INTRODUCTION TO NLP

Session 2

David Buchaca Prats

2021

BASIC DEFINITIONS

- Definition: A corpus is a set of documents.
- Definition: A vocabulary is a set of words.
- Definition: A vocabulary extracted from a corpus is a set of words containing the atomic symbols used to represent the corpus.
- Vocabularies are usually extracted tokenizing a corpus. A token is a substring from a document with 'atomic' meaning (usually tokens refer to words).

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).

BAG OF WORDS 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`.
  - `word_to_pos = {‘the’:0, ‘man’:1, ‘that’:2, ‘went’:3, ‘to’:4, ‘moon’:5}`

the man that went to moon

- `“The man that went to the moon”`
- x = [2, 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”

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

Notice word_to_pos contains "the" and “The”.
Unless we do something to each token that is what will happen.

- Bag of words feature descriptor
  - “the cat 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]

```
Positions [ 0, 1, 2, 3, 4, 5, 6, 7 ]
```

BASIC DICTIONARIES FOR BAG OF WORDS REPRESENTATION

Standard dict (keys have no order)

```
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 : 1, value : A
key : 2, value : B
key : 4, value : D
key : 5, value : E
```

OrderedDic (keys have order)

```
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 Ordered 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:
  - I) How do we generate tokens?
  - II) Do we need to clean tokens?
  - III) Do we create combinations of tokens?
  - IV) 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 we generate tokens from strings.
  - How we clean the tokens.
- Many packages contain classes build to create vectorizer with a `.fit` method
- Scikit-learn has the `CountVectorizer` class during fit
  - 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 SKLEARN

- `.fit(X)` Learns a vocabulary from raw data `X`.
- `.transform(x)` Generates an array with the feature descriptor for `x`.
- How should be store `.transform(x)` ?
  - Numpy array
  - List
  - Pandas dataframe
  - **Scipy csr matrix**

HIGH DIMENSIONAL FEATURE VECTORS

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

COMPRESSED SPARSE ROW MATRIX (CSR MATRIX)

- A CSR matrix is constructed from 3 arrays:
  - `data`: contains the coordinate values of the array.
  - `ind_col`: contains the column indices of the elements in the matrix.
  - `ind_ptr`: contains the pointers that define which elements belong to to each row.

```
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]])
X
```

executed in 3ms, finished 12:25:55 2021-02-22

```
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]
```

executed in 2ms, finished 12:26:55 2021-02-22

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

executed in 3ms, finished 12:27:29 2021-02-22

```
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:
  - I) Iterate over the data to create a vocabulary
  - II) 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']]

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))
```

executed in 3ms, finished 12:34:02 2021-02-22

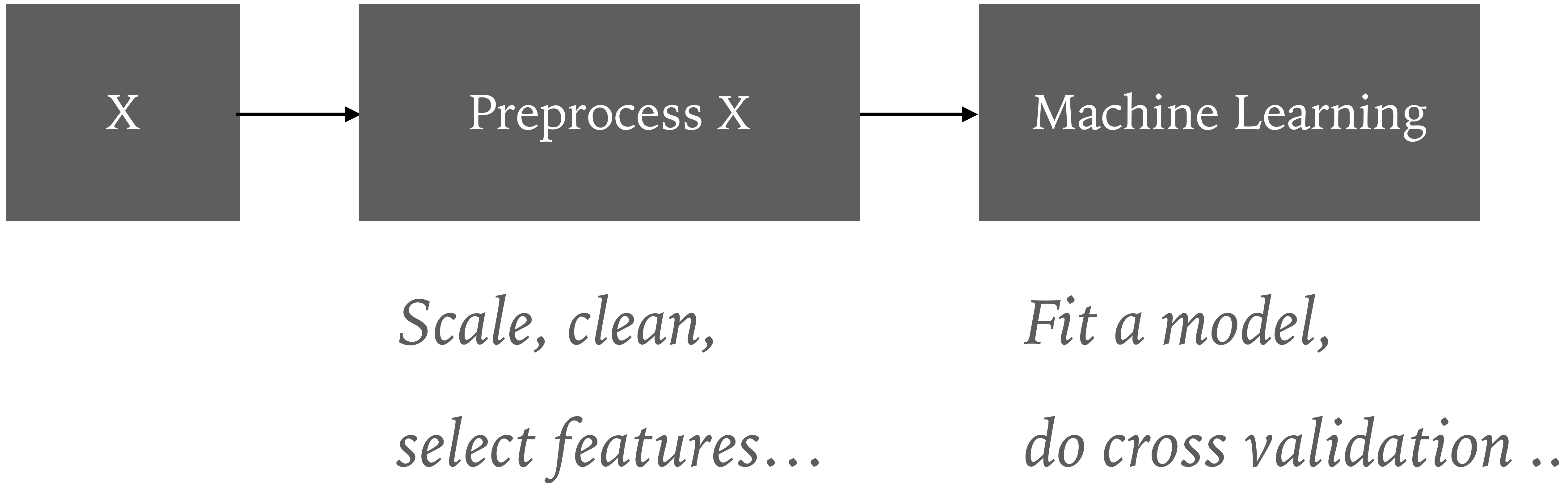
```
sp.csr_matrix((data, ind_col, ind_ptr)).toarray()
```

executed in 3ms, finished 12:34:07 2021-02-22

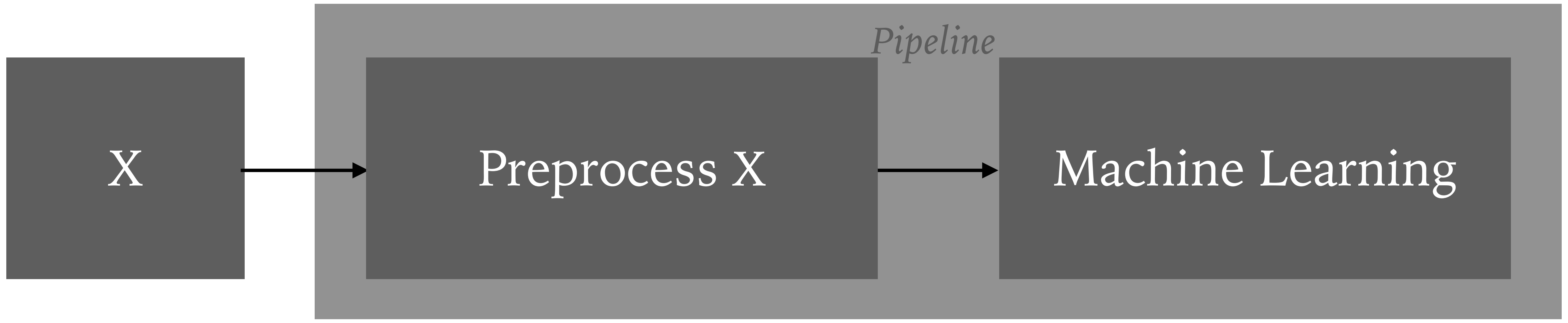
```
array([[2, 1, 0, 0, 0],
       [0, 0, 1, 1, 1]])
```

DIFFERENT WAYS TO CAST A LEARNING TASK

- Typical ML pipeline

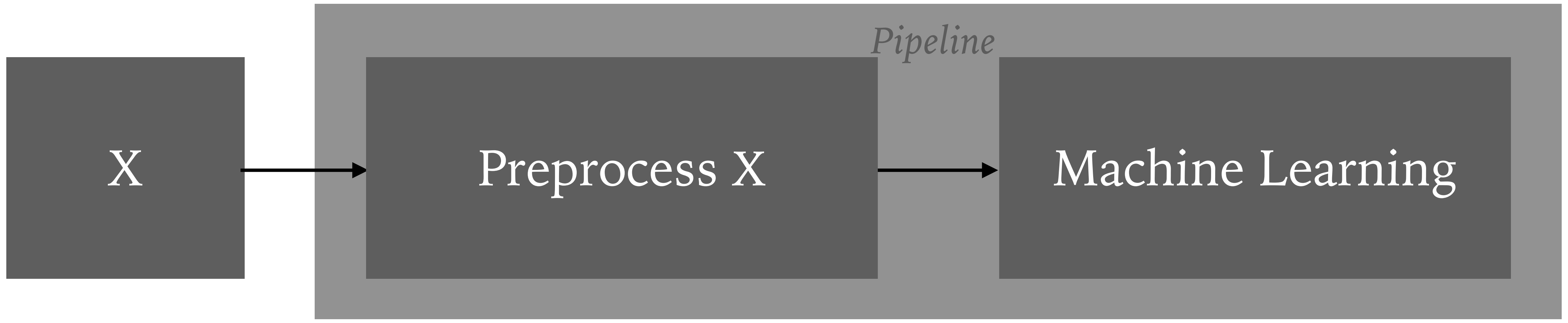


- Composable models



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)
```



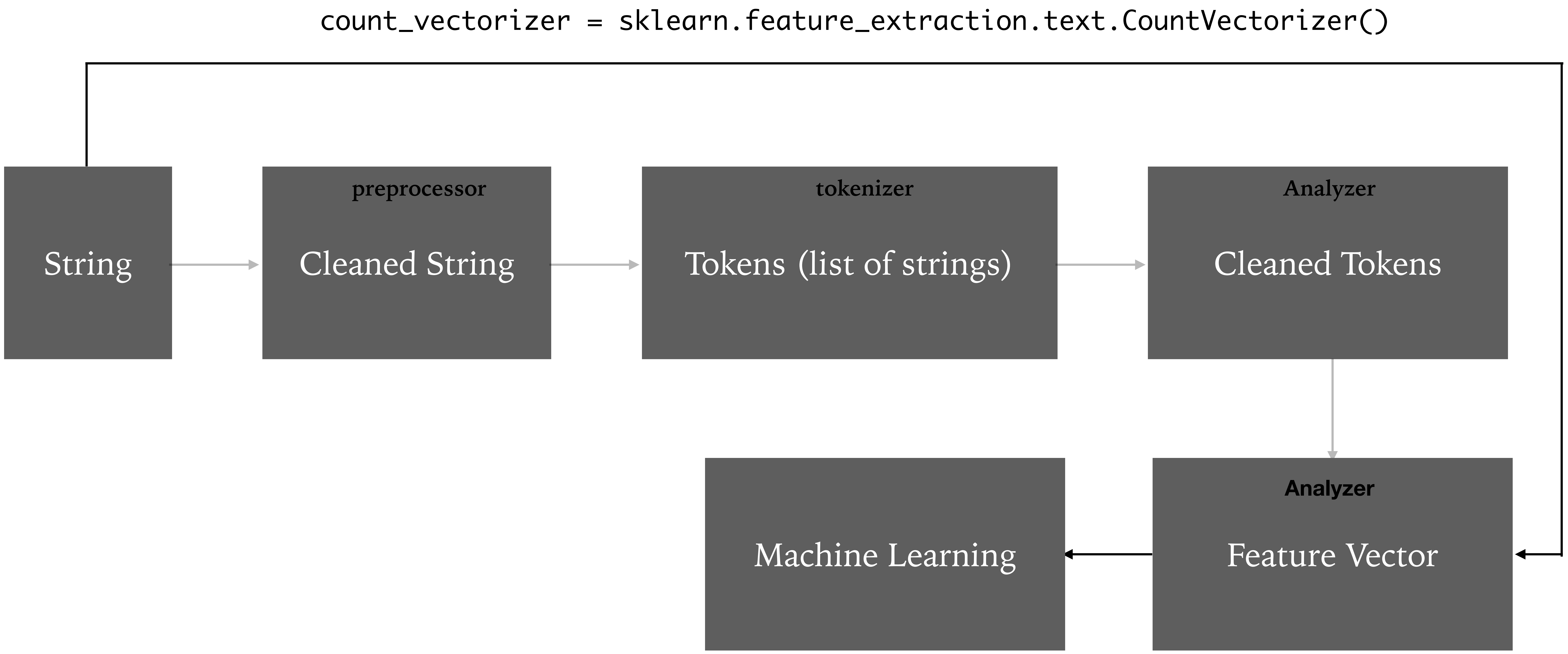

```
count_vectorizer = sklearn.feature_extraction.text.CountVectorizer()
logistic = sklearn.linear_model.LogisticRegression(C=0.1)

model_pipe_3 = sklearn.pipeline.Pipeline([("countvectorizer", count_vectorizer),
                                          ("logisticregression", logistic)],
                                          )# memory='/Users/Shared/sklearn_mem/')

model_pipe_3.fit(X_train,y_train)
```

PIPLELINES OR COMPOSITE MODELS

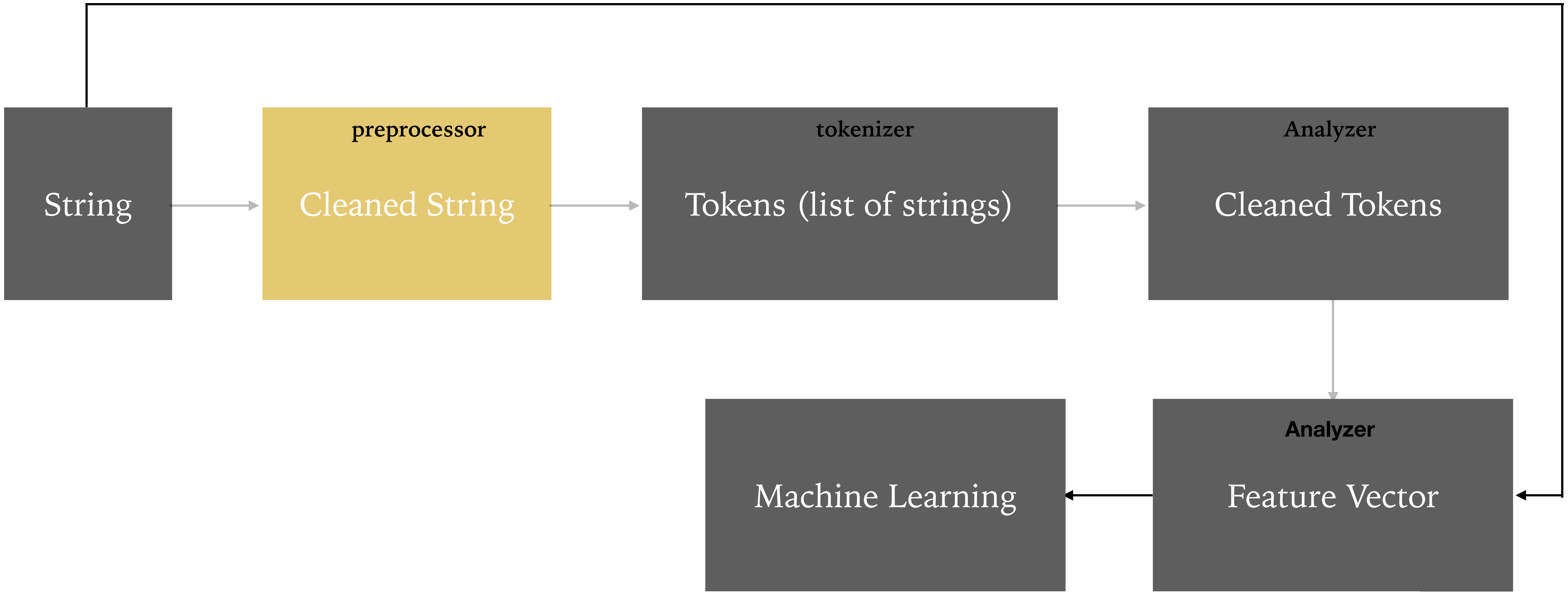
- The purpose of the Pipeline is to assemble a composite model which consist on several steps that can be cross-validated together.
- Pipelines allow practitioners to easily compose and validate decisions made during training, instead of relying on pre-processing steps with ‘hand crafted’ decisions.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

COUNTVECTORIZER

- A CounVectorizer is one of the most straight forward methods to generate descriptors for documents.



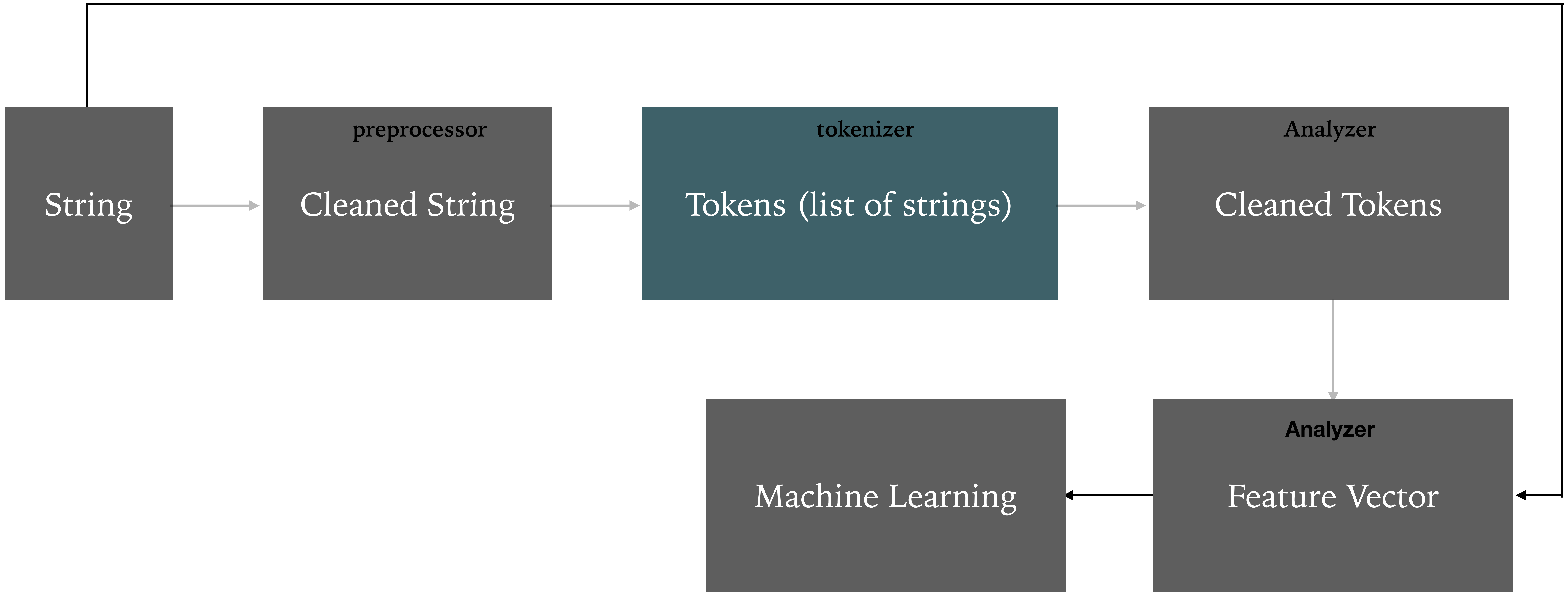
COUNTVECTORIZER: PREPROCESSOR




```
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)
```

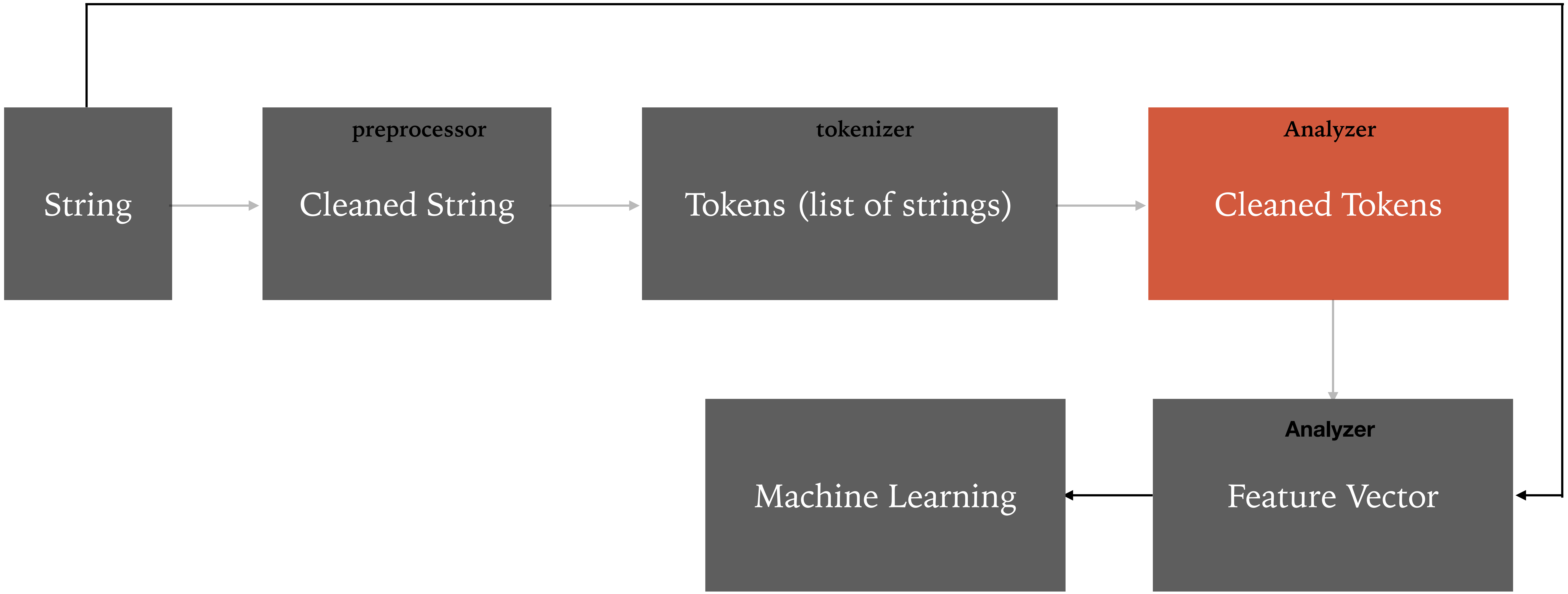
COUNTVECTORIZER: TOKENIZER




```
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)
```

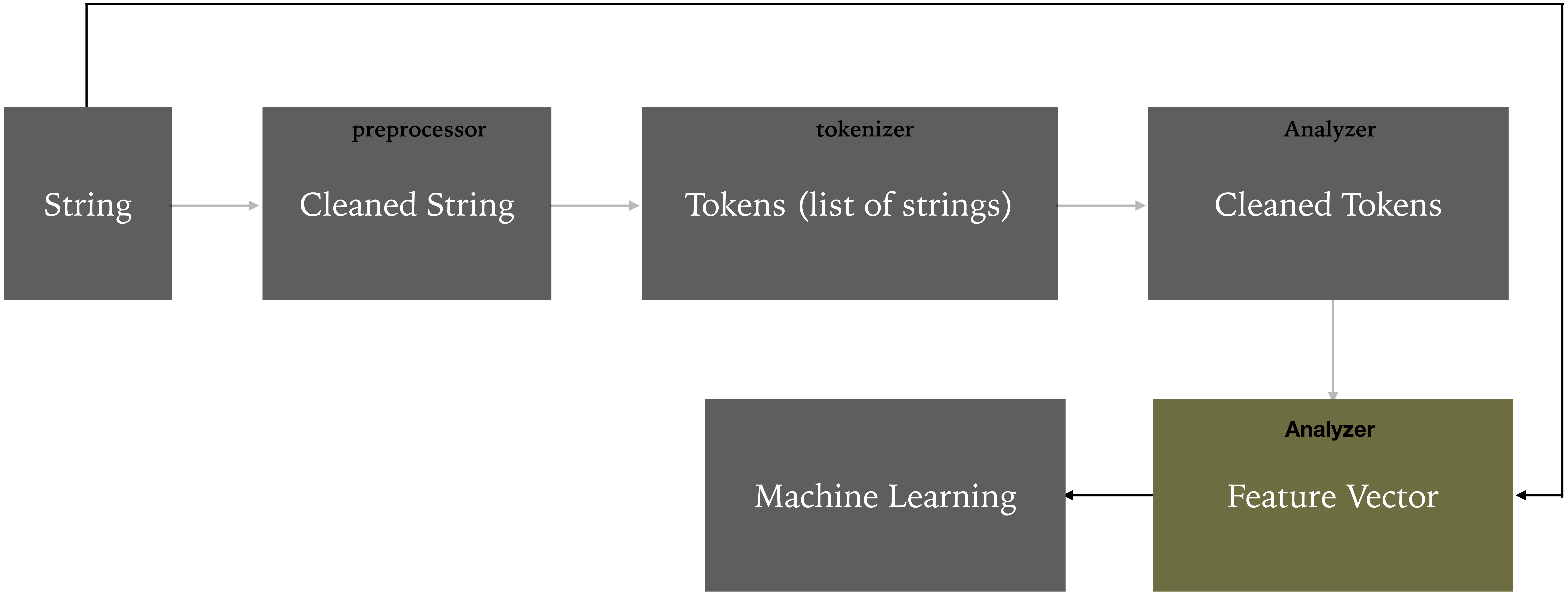
COUNTVECTORIZER: ANALYZER




```
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: FEATURE VECTOR




```
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: CUSTOMISING 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 IT IS IS IMPORTANT TO TUNE VECTORIZERS

- Many times vocabulary can be too rare that is not worth storing it.

```
count_vectorizer = sklearn.feature_extraction.text.CountVectorizer()
count_vectorizer.fit(X_train)
vocabulary = count_vectorizer.vocabulary_.keys()
vocabulary = list(vocabulary)
vocabulary.sort()
```

```
vocabulary
```

```
['00',
 '000',
 '0000',
 '00000',
 '00000000',
 '0000000004',
 '0000000005',
 '00000000b',
 '00000001',
 '00000001b',
 '00000010',
 '00000010b',
 '00000011',
 '00000011b',
 '0000001200',
 '00000074',
 '00000093',
 '000000e5',
 '00000100',
 '00000100b',
```

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