Paraphrase Identification;
Numpy;
Scikit-Learn

Benjamin Roth

Centrum für Informations- und Sprachverarbeitung
Ludwig-Maximilian-Universität München
beroth@cis.uni-muenchen.de
Paraphrase Identification
Paraphrase Identification

- Is a sentence (A) a paraphrase of another sentence (B)?
- Do two tweets contain the same information?
- This is a difficult problem
  - What is a paraphrase?
  - Do two exact paraphrases even exist?
    - paraphrase ⇔ strong similarity, approximately equal meaning
  - Linguistic variation
  - Even more difficult in twitter: abbreviations, spelling errors, ...

- Examples:
  - (A) I hate Mario Chalmers dont know why
    (B) idc idc chalmers be making me mad
  - (A) It fits the larger iPhone 5
    (B) Should I get the iPhone 5 or an Android
SemEval-2015 Task 1: Paraphrase Similarity in Twitter

- ca. 19000 tweet pairs annotated with Amazon Mechanical Turk
- Binary classification: Pair is paraphrase (True) or not (False)
- Brainstorming: good features for recognizing paraphrases?
Strong baseline features

- **Word overlap.**
  - Most simple form: Number common words that occur in both tweets (ignore frequency).
    - "overlap"
  - Needs some normalization (so that there is no bias for longer tweets).
  - Simple solution: Extra feature for number of unique tokens in text1 and text2.
    - "union"

- **Ngram overlap.**
  - Accounts for some ordering information.
  - Otherwise same approach as for word overlap.
  - 3-grams perform well for this task

- **Word-pair features**
  - What if paraphrases use different, but semantically similar words?
  - Learn equivalences from tweets in training data!
  - Features for combinations: Word from text1 with word from text2.

---

1 Thanks to Kevin Falkner for providing extensive feature analysis.
Example: feature representation

(A) happy Memorial Day have a happy weekend
(B) wishing everyone a happy Memorial Day

{"word_overlap":4,
"three_gram_overlap":1,
"word_union":8,
"threegram_union":8,
"happy#wishing":1,
"memorial#everyone":1,
"happy#happy":1,
...}
What is the result of the following list comprehension?

```python
l = ["wishing", "everyone", "a", "happy", "memorial", "day"]
n = 2
[l[i:i+n] for i in range(len(l)-n+1)]
```

How to implement word-pair features?
Data Representation for Machine Learning
Data Representation

- **Dataset**: collection of instances
- **Design matrix**
  \[ X \in \mathbb{R}^{n \times m} \]
  - \( n \): number of instances
  - \( m \): number of features (also called *feature space*).
  - For example: \( X_{i,j} \) count of feature \( j \) (e.g. a stem form) in document \( i \).

**Unsupervised learning**:
- Model \( X \), or find interesting properties of \( X \).
- Training data: only \( X \).

**Supervised learning**:
- Predict *specific* additional properties from \( X \).
- Training data: Label vector \( y \in \mathbb{R}^n \) (or label matrix \( Y \in \mathbb{R}^{n \times k} \)) together with \( X \).
Use matrix $X$ and vector $y$ to stack instances on top of each other.

$$X = \begin{bmatrix}
x_{12} & x_{13} & \ldots & x_{1n} \\
x_{22} & x_{23} & \ldots & x_{2n} \\
\vdots & \vdots & \ddots & \vdots \\
x_{m2} & x_{m3} & \ldots & x_{mn}
\end{bmatrix} \quad y = \begin{bmatrix}
y_1 \\
y_2 \\
\vdots \\
y_m
\end{bmatrix}$$

Binary classification:

$$y = \begin{bmatrix}
0 \\
1 \\
\vdots \\
0
\end{bmatrix} \quad \text{or} \quad \begin{bmatrix}
-1 \\
1 \\
\vdots \\
-1
\end{bmatrix}$$

Multi-class classification (one-hot-encoding):

$$Y = \begin{bmatrix}
0 & 0 & 1 \\
1 & 0 & 0 \\
\vdots \\
0 & 1 & 0
\end{bmatrix}$$
Data Representation

- For performance reasons, machine-learning toolkits (scikit-learn, Keras, ...) use matrix representations (rather than e.g. string-to-count dictionaries).
- These matrix classes usually contain efficient implementations of
  - mathematical operations (matrix multiplication, vector addition ...)
  - data access and transformation (getting a certain row/column, inverting a matrix)
- What would be an appropriate underlying data-structures for the following feature sets:
  - Each feature is the grey-scale value of a pixel in a 100 × 100 gray-scale image?
  - Each feature is the indicator whether a particular word (vocab size 10000) occurs in a document or not?
Data Representation

What would be an appropriate underlying data-structures for the following feature sets:

▶ Each feature is the grey-scale value of a pixel in a 100 × 100 gray-scale image?

Most of the features have a distinct value ≠ 0. The appropriate data structure is similar to a nested list (list-of-lists). ⇒ **Numpy Arrays**

▶ Each feature is the indicator whether a particular word (vocab size 10000) occurs in a document or not?

Most of the features have a value equal to 0. The appropriate data structure only stores those entries that are different than 0. (E.g with a dictionary: (row, col) → value.) ⇒ **SciPy Sparse Matrices**
Introduction to Numpy
What is NumPy?

- Acronym for “Numeric Python”
- Open source extension module for Python.
- Powerful data structures for efficient computation of multi-dimensional arrays and matrices.
- Fast precompiled functions for mathematical and numerical routines.
- Used by many scientific computing and machine learning packages. For example
  - `Scipy` (Scientific Python): Useful functions for minimization, regression, Fourier-transformation and many others.
  - Similar datastructures exist in `Tensorflow, Pytorch`: Deep learning, minimization of custom objective functions, auto-gradients.
- Downloading and installing numpy: [www.numpy.org](http://www.numpy.org)
Python in combination with Numpy, Scipy and Matplotlib can be used as a replacement for MATLAB.

Matplotlib provides MATLAB-like plotting functionality.
Comparison between Core Python and Numpy

“Core Python”: Python without any special modules, i.e. especially without NumPy.

Advantages of Core Python:
- high-level number objects: integers, floating point
- containers: lists with cheap insertion and append methods, dictionaries with fast lookup

Advantages of using Numpy with Python:
- array oriented computing
- efficiently implemented multi-dimensional arrays
- designed for scientific computation
A simple numpy Example

- NumPy needs to be imported. Convention: use short name `np`
  ```python
  import numpy as np
  ```
- Turn a list of temperatures in Celsius into a one-dimensional numpy array:
  ```python
  >>> cvvalues = [25.3, 24.8, 26.9, 23.9]
  >>> np.array(cvvalues)
  [25.3 24.8 26.9 23.9]
  ```
- Turn temperature values into degrees Fahrenheit:
  ```python
  >>> C = 9 / 5 + 32
  [77.54 76.64 80.42 75.02]
  ```
- Compare to using core python only:
  ```python
  >>> [x*9/5 + 32 for x in cvvalues]
  [77.54, 76.64, 80.42, 75.02]
  ```
Creation of evenly spaced values (given stepsize)

- Useful for plotting: Generate values for $x$ and compute $y = f(x)$
- Syntax:
  
  ```python
  np.arange([start], stop[, step[,], dtype=None])
  ```
- Similar to core python `range`, but returns `ndarray` rather than a list iterator.
- Defaults for `start` and `step`: 0 and 1
- `dtype`: If it is not given, the type will be automatically inferred from the other input arguments.
- Don’t use non-integer step sizes (use `linspace` instead).
- Examples:

  ```python
  >>> np.arange(3.0)
  array([0., 1., 2.])
  >>> np.arange(1, 5, 2)
  array([1, 3])
  ```
Creation of evenly spaced values (given number of values)

```
linspace(start, stop, num=50, endpoint=True, retstep=False)
```

- Creates ndarray with `num` values equally distributed between `start` (included) and `stop`.
- If `endpoint=True` (default), the end point is included, otherwise (endpoint=False) it is excluded.

```
>>> np.linspace(1, 3, 5)
array([1. , 1.5, 2. , 2.5, 3. ])
```

```
>>> np.linspace(1, 3, 4, endpoint=False)
array([1. , 1.5, 2. , 2.5])
```

- If `retstep=True`, the stepsize is returned additionally:

```
>>> np.linspace(1, 3, 4, endpoint=False, retstep=True)
(array([1. , 1.5, 2. , 2.5]), 0.5)
```
Exercise

- Compare the speed of vector addition in core Python and Numpy
Multidimensional Arrays

- NumPy arrays can be of arbitrary dimension.
- 0 dimensions (scalar):
  ```python
  np.array(42)
  ```
- 1 dimension (vector):
  ```python
  np.array([3.4, 6.9, 99.8, 12.8])
  ```
- 2 dimensions (matrix):
  ```python
  np.array([[3.4, 8.7, 9.9],
            [1.1,  -7.8, -0.7],
            [4.1,  12.3,  4.8]])
  ```
- 3 or more dimensions (tensor):
  ```python
  np.array([[[111, 112], [121, 122]],
            [[211, 212], [221, 222]],
            [[311, 312], [321, 322]]])
  ```
Question

- When can a 3 dimensional array be an appropriate representation?
Shape of an array

```python
>>> x = np.array([  
    [67, 63, 87],
    [77, 69, 59],
    [85, 87, 99],
    [79, 72, 71],
    [63, 89, 93],
    [68, 92, 78]]
)
```

```python
>>> np.shape(x)
(6, 3)
```
Changing the shape

- **reshape creates new array:**

  ```python
  >>> a = np.arange(12).reshape(3, 4)
  >>> a
  array([[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]])
  ```

- **Changing shape value (for existing array):**

  ```python
  >>> a.shape = (2, 6)
  >>> a
  array([[ 0,  1,  2,  3,  4,  5],
         [ 6,  7,  8,  9, 10, 11]])
  ```

- Obviously, product of shape sizes must match number of elements!
- If a dimension is given as -1 in a reshaping operation, the other dimensions are automatically calculated.
Shape of 3D Array

```python
>>> a = np.arange(24).reshape(2,3,4)
```

```python
>>> a
```

```
array([[[ 0, 1, 2, 3],
        [ 4, 5, 6, 7],
        [ 8, 9, 10, 11]],
       [[12, 13, 14, 15],
        [16, 17, 18, 19],
        [20, 21, 22, 23]]])
```
Transposing an Array

- **2D case:**

  ```
  >>> a = np.arange(6).reshape(2,3)
  array([[0, 1, 2],
          [3, 4, 5]])
  >>> a.T
  array([[0, 3],
          [1, 4],
          [2, 5]])
  ```

- **Multidimensional case:**

  - `a.transpose(...)` takes tuple of indices, indicating which axis of the old (input) array is used for each axis of the new (output) array.
  - 3D example:
    ```
    b = a.transpose(1,0,2)
    ```
    
    - `⇒` axis 1 in `a` is used as axis 0 for `b`, axis 0 (`a`) becomes 1 (`b`), and axis 2 (`a`) stays axis 2 (`b`).
Basic Operations

- By default, arithmetic operators on arrays apply *elementwise*:

  ```python
  >>> a = np.array([20, 30, 40, 50])
  >>> b = np.array([0, 1, 2, 3])
  >>> c = a - b
  array([20, 29, 38, 47])
  >>> b**2
  array([0, 1, 4, 9])
  >>> a<35
  array([ True,  True, False, False], dtype=bool)
  ```

  In particular, the *elementwise multiplication* ...

  ```python
  >>> a * b
  array([  0,  30,  80, 150])
  ```

  ... is not to be confused with the *dot product*:

  ```python
  >>> a.dot(b)
  260
  ```
Unary Operators

- Numpy implements many standard unary (elementwise) operators:

  ```python
  >>> np.exp(b)
  >>> np.sqrt(b)
  >>> np.log(b)
  ```

- For some operators, an axis can be specified:

  ```python
  >>> b = np.arange(12).reshape(3,4)
  array([[ 0,  1,  2,  3],
         [ 4,  5,  6,  7],
         [ 8,  9, 10, 11]])
  
  >>> b.sum(axis=0)
  array([12, 15, 18, 21])
  
  >>> b.min(axis=1)
  array([0, 4, 8])
  ```
Indexing elements

- Indexing single elements:

  ```
  >>> B = np.array([[[[111, 112], [121, 122]],
                   [[211, 212], [221, 222]]],
                   [[311, 312], [321, 322]]])
  >>> B[2][1][0]
  321
  >>> B[2, 1, 0]
  321
  ```

- Indexing entire sub-array:

  ```
  >>> B[1]
  array([[211, 212],
          [221, 222]])
  ```

- Indexing starting from the end:

  ```
  >>> B[-1, -1]
  array([321, 322])
  ```
Indexing with Arrays/Lists of Indices

```python
>>> a = np.arange(12)**2
>>> i = np.array([1,1,3,8,5])
>>> # This also works:
>>> # i = [1,1,3,8,5]
>>> a[i]
array([ 1, 1,  9, 64, 25])
```
Indexing with Boolean Arrays

Boolean indexing is done with a boolean matrix of the same shape (rather than of providing a list of integer indices).

```python
>>> a = np.arange(12).reshape(3, 4)
>>> b = a > 4
array([[False, False, False, False],
       [False,  True,  True,  True],
       [ True,  True,  True,  True]], dtype=bool)

>>> a[b]
array([ 5,  6,  7,  8,  9, 10, 11])

>>> a[b] = 0
array([[0, 1, 2, 3],
       [4, 0, 0, 0],
       [0, 0, 0, 0]])
```
Slicing

- Syntax for slicing lists and tuples can be applied to multiple dimensions in NumPy.

- Syntax:
  
  \[ A[\text{start0:stop0:step0, start1:stop1:step1, ...}] \]

- Example in 1 dimension:

  ```python
  >>> S = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
  >>> S[3:6:2]
  array([3, 5])
  >>> S[:4]
  array([0, 1, 2, 3])
  >>> S[4:]
  array([4, 5, 6, 7, 8, 9])
  >>> S[:]
  array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
  ```
Slicing 2D

\[ A = \text{np.arange}(25).\text{reshape}(5,5) \]
\[ B = A[::3,2:] \]
\[ B = A[3:,::] \]

\[ X = \text{np.arange}(28).\text{reshape}(4,7) \]
\[ Y = X[::2,::3] \]
\[ Y = X[:,::3] \]
Slicing: Caveat

- Slicing only creates a new **view**: the underlying data is shared with the original array.

```python
>>> A = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> S[0] = 22
>>> S[1] = 23
>>> A
array([ 0,  1, 22, 23,  4,  5,  6,  7,  8,  9])
```

- If you want a deep copy that does not share elements with A, use: `A[2:6].copy()`
Quiz

What is the value of $b$?

```python
>>> a = np.arange(4)
>>> b = a [:]
>>> a *= b
```
Arrays of Ones and of Zeros

```python
>>> np.ones((2,3))
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])

>>> a = np.ones((3,4), dtype=int)
array([[1, 1, 1, 1],
       [1, 1, 1, 1],
       [1, 1, 1, 1]])

>>> np.zeros((2,4))
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])

>>> np.zeros_like(a)
array([[0, 0, 0, 0],
       [0, 0, 0, 0],
       [0, 0, 0, 0]])
```
Creating Random Matrices

- Array of floats uniformly drawn from the interval $[0, 1)$:
  ```python
  >>> np.random.rand(2,3)
  array([[0.53604809, 0.54046081, 0.84399025],
         [0.59992296, 0.51895053, 0.09988041]])
  ```

- Generate floats drawn from standard normal distribution $\mathcal{N}(0, 1)$:
  ```python
  >>> np.random.randn(2,3)
  array([[-1.28520219, -1.02882158, -0.20196267],
         [ 0.48258382, -0.2077209 , -2.03846176]])
  ```

- For repeatability of your experiment, initialize the seed at the beginning of your script:
  ```python
  >>> np.random.seed = 0
  ```
  Otherwise, it will be initialized differently at every run (from system clock).
  ```python
  >>> import random
  >>> random.seed(9001)
  ```
Creating Diagonal Matrices

- \texttt{eye(N, M=None, k=0, dtype=float)}
  - \texttt{N} Number of rows.
  - \texttt{M} Number of columns.
  - \texttt{k} Diagonal position.
    - 0: main diagonal, starting at \((0, 0)\)
    - \(+n, –n\): move diagonal \(n\) up/down
  - \texttt{dtype} Data type (e.g. int or float)

\[ k= \]

\[
\begin{array}{cccccccc}
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
  & & & & & & & \\
\end{array}
\]

- To create an identity matrix (symmetric \(N = M, k = 1\)) the size \(N\) is the only argument.
Iterating

Iterating over rows:

```python
>>> for row in b:
    ...    print(row)
...
[0  1  2  3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

⇒ but (!) prefer matrix operations over iterating, if possible.
Stacking of arrays

- **Vertical stacking:**

  ```python
  >>> a = np.array([[1, 2], [3, 4]])
  >>> b = np.array([[11, 22], [33, 44]])
  >>> np.vstack((a, b))
  array([[ 1,  2],
         [ 3,  4],
         [11, 22],
         [33, 44]])
  ```

- **Horizontal stacking:**

  ```python
  >>> np.hstack((a, b))
  array([[ 1,  2, 11, 22],
         [ 3,  4, 33, 44]])
  ```
Broadcasting

Operations can work on arrays of different sizes if Numpy can transform them so that they all have the same size!
Plotting data

- Often it is a good idea to plot some properties of the data.
  - Verify expectations that you have about the data.
  - Spot trends, maxima/minima, (ir-)regularities and outliers.
  - Similarities / dissimilarities between two data sets.

- Recommended package: Matplotlib/Pyplot
Pyplot

- Plotting data and functions with Python.

- Package of the matplotlib library.
- Uses numpy data structures
- Inspired by the matlab plotting commands
- Import pyplot as:
  import matplotlib.pyplot as plt
Example: Histograms

- Show the empirical distribution of one variable.
- Frequency of values with equally-spaced intervals.

```python
x = 100 + 15 * np.random.randn(10000)
plt.hist(x, 50)
```
Ressources

- NumPy Quickstart:
  http://docs.scipy.org/doc/numpy-dev/user/quickstart.html
Scipy Sparse Matrices
Scipy Sparse Matrices

- **SciPy** is another package of the Python scientific computing stack. (NumPy+SciPy+Matplotlib=Matlab)
- `scipy.sparse` contains a range of sparse matrix implementations
  - Different underlying datastructures
  - slightly different use-cases
- **All implementations**
  - inherit from the same base class
  - provide basic matrix operations, e.g.:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_shape()</code></td>
<td>Get shape of a matrix.</td>
</tr>
<tr>
<td><code>getnnz()</code></td>
<td>Number of stored values, including explicit zeros.</td>
</tr>
<tr>
<td><code>transpose([axes, copy])</code></td>
<td>Reverses the dimensions of the sparse matrix.</td>
</tr>
<tr>
<td><code>+</code>, <code>__add__</code>(other)</td>
<td>Add two matrices.</td>
</tr>
<tr>
<td><code>*</code>, <code>__mul__</code>(other), <code>dot</code>(other)</td>
<td>Matrix (vector) multiplication</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
## Types of Sparse Matrices

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>scipy.sparse.csr_matrix</code></td>
<td>Compressed Sparse Row matrix (default). Very efficient format for arithmetic operations. (Less efficient for column slicing or inserting/removing values)</td>
</tr>
<tr>
<td><code>scipy.sparse.lil_matrix</code></td>
<td>Row-based linked list sparse matrix. Efficient changes to matrix structure. (Less efficient for column slicing or arithmetic)</td>
</tr>
<tr>
<td><code>scipy.sparse.coo_matrix</code></td>
<td>A sparse matrix in COOrdinate format. Triples of row, column and value. Good (only) for building matrices from coordinates. For any operations convert to CSR or LIL.</td>
</tr>
<tr>
<td><code>scipy.sparse.dia_matrix</code></td>
<td>Sparse matrix with DIAgonal storage</td>
</tr>
</tbody>
</table>
import numpy as np
from scipy.sparse import csr_matrix
A = csr_matrix([[1, 2, 0], [0, 0, 3], [4, 0, 5]])
v = np.array([1, 0, -1])
A.dot(v)
Memory Saving

- **Given:**
  - 1000 docs
  - 100 unique words per doc
  - 10000 vocabulary size

- What is the expected percentage of memory used by sparse matrix (compared to dense)?
from timeit import default_timer as timer
from scipy import sparse
import numpy as np
rnd = np.random.RandomState(seed=123)

X = rnd.uniform(low=0.0, high=1.0, size=(200000, 1000))
v = rnd.uniform(low=0.0, high=1.0, size=(1000,1))
X[X<0.99]=0
v[v<0.99]=0
X_csr = sparse.csr_matrix(X)
v_csr = sparse.csr_matrix(v)

start = timer()
X_2 = X.dot(v)
time_dense = timer() - start

start = timer()
X_2 = X_csr.dot(v_csr)
time_sparse = timer() - start

⇒ 0.16 seconds (time_dense) vs. 0.01 seconds (time_sparse)
Scikit-learn Data Structures

Now that we know about Numpy (dense) matrices, and Scipy (sparse) matrices, let’s see how we can use them for machine learning with Scikit-learn.
Scikit-learn Vectorizers

- For efficiency, Scikit-learn uses matrices for its algorithms.
- However, data is often present in different forms (text; dictionaries: feature $\rightarrow$ count; ...)
- Scikit-learn provides **Vectorizers** to convert other data-types into matrices.
- A vectorizer object provides a mapping (e.g. from vocabulary to column indices): it is important that the same mapping is used for training, test and dev data!
- For most vectorizers, one can choose whether Dense or Sparse representation is preferred.
Loading features from dicts

- DictVectorizer can be used to convert feature arrays represented as lists of dict objects to the NumPy/SciPy representation.
- **Input:** one dict per instance (feature counts)
  - key: feature
  - value: observed value of that feature
- **Output:** Design matrix

The vectorizer constructs a feature map - use the same feature map for new data! (I.e. do not create a new feature map).

Values of the dictionary can be:
- **Numerical:** the numerical value is stored in the resulting matrix in the column for that feature.
- **Boolean:** two columns are created in the matrix for that feature.
- **String:** several columns are created in the matrix, one for each possible value for that feature.
DictVectorizer: Example

```python
>>> measurements = [
...     {'city': 'Dubai', 'temperature': 33.},
...     {'city': 'London', 'temperature': 12.},
...     {'city': 'San Francisco', 'temperature': 18.},
... ]

>>> from sklearn.feature_extraction import DictVectorizer
>>> v = DictVectorizer()

>>> v.fit_transform(measurements).toarray()
array([[ 1.,  0.,  0.,  33.],
       [ 0.,  1.,  0.,  12.],
       [ 0.,  0.,  1.,  18.]])

>>> v.get_feature_names()
['city=Dubai', 'city=London', 'city=San Francisco', 'temperature']
```
DictVectorizer

- Creates sparse matrices by default, can be changed to dense.
- `v.fit(list_of_dicts)`: Creates and stores a mapping from features to matrix columns.
- `v.transform(list_of_other_dicts)`: Applies the stored mapping to (potentially new) dictionaries.
- `v.fit_transform(list_of_dicts)`: fit and transform in one step.

```python
>>> v = DictVectorizer(sparse=False)
>>> D = [{'foo': 1, 'bar': 2}, {'foo': 3, 'baz': 1}]
>>> X = v.fit_transform(D)
>>> X
array([[ 2., 0., 1.],
       [ 0., 1., 3.]])
>>> v.inverse_transform(X) == [{'bar': 2.0, 'foo': 1.0},
                                {'baz': 1.0, 'foo': 3.0}]
True
>>> v.transform({'foo': 4, 'unseen_feature': 3})
array([[ 0., 0., 4.]])
```
Feature hashing

- **Hash function** (not a rigorous definition, but sufficient for our purposes):
  - Function that maps every object from input space to an integer in pre-specified range
  - Regularities (e.g. sequential order) from input space are not preserved in output space, assignment looks random (for properties of interest)

- **Hash collision**: Two different values from input space are mapped to same output

- **Applications of hash functions?**
Feature hashing

- Large amounts of features also means many model parameters to learn and store (no sparsity here)
- One way of fighting amount of features: sort and take most frequent.
- Another way: use hash function to “randomly” group features together
- Hashing trick:
  - Input space: features
  - Output space: columns in design matrix
- FeatureHasher: Vectorizer that uses the hashing trick.
  $\Rightarrow$ inverse transform is not possible
CountVectorizer: Transforming text into a design matrix

- SciPy CountVectorizer provides some functionality to create feature matrices from raw text
  - tokenization
  - lowercasing
  - ngram creation
  - occurrence counting
  - filtering by minimum word length (default=2)
  - filtering by minimum and maximum document frequency.
  - ...

- Use cases:
  - **CountVectorizer:** Very convenient for standard usage!
  - **DictVectorizer:** You have more control if you create the features (dictionaries) yourself and use DictVectorizer
CountVectorizer

```python
>>> from sklearn.feature_extraction.text import CountVectorizer
>>> corpus = [
...    'This is the first document.',
...    'This is the second second document.',
...    'And the third one.',
...    'Is this the first document?',
...]
>>> vectorizer = CountVectorizer()
>>> X = vectorizer.fit_transform(corpus)
>>> vectorizer.get_feature_names()
['and', 'document', 'first', 'is', 'one', 'second', 'the', 'third', 'this']
>>> X.toarray()
array([[0, 1, 1, 1, 0, 0, 1, 0, 1],
       [0, 1, 0, 1, 0, 2, 1, 0, 1],
       [1, 0, 0, 0, 1, 0, 1, 1, 0],
       [0, 1, 1, 1, 0, 0, 1, 0, 1]], dtype=int64)
```
CountVectorizer: unigrams, bigrams, document frequency

```python
>>> vectorizer = CountVectorizer(min_df=2, ngram_range=(1, 2))
>>> X = vectorizer.fit_transform(corpus)
>>> vectorizer.get_feature_names()
['document', 'first', 'first document', 'is', 'is the', 'the', 'the first', 'this', 'this is']
>>> X.toarray()
array([[1, 1, 1, 1, 1, 1, 1, 1, 1],
       [1, 0, 0, 1, 1, 1, 0, 1, 1],
       [0, 0, 0, 0, 0, 1, 0, 0, 0],
       [1, 1, 1, 1, 0, 1, 1, 1, 0]], dtype=int64)
```
Summary

- **Features for Paraphrase identification**
  - Number of overlapping words and ngrams
  - Normalization for tweet length
  - Word pair features

- **Dense and Sparse Matrices**
  - Numpy arrays
    docs.scipy.org/doc/numpy-dev/user/quickstart.html
  - Scipy sparse matrices
    docs.scipy.org/doc/scipy-0.18.1/reference/sparse.html

- **Scikit-learn Vectorizers**

- **Questions?**