Text Representation Techniques

Natural Language Processing

Daniel Ortiz Martínez, David Buchaca Prats

Basic Definitions

- **Corpus**: a set of documents
- **Vocabulary**: a vocabulary is a set of words
- **Corpus vocabulary**: a set of words containing the atomic symbols used to represent the corpus
- **Token**: a substring from a document with *atomic* meaning (usually tokens refer to words)
- Vocabularies are usually extracted tokenizing a corpus

Basic Text Representation

- In order to input text to a machine learning algorithm we need to convert the string representation to vectors
- The most basic way to encode text is a bag of words representation
  - A bag-of-words describes the occurrence of words within a text.
  - A bag of words representation involves:
    - A vocabulary of known words
    - A measure of the presence of known words (such as word counts)

Basic Text Representation

- The vocabulary is usually stored as a dictionary (`Dict` or `OrderedDict`) that we will call `word_to_pos`
- Keys in `word_to_pos` are the words in the vocabulary
- Values in `word_to_pos` are the positions assigned to the words
- The bag of words feature vector x for a document d is constructed using the counts of the words in `d`
- Coordinate `k` in `x` contains the number of times the `k`'th word from `word_to_pos` appears in `d`

Basic Text Representation

```
word_to_pos = {'the':0, 'man':1, 'that':2, 'went':3, 'to':4, 'moon':5}
```

"The man that went to the moon" $\rightarrow$ `x = [1, 1, 1, 1, 1, 1]`

Examples Bag of Words Representation

- Consider the corpus:

```
The cat sat on the mat
the cat and the dog sat on the mat
```

- Resulting in the following vocabulary:

```
word_to_pos = {'the':0, 'cat':1, 'sat':2, 'on':3, 'the':4, 'mat':5, 'and':6, 'dog':7}
```

- Bag of words representations:

"The cat that sat on the mat"	`[0, 1, 1, 1, 2, 1, 0, 0]`
"the cat and the dog sat on the mat"	`[0, 1, 1, 1, 3, 1, 1, 1]`

Dictionaries for Bag of Words Representation: Standard Dict

```
normal_dict = {}
normal_dict['1'] = "A"
normal_dict['2'] = "B"
normal_dict['3'] = "C"
normal_dict['4'] = "D"
normal_dict['5'] = "E"

print("Printing normal dictionary:")
for k,v, in normal_dict.items():
    print("key : {0}, value: {1}".format(k,v))

Printing normal dictionary:
key : 3, value: C
key : 2, value: B
key : 1, value: A
key : 4, value: D
key : 5, value: E
```

Dictionaries for Bag of Words Representation: Ordered Dict

```
import collections

ordered_dict = collections.OrderedDict()
ordered_dict['1'] = "A"
ordered_dict['2'] = "B"
ordered_dict['3'] = "C"
ordered_dict['4'] = "D"
ordered_dict['5'] = "E"

print("Printing ordered dictionary:")
for k,v, in ordered_dict.items():
    print("key : {0}, value: {1}".format(k,v))

Printing normal dictionary:
key : 1, value: A
key : 2, value: B
key : 3, value: C
key : 4, value: D
key : 5, value: E
```

Text Representation

- How can we apply machine learning techniques when the input is a text description?
  - We need to transform strings to vectors
- Challenges when working with text:
  - The feature vector dimensionality can be huge
  - Words outside the vocabulary (such as misspelled words) might bring problems

Creating Feature Vectors

- Given a corpus, how do we define a vocabulary?
  - We need to iterate over words, but raw data is not provided with words
- There are several decisions that impact vocabulary creation:
  - How do we generate tokens?
  - Do we need to clean tokens?
  - Do we create combinations of tokens?
  - Do we select combinations of tokens?

Constructing a Vocabulary

- In order to build feature descriptors we need to create a `word_to_pos`
- `word_to_pos` depends on
  - How do we generate tokens from strings
  - How do we clean the tokens
- Many packages contain classes to create vectorizers with a `fit` method
- scikit-learn has the `CountVectorizer` class during fitting
  - Receives as input an iterable of strings
  - For each string the class finds the tokens (or words) in the string
  - Words are stored in `word_to_pos`

`CountVectorizer` from scikit-learn

- `.fit(X)` learns a vocabulary from raw data `X`
- `.transform(x)` generates an array with the feature descriptor for `x`
- How should we store `.transform(x)`?
  - Numpy array
  - List
  - Pandas dataframe
  - **Scipy Compressed Sparse Row (CSR) matrix**

High Dimensional Feature Vectors

- In the context of document descriptors feature vectors can contain millions of words
- Feature vectors are often sparse
- Storing such vectors using lists of Numpy arrays is very inefficient

Compressed Sparse Row Matrix (CSR Matrix)

- Given an $m \times n$ matrix, a CSR matrix is constructed from 3 arrays:
  - `data`: contains the non-zero values of the matrix
  - `ind_col`: contains the column indices of the elements in the matrix
  - `ind_ptr`: contains $m + 1$ pointers, the m'th pointer stores the position of the first value in the `data` array that belongs to the m'th row. The last element is equal to the length of the `data` array

```
X = np.array([[0, 5, 7, 6, 0, 0, 0, 0, 0, 0],
              [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
              [7, 0, 4, 9, 0, 0, 0, 0, 0, 0]])

data    = [5, 7, 6, 1, 7, 4, 9]
ind_col = [1, 2, 3, 9, 0, 2, 3]
ind_ptr = [0, 3, 4, 7]

X_csr = sp.csr_matrix((data, ind_col, ind_ptr))
X_csr.toarray()

array([[0, 5, 7, 6, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0, 0, 0, 1],
       [7, 0, 4, 9, 0, 0, 0, 0, 0, 0]])
```

Transforming an Iterable of Strings to a Sparse Matrix

- How can we transform a sequence of strings to a sparse matrix?
- One approach would be:
    1. Iterate over the data to create a vocabulary
    2. Iterate over the data to generate the feature vectors for the elements in the vocabulary
- Can we improve this?

Generating a Feature Descriptor for a List of Documents

```
docs = [['hello', 'world', 'hello'],['goodbye', 'cruel', 'teacher' ]]

def prepare_word_counts_with_dict(docs, verbose=False):
    ind_ptr = [0]
    ind_col = []
    data = []
    vocabulary = {}

    for m, doc in enumerate(docs):
        # TO-BE-DONE: Use an auxiliary dictionary to keep track of counts
        #
        #
        #
        #
        #
        #
        #
        #
    return (data, ind_col, ind_ptr)

sp.csr_matrix(prepare_word_counts_with_dict(docs)).toarray()

array([[2, 1, 0, 0, 0],
       [0, 0, 1, 1, 1]])
```

Generating a Feature Descriptor for a List of Documents

```
docs = [['hello', 'world', 'hello'],['goodbye', 'cruel', 'teacher' ]]

def prepare_word_counts_with_dict(docs, verbose=False):
    ind_ptr = [0]
    ind_col = []
    data = []
    vocabulary = {}

    for m, doc in enumerate(docs):
        word_ind_counter = collections.defaultdict(int)
        for word in doc:
            vocabulary.setdefault(word, len(vocabulary))
            word_ind_counter[word] += 1

        data.extend(word_ind_counter.values())
        ind_ptr.append(ind_ptr[-1] + len(word_ind_counter))
        ind_col.extend([vocabulary[w] for w in word_ind_counter.keys()])

    return (data, ind_col, ind_ptr)

sp.csr_matrix(prepare_word_counts_with_dict(docs)).toarray()

array([[2, 1, 0, 0, 0],
       [0, 0, 1, 1, 1]])
```

Generating a Feature Descriptor for a List of Documents

```
docs = [['hello', 'world', 'hello'],['goodbye', 'cruel', 'teacher' ]]

ind_ptr = [0]
ind_col = []
data = []
vocabulary = {}

for d in docs:
    # TO-BE-DONE: Implementation WITHOUT auxiliary dictionary
    #
    #
    #
    #

sp.csr_matrix((data, ind_col, ind_ptr)).toarray()

array([[2, 1, 0, 0, 0],
       [0, 0, 1, 1, 1]])
```

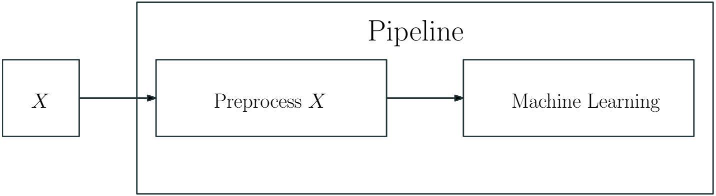
Generating a Feature Descriptor for a List of Documents

```
docs = [['hello', 'world', 'hello'],['goodbye', 'cruel', 'teacher' ]]

ind_ptr = [0]
ind_col = []
data = []
vocabulary = {}

for d in docs:
    for term in d:
        index = vocabulary.setdefault(term, len(vocabulary))
        ind_col.append(index)
        data.append(1)
    ind_ptr.append(len(ind_col))

sp.csr_matrix((data, ind_col, ind_ptr)).toarray()

array([[2, 1, 0, 0, 0],
       [0, 0, 1, 1, 1]])
```

Different Ways to Cast a Learning Task

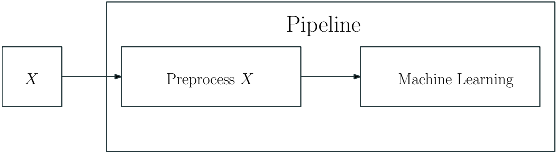
- Typical ML pipeline

- Composable/composite models



Examples


```
count_vectorizer = sklearn.feature_extraction.text.CountVectorizer()
logistic = sklearn.linear_model.LogisticRegression(C=0.1)

X_train_feature_vec = count_vectorizer.fit_transform(X_train)
logistic.fit(X_train_feature_vec, y_train)
```

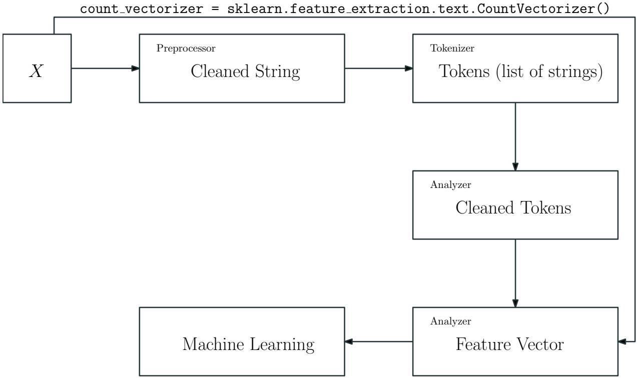



```
count_vectorizer = sklearn.feature_extraction.text.CountVectorizer()
logistic = sklearn.linear_model.LogisticRegression(C=0.1)

model_pipe = sklearn.pipeline.Pipeline([("countvectorizer", count_vectorizer),
                                        ("logisticregression", logistic)])

model_pipe.fit(X_train, y_train)
```

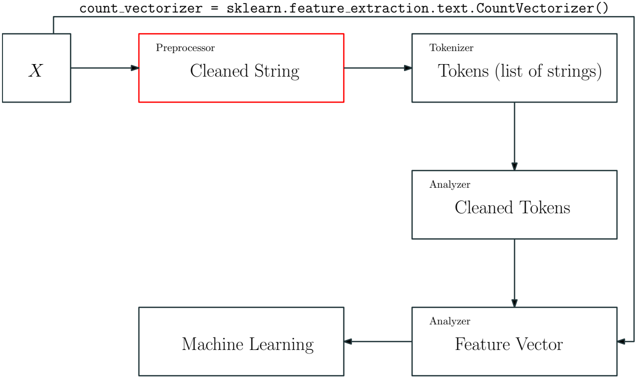
Pipelines or Composite Models

- The purpose of the pipeline is to assemble a composite model which consists on several steps that can be cross-validated together
- Pipelines allow practitioners to easily compose and validate decisions made during training, instead of relying on pre-processing steps with *hand-crafted* decisions
- Some of this decisions might include:
    - Type of tokenizer
    - Removing stop words
    - Using only words or bigram, trigrams etc..
    - Regularization parameters

- A `CountVectorizer` is one of the most straight forward methods to generate descriptors for documents

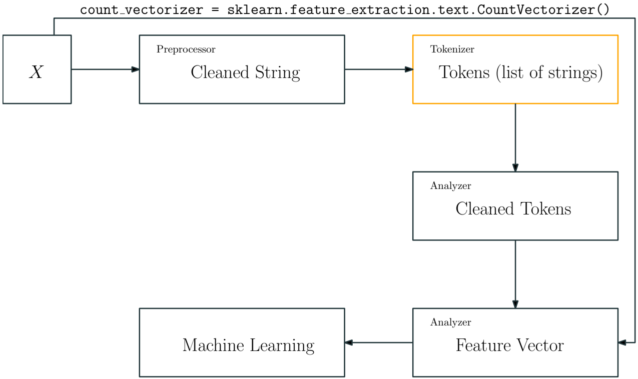


CountVectorizer: Preprocessor




```
count_vectorizer = sklearn.feature_extraction.text.CountVectorizer()
count_vectorizer.fit(X_train)
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
    dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
    lowercase=True, max_df=1.0, max_features=None, min_df=1,
    ngram_range=(1, 1), preprocessor=None, stop_words=None,
    strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
    tokenizer=None, vocabulary=None)
```

CountVectorizer: Tokenizer




```
count_vectorizer = sklearn.feature_extraction.text.CountVectorizer()
count_vectorizer.fit(X_train)
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
    dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
    lowercase=True, max_df=1.0, max_features=None, min_df=1,
    ngram_range=(1, 1), preprocessor=None, stop_words=None,
    strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
    tokenizer=None, vocabulary=None)
```

Tokenizer limitations

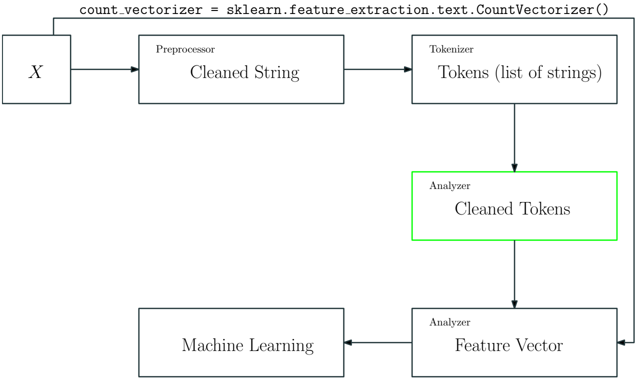
- Single letter words such as “I” are ignored

```
re.findall(r'(?u)\b\w\w+\b', "I can't wait to go there!")
['can', 'wait', 'to', 'go', 'there']
```

- Expressions such as “can’t” are modified and might even change meaning

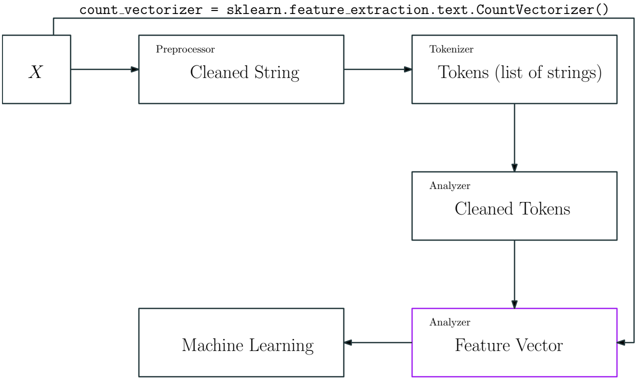
```
re.findall(r"\w+\'\w+", "I can't wait but I won't go there")
["can't", "won't"]
```

CountVectorizer: Analyzer




```
count_vectorizer = sklearn.feature_extraction.text.CountVectorizer()
count_vectorizer.fit(X_train)
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
    dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
    lowercase=True, max_df=1.0, max_features=None, min_df=1,
    ngram_range=(1, 1), preprocessor=None, stop_words=None,
    strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
    tokenizer=None, vocabulary=None)
```

`CountVectorizer`: Analyzer




```
count_vectorizer = sklearn.feature_extraction.text.CountVectorizer()
count_vectorizer.fit(X_train)
CountVectorizer(analyzer='word', binary=False, decode_error='strict',
    dtype=<class 'numpy.int64'>, encoding='utf-8', input='content',
    lowercase=True, max_df=1.0, max_features=None, min_df=1,
    ngram_range=(1, 1), preprocessor=None, stop_words=None,
    strip_accents=None, token_pattern='(?u)\\b\\w\\w+\\b',
    tokenizer=None, vocabulary=None)
```

Summary: Customizing Vectorizer Classes

- **Preprocessor**: a callable that takes an entire document as input (as a single string), and returns a possibly transformed version of the document, still as an entire string. This can be used to remove HTML tags, lowercase the entire document, etc.
- **Tokenizer**: a callable that takes the output from the preprocessor and splits it into tokens, then returns a list of these
- **Analyzer**: a callable that replaces the preprocessor and tokenizer. The default analyzers all call the preprocessor and tokenizer, but custom analyzers will skip this. N-gram extraction and stop word filtering take place at the analyzer level, so a custom analyzer may have to reproduce these steps

Why Is Important to Tune Vectorizers

- Many times vocabulary can be so rare that is not worth storing it
- One good example would be numbers, alphanumeric codes, etc.
- These words can be categorized, using special symbols to represent them

Word Transformations: Stemming

- Stemming consist on removing the suffixes or prefixes used in word
- The returned string from a stemmer might not be a valid word from the language
- Example:

```
Stem(saw) = saw
Stem(destabilize) = destabil
```

Word Transformations: Lemmatization

- Lemmatization consist on properly use of a vocabulary and morphological analysis of words, aiming to remove inflectional endings only with the goal of returning any word to a set of base (or dictionary form) words
- The returned string from a lemmatizer should be a valid word from the language
- Example:

```
Lemmatize(saw) = see
Lemmatize(destabilize) = destabilize
```