram_pos = false;

pos_shift = 0;

The parameter **ngram_pos** can be used to enable positional n-grams. In contrast to regular n-grams, these substrings of length n are associated with a position in the originating string. Positional n-grams thus only match if they appear at the same location in a string. The additional parameter **pos_shift** can be used to add a shift to the n-grams. If the parameter is set to *k*, multiple positional n-grams are extracted with a shift from *-k* to *+k*.

ngram_blend = false;

The parameter **ngram_blend** can be used to enable blended n-grams. In this setting, n-grams starting from length 1 up to length **ngram_len** are extracted and used for creating a feature vector. That is, all n-gram lengths are blended in a joint feature vector, hence the name of this approach.

ngram_sort = false;

This parameter can be used to enable sorted n-grams (n-perms). During the extraction of an n-gram its symbols are sorted according to their lexographical order. That is, if n-grams are composed of bytes, these bytes are sorted. If the n-grams are composed of tokens, the respective tokens are sorted. Sorted n-grams can be used to compensate minor perturbation in string data. For example, the strings "abac" and "aabc" contain the same sorted 3-grams, namely "aab" and "abc".

vect_embed = "bin";

This parameter specifies how the features are embedded in the vector space. Supported values are "bin" for associating each dimension with a binary value, "cnt" for associating each dimension with a count value and "tfidf" for using a TF-IDF weighting.

vect_norm = "none";

The feature vectors extracted by **sally** can be normalized to a given vector norm. Supported norms are "l1" for the Taxicab norm (L1) and "l2" for the Euclidean norm (L2). Normalization is often useful, if the lengths of the strings varies significantly and thus vectors differ just due to the number of extracted features.

vect_sign = false;

sally uses a hash function to map individual features to dimensions in the vector space. Depending on the number of bits used by this functions, different features may collide on the same dimension. Such collisions often result in larger values at these dimensions. The parameter **vect_sign** can be used to enable a signed embedding. That is, one bit of the hash function is used to define a sign for each feature. Collisions are still possible, yet their impact is lessened as colliding features not necessary induce larger values.

thres_low = 0;

This parameter defines a minimum threshold for the entries in each vector. Values below the threshold are removed. The parameter can be used to filter low-valued features from the vectors, such as very infrequent tokens. If the parameter is set to 0, the thresholding is disabled.

thres_high = 0;

This parameter defines a maximum threshold for the entries in each vector. Values above the threshold are removed. The parameter can be used to filter high-valued features from the vectors, such as very frequent tokens. If the parameter is set to 0, the thresholding is disabled.

hash_bits = 22;

sally uses a hash function to map individual features to dimensions. This parameter specifies the number of bits used by this hash function and defines the maximum dimensionality of the vector space with 2 ** bits.

As the embedding of **sally** is usually sparse and only a small fraction of dimensions is non-zero, 22 to 26 bits are often sufficient for representing the data. If this parameter is chosen too small, the embedding may suffer from collisions of features in the vector space. If it is chosen too large, several application may choke from the vast amount of dimensions.

explicit_hash = false;

For performance reasons **sally** maps features to dimensions without memorizing the features associated with these dimensions. For certain analysis tasks, however, it may be necessary to retrieve a full mapping including the features of each dimension. If this parameter is enabled **sally** keeps track of all features and depending on the selected output format stores them with the extracted feature vectors. The mapping may suffer from collisions due to hashing. You can control the size of the hash table using the parameter **hash_bits**.

hash_file = "";

This parameter enables saving the mapping between features and dimensions to a gzip-compressed file. The functionality is similar to **explicit_hash**, except that **sally** does not store the features with the feature vectors but separately in a file. Note that the tracking of features and dimensions induces a considerable performance overhead. Also note that mapping may contain collisions, where simply the latest colliding entry overrides previous entries. You can control the size of the hash table using the parameter **hash_bits**.

tfidf_file = "tfidf.fv";

This parameter specifies a file to store the TF-IDF weights. If the embedding **tfidf** is selected, **sally** first checks if the given file is present. If it is not available, TF-IDF weights will be computed from the input data and stored to this file, otherwise the weights will be read from the file. Keeping a separate file for TF-IDF weights allows for computing the weighting for a data set, say the training set, and applying the exact same weighting to further data sets.

};

Filtering and dimension reduction

sally supports some simple methods for unsupervised filtering and dimension reduction. These can be applied to reduce the number of dimensions in the feature vectors. Due to their simplicitiy, however, these methods only capture a small amount of information from the original feature vectors and should be used with caution.

filter = {

dim_reduce = "none";

Following is a list of methods for unsupervised filtering and dimension reduction supported by **sally**:

"none"

No dimension reduction is performed.

"simhash"

Each feature vector is reduced to a similarity hash with **dim_num** bits. The string features associated with each dimension are hashed and aggregated to a single hash value as proposed by Charikar (STOC 2002). For convenience, the computed hash value is again represented as a feature vector. Note that the number of maximum bits is defined by **hash_bits** in the feature configuration section.

"minhash"

Each feature vector is reduced to a "minimum hash" with **dim_num** bits. The string features associated with each dimension are hashed and sorted multiple times as proposed by Broder (1997). In each round the smallest hash value is appended to the minimum hash. The number of hash bits in each round is defined by **hash_bits** in the feature configuration section and the number of rounds is simply determined by ceil(**dim_num** / **hash_bits**).

"bloom"

Each feature vector is reduced to a small Boom filter with **dim_num** bits. The string features associated with each dimension are hashed using **bloom_num** hash functions. For each string feature and each hash function one bit in the filter is set. As a result, the filter is populated with bits similar to a real Bloom filter. To avoid saturating the filter, **dim_num** should be significantly larger than **bloom_num** if several string features are associated with one feature vector. As a rule of thumb you can set **bloom_num** = 0.7 * **dim_num** / *vec_len*, where *vec_len* is the expected number of string features.

dim_num = 32;

This parameter defines the number of dimensions/bits to generate using unsupervised dimension reduction.

bloom_num = 2;

This parameters specifies how many hash functions are used to map a feature vector and its string features to a Bloom filter. To avoid saturating the filter, you should choose **bloom_num** significantly smaller than the size of the Bloom filter **dim_num**.

};

Output formats

Once the input strings have been embedded in a vector space, **sally** stores the resulting vectors in one of several common formats, which allows for applying typical tools of statistics and machine learning to the data, for example, Matlab, Octave, Shogun, Weka, SVMLight and LibSVM.

output = {

output_format = "libsvm";

Following is a list of output formats supported by **sally**:

"libsvm"

The feature vectors of the embedded strings are stored in the common libsvm format, which is supported by Shogun, SVMLight and LibSVM. The name of the output file is given as *output* to **sally**.

"text"

The feature vectors of the embedded strings are stored as plain text. Each feature vector is represented as a list of dimensions, which is written to *output* in the following form

```
dimension:feature:value,... source
```

dimension specifies the index of the dimension, *feature* a textual representation of the feature and *value* the value at the dimension. If parameter **explicit_hash** is not enabled in the configuration, the field *feature* is empty.

"stdout"

The feature vectors of the embedded strings are written to standard output (stdout) as text. Each feature vector is represented as a list of dimensions in the following form:

```
dimension:feature:value,... source
```

dimension specifies the index of the dimension, *feature* a textual representation of the feature and *value* the value at the dimension. If parameter **explicit_hash** is not enabled in the configuration, the field *feature* is empty.

"matlab"

The feature vectors of the embedded strings are stored in Matlab format (v5). The vectors are stored as a 1 x n struct array with the fields: data, src, label and feat. The name of the output file is given as *output* to **sally**. Note that great care is required to efficiently operate with sparse vectors in Matlab. If the sparse representation is lost during computations, excessive run-time and memory requirements are likely.

"cluto"

The feature vectors of the embedded strings are stored as a sparse matrix suitable for the clustering tool Cluto. The first line of the file is a header for Cluto, while the remaining lines correspond to feature vectors. The name of the output file is given as *output* to **sally**. Note that Cluto can not handle arbitrarily large vector spaces and thus the **"hash_bits"** should be set to values below 24.

"json"

The feature vectors of the embedded strings are stored as JSON objects. Each object contains a list of dimension indices denoted *dim* and corresponding values denoted as *val*. Depending on the configuration the source for each object as well as the actual string feature associated with each dimension are also stored in the JSON object.

skip_null = false;

If this parameter is enabled, **sally** will not output null vectors, that is, vectors where all dimensions are zero. These vectors occur if a string does not contain a single string feature.

};

5 OPTIONS

The configuration of **sally** can be refined and altered using several command-line options. In particular, all parameters of the configuration can be specified on the command-line. That is, if a parameter is specified as **xx = ”yy”** in the configuration file, it can be changed by using the command-line option **--xx ”zz”**. Following is a list of common options:

I/O options

```
  -i,  --input_format <format>   Set input format for strings.
       --chunk_size <num>        Set chunk size for processing.
       --decode_str              Enable URI-decoding of strings.
       --fasta_regex <regex>     Set RE for labels in FASTA data.
       --lines_regex <regex>     Set RE for labels in text lines.
       --reverse_str             Reverse (flip) all strings.
       --stoptoken_file <file>   Provide a file with stop tokens.
  -o,  --output_format <format>  Set output format for vectors.
  -k,  --skip_null               Skip null vectors in output.
```

Feature options

```
  -n,  --ngram_len <num>         Set length of n-grams.
  -d,  --token_delim <delim>     Set delimiters of tokens.
  -g   --granularity <type>      Set granularity: bytes, tokens.
  -p,  --ngram_pos               Enable positional n-grams.
       --pos_shift <num>         Set shift of positional n-grams.
  -B,  --ngram_blend             Enable blended n-grams.
  -s,  --ngram_sort              Enable sorted n-grams (n-perms).
  -E,  --vect_embed <embed>      Set embedding mode for vectors.
  -N,  --vect_norm <norm>        Set normalization mode for vectors.
  -S,  --vect_sign               Enabled signed embedding.
       --thres_low <float>       Enable minimum threshold for vectors.
       --thres_high <float>      Enable maximum threshold for vectors.
  -b,  --hash_bits <num>         Set number of hash bits.
  -X,  --explicit_hash           Enable explicit hash table.
       --hash_file <file>        Set file name for explicit hash table.
       --tfidf_file <file>       Set file name for TFIDF weighting.
```

Generic options

```
  -c,  --config_file <file>      Set configuration file.
  -v,  --verbose                 Increase verbosity.
  -q,  --quiet                   Be quiet during processing.
  -C,  --print_config            Print the current configuration.
  -D,  --print_defaults          Print the default configuration.
  -V,  --version                 Print version and copyright.
  -h,  --help                    Print this help screen.
```

6 FILES

PREFIX/share/doc/sally/example.cfg

An example configuration file for **sally**. See the configuration section for further details.

7 COPYRIGHT