GENERAL ASSEMBLY

INTRO TO DATA SCIENCE

LECTURE 17: DATASET TRANSFORMATIONS

AGENDA

I. FEATURE EXTRACTION

II. PREPROCESSING DATA

I. FEATURE EXTRACTION

Feature Extraction:
Transforming arbitrary data into numerical features usable for machine learning

Loading Features from Dictionaries

The class **DictVectorizer** can be used to convert lists of Python dictionaries to the NumPy/SciPy matrices.

FEATURE EXTRACTION

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

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

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

>>> vec.get_feature_names()
['city=Dubai', 'city=London', 'city=San Fransisco', 'temperature']
```

Feature Hashing

Uses a hash function to map a feature to a number (column), size of which you specify.

Pros: Reduces dimensionality, fast, constant memory

Cons: Collisions, No way to invert (go from vectorized to original feature)

Feature Hashing

```
>>> corpus = [
...     'This is the first document.',
...     'This is the second second document.',
...     'And the third one.',
...     'Is this the first document?',
... ]
```

Feature Hashing

```
>>> from sklearn.feature_extraction.text import HashingVectorizer
>>> hv = HashingVectorizer(n_features=10)
>>> hv.transform(corpus)
...
<4x10 sparse matrix of type '<... 'numpy.float64'>'
    with 16 stored elements in Compressed Sparse Row format>
```

Drop-in replacement for CountVectorizer

Bag of Words

Documents are described by word occurrences while completely ignoring the relative position information of the words in the document.

We call vectorization the general process of turning a collection of text documents into numerical feature vectors.

This specific strategy (tokenization, counting and normalization) is called the Bag of Words or “Bag of n-grams” representation.

Bag of Words

```
from sklearn.feature_extraction.text import CountVectorizer
vectorizer = CountVectorizer(min_df=1)
corpus = [
   'This is the first document.',
   'This is the second second document.',
   'And the third one.',
   'Is this the first document?',
]
X = vectorizer.fit_transform(corpus)
print X
  (2, 0)        1
  (0, 1)        1
  (1, 1)        1
  (3, 1)        1
…
```

Bag of Words

```
In [5]: vectorizer.vocabulary_
Out[5]:
{u'and': 0,
 u'document': 1,
 u'first': 2,
 u'is': 3,
 u'one': 4,
 u'second': 5,
 u'the': 6,
 u'third': 7,
 u'this': 8}
```

Tf–idf term weighting

Tf means *term-frequency*

tf–idf means *term-frequency times inverse document-frequency*.

This is a originally a term weighting scheme developed for information retrieval (as a ranking function for search engines results), that has also found good use in document classification and clustering.

Tf–idf term weighting

```
>>> from sklearn.feature_extraction.text import TfidfVectorizer
>>> vectorizer = TfidfVectorizer(min_df=1)
>>> vectorizer.fit_transform(corpus)
...
<4x9 sparse matrix of type '<... 'numpy.float64'>'
    with 19 stored elements in Compressed Sparse Row format>
```

Tf–idf and its many variants

Term frequency		Document frequency		Normalization	
n (natural)	$\mathrm{tf}_{t,d}$	n (no)	1	n (none)	1
l (logarithm)	$1 + \log(\mathrm{tf}_{t,d})$	t (idf)	$\log \frac{N}{\mathrm{df}_t}$	c (cosine)	$\frac{1}{\sqrt{w_1^2+w_2^2+\ldots+w_M^2}}$
a (augmented)	$0.5 + \frac{0.5\times\mathrm{tf}_{t,d}}{\max_t(\mathrm{tf}_{t,d})}$	p (prob idf)	$\max\{0, \log \frac{N-\mathrm{df}_t}{\mathrm{df}_t}\}$	u (pivoted unique)	$1/u$
b (boolean)	$\begin{cases} 1 & \text{if } \mathrm{tf}_{t,d} > 0 \\ 0 & \text{otherwise} \end{cases}$			b (byte size)	$1/CharLength^{\alpha}$, $\alpha < 1$
L (log ave)	$\frac{1+\log(\mathrm{tf}_{t,d})}{1+\log(\mathrm{ave}_{t\in d}(\mathrm{tf}_{t,d}))}$				

Pro-tip: Cleaning up Text with *ftfy*

https://github.com/LuminosoInsight/python-ftfy

```
>>> from __future__ import unicode_literals
>>> from ftfy import fix_text

>>> print(fix_text('This â€” should be an em dash'))
This — should be an em dash

>>> print(fix_text('uÌˆnicode'))
ünicode

>>> print(fix_text('Broken text&hellip; it&#x2019;s ﬂubberiﬁc!'))
Broken text... it's flubberific!

>>> print(fix_text('HTML entities &lt;3'))
HTML entities <3

>>> print(fix_text('<em>HTML entities &lt;3</em>'))
<em>HTML entities &lt;3</em>
```

Limitations of the Bag of Words representation

Bag of Words Unigrams do not capture:

* Phrases
* Multi-word expressions
* Order dependence
* Misspellings
* Word derivations

Limitations of the Bag of Words representation

Can be partially addressed by:

* N-grams
* Character N-grams
* Stemming
* Spell Correction

Limitations of the Bag of Words representation

```
>>> ngram_vectorizer = CountVectorizer(analyzer='char_wb', ngram_range=(5, 5),
min_df=1)
>>> ngram_vectorizer.fit_transform(['jumpy fox'])
>>> ngram_vectorizer.get_feature_names() == (
...     [' fox ', ' jump', 'jumpy', 'umpy '])
True

>>> ngram_vectorizer = CountVectorizer(analyzer='char', ngram_range=(5, 5),
min_df=1)
>>> ngram_vectorizer.fit_transform(['jumpy fox'])
>>> ngram_vectorizer.get_feature_names() == (
...     ['jumpy', 'mpy f', 'py fo', 'umpy ', 'y fox'])
True
```

II. PRE-PROCESSING DATA

Standardization, or **Mean removal** and **variance scaling**

Standardization of datasets is a common requirement for many machine learning estimators:

They might behave badly if the individual feature do not more or less look like standard normally distributed data: Gaussian with zero mean and unit variance.

Standardization

The **preprocessing** module provides a **StandardScaler** to compute the *mean* and *standard deviation* on a training, to be reapplied to a testing set.

Standardization

```
>>> from sklearn import preprocessing
>>> import numpy as np
>>> X = np.array([[ 1., -1.,  2.],
...               [ 2.,  0.,  0.],
...               [ 0.,  1., -1.]])
>>> scaler = preprocessing.StandardScaler().fit(X)
>>> scaler.mean_
array([ 1. ...,  0. ...,  0.33...])

>>> scaler.std_
array([ 0.81...,  0.81...,  1.24...])
., -0.26...]])
```

Standardization

```
>>> scaler.transform(X)
array([[ 0.  ..., -1.22...,  1.33...],
       [ 1.22...,  0.  ..., -0.26...],
       [-1.22...,  1.22..., -1.06…]])

>>> scaler.transform([[-1.,  1., 0.]])
array([[-2.44...,  1.22..
```

Scaling features to a range

An alternative standardization is scaling features to lie between a given minimum and maximum value, often between zero and one.

This can be achieved using **MinMaxScaler**.

Scaling features to a range

```
>>> from sklearn import preprocessing
>>> import numpy as np
>>> min_max_scaler = preprocessing.MinMaxScaler()
>>> X_train_minmax = min_max_scaler.fit_transform(X_train)
>>> X_train_minmax
array([[ 0.5       ,  0.        ,  1.        ],
       [ 1.        ,  0.5       ,  0.33333333],
       [ 0.        ,  1.        ,  0.        ]])
```

Scaling features to a range

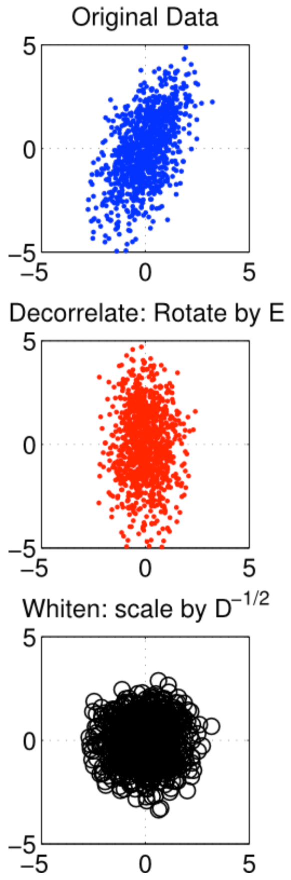
The same instance of the transformer can then be applied to some new test data unseen during the fit call

```
>>> X_test = np.array([[ -3., -1.,  4.]])
>>> X_test_minmax = min_max_scaler.transform(X_test)
>>> X_test_minmax
array([[-1.5       ,  0.        ,  1.66666667]])
```

Scaling vs Whitening

It is sometimes not enough to center and scale the features independently, since a downstream model can further make some *assumption on the **linear independence** of the features*.

To address this issue you can use **sklearn.decomposition.PCA** or **sklearn.decomposition.RandomizedPCA** with **whiten=True** to further remove the linear correlation across features.



Normalization

Normalization is the process of scaling individual samples to have **unit norm**.

This assumption is the base of the Vector Space Model often used in text classification and clustering contexts.

Normalization

The function **normalize** provides a quick and easy way to perform this operation on a single array-like dataset, either using the **l1** or **l2** norms:

```
>>> X = [[ 1., -1.,  2.],
...      [ 2.,  0.,  0.],
...      [ 0.,  1., -1.]]
# fit does nothing, but is required for the 'transformer' API.
>>> normalizer = preprocessing.Normalizer().fit(X)
>>> normalizer.transform(X) # Now apply it to some data
array([[ 0.40..., -0.40...,  0.81...],
       [ 1.  ...,  0.  ...,  0.  ...],
       [ 0.  ...,  0.70..., -0.70...]])
>>> normalizer.transform([[-1.,  1., 0.]])
array([[-0.70...,  0.70...,  0.  ...]])
```

Feature binarization

Feature binarization is the process of thresholding numerical features to get boolean values. This can be useful for downstream probabilistic estimators that make assumptions that the input data is distributed according to a multi-variate Bernoulli distribution.

It is also common among the text processing community to use binary feature values (probably to simplify the probabilistic reasoning) even if normalized counts (a.k.a. term frequencies) or TF-IDF valued features often perform slightly better in practice.

Feature binarization

The function **normalize** provides a quick and easy way to perform this operation on a single array-like dataset, either using the **`l1`** or **`l2`** norms:

```
>>> X = [[ 1., -1.,  2.],
...      [ 2.,  0.,  0.],
...      [ 0.,  1., -1.]]
>>> binarizer = preprocessing.Binarizer().fit(X)  # fit does nothing
>>> binarizer
Binarizer(copy=True, threshold=0.0)

>>> binarizer.transform(X)
array([[ 1.,  0.,  1.],
       [ 1.,  0.,  0.],
       [ 0.,  1.,  0.]])
```

Encoding categorical features

Often features are not given as continuous values but categorical.

For example a person could have features
["male", "female"],
["from Europe", "from US", "from Asia"],
["uses Firefox", "uses Chrome", "uses Safari", "uses Internet Explorer"].

Such features can be efficiently coded as integers, for instance
["male", "from US", "uses Internet Explorer"] could be expressed as
[0, 1, 3] while
["female", "from Asia", "uses Chrome"] would be
[1, 2, 1].

Encoding categorical features

Such integer representation can not be used directly with scikit-learn estimators, as these expect continuous input, and would interpret the categories as being ordered, which is often not desired (i.e. the set of browsers was ordered arbitrarily).

One possibility to convert categorical features to features that can be used with scikit-learn estimators is to use a **one-of-K** or **one-hot encoding**, which is implemented in **OneHotEncoder**.

This estimator transforms each categorical feature with **m** possible values into **m binary features**, with only one active.

Imputation of missing values

For various reasons, many real world datasets contain missing values, often encoded as blanks, NaNs or other placeholders. **scikit-learn** estimators assume that all values in an array are numerical, and that all have and hold meaning.

You can drop rows with missing values.

A better strategy is to **impute** the missing values, i.e., to infer them from the known part of the data.

Imputation of missing values

The **Imputer** class provides basic strategies for imputing missing values, either using the **mean**, the **median** or the **most frequent value** of the row or column in which the missing values are located.

The following snippet demonstrates how to replace missing values, encoded as **np.nan**, using the mean value of the columns (axis 0) that contain the missing values:

```
>>> import numpy as np
>>> from sklearn.preprocessing import Imputer
>>> imp = Imputer(missing_values='NaN', strategy='mean', axis=0)
>>> imp.fit([[1, 2], [np.nan, 3], [7, 6]])
Imputer(axis=0, copy=True, missing_values='NaN', strategy='mean', verbose=0)
>>> X = [[np.nan, 2], [6, np.nan], [7, 6]]
>>> print(imp.transform(X))
[[ 4.          2.        ]
 [ 6.          3.66666667]
 [ 7.          6.        ]]
```