**Extracting, transforming and selecting features**

This section covers algorithms for working with features, roughly divided into these groups:

* Extraction: Extracting features from “raw” data
* Transformation: Scaling, converting, or modifying features
* Selection: Selecting a subset from a larger set of features

**Extractors:**

* **TF-IDF**

In information retrieval or text mining, the [term frequency – inverse document frequency](http://en.wikipedia.org/wiki/Tf%E2%80%93idf) (also called **tf-idf**), is a well know method to evaluate how important is a word in a document. tf-idf are is a very interesting way to convert the textual representation of information into a [Vector Space Model](http://en.wikipedia.org/wiki/Vector_space_model) (VSM), or into sparse features

**VSM** - is a space where text is represented as a vector of numbers instead of its original string textual representation; the VSM represents the features extracted from the document.

### Going to the vector space

1. In modeling the document into a vector space is to create a dictionary of terms present in documents. To do that, you can simple select all terms from the document and convert it to a dimension in the vector space, but we know that there are some kind of words (stop words) that are present in almost all documents, and what we’re doing is extracting important features from documents, features do identify them among other similar documents, so using terms like “the, is, at, on”, etc.. Isn’t going to help us, so in the information extraction, we’ll just ignore them.

**Train Document Set:**

d1: The sky is blue.

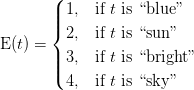
d2: The sun is bright.

**Test Document Set:**

d3: The sun in the sky is bright.

d4: We can see the shining sun, the bright sun.

1. We have to do is to create a index vocabulary (dictionary) of the words of the train document set, using the documents d1 and d2 from the document set, we’ll have the following index vocabulary denoted as \mathrm{E}(t) where the t is the term:



1. We can convert the test document set into a vector space where each term of the vector is indexed as our index vocabulary, so the first term of the vector represents the “blue” term of our vocabulary, the second represents “sun” and so on. Now, we’re going to use the **term-frequency** to represent each term in our vector space; the term-frequency is nothing more than a measure of how many times the terms present in our vocabulary \mathrm{E}(t) are present in the documents d3 or d4, we define the term-frequency as a couting function:

   \mathrm{tf}(t,d) = \sum\limits_{x\in d} \mathrm{fr}(x, t)   

where the \mathrm{fr}(x, t) is a simple function defined as:

   \mathrm{fr}(x,t) =   \begin{cases}   1, & \mbox{if } x = t \\   0, & \mbox{otherwise} \\   \end{cases}   

So, what the tf(t,d) returns is how many times is the term t is present in the document d. An example of this, could be  tf(``sun'', d4) = 2  since we have only two occurrences of the term “sun” in the document d4. Now you understood how the term-frequency works, we can go on into the creation of the document vector, which is represented by:

Each dimension of the document vector is represented by the term of the vocabulary, for example, the \mathrm{tf}(t_1,d_2) represents the frequency-term of the term 1 or t_1 (which is our “blue” term of the vocabulary) in the documentd_2.

Let’s now show a concrete example of how the documents d_3 and d_4are represented as vectors:



which evaluates to:

   \vec{v_{d_3}} = (0, 1, 1, 1) \\   \vec{v_{d_4}} = (0, 2, 1, 0)   

The resulting vector \vec{v_{d_3}} shows that we have, in order, 0 occurrences of the term “blue”, 1 occurrence of the term “sun”, and so on. In the \vec{v_{d_3}}, we have 0 occurences of the term “blue”, 2 occurrences of the term “sun”, etc.

But wait, since we have a collection of documents, now represented by vectors, we can represent them as a matrix with |D| \times F shape, where |D| is the cardinality of the document space, or how many documents we have and the F is the number of features, in our case represented by the vocabulary size. An example of the matrix representation of the vectors described above is:

   M_{|D| \times F} =   \begin{bmatrix}   0 & 1 & 1 & 1\\   0 & 2 & 1 & 0   \end{bmatrix}   

As you may have noted, these matrices representing the term frequencies tend to be very [sparse](http://en.wikipedia.org/wiki/Sparse_matrix) (with majority of terms zeroed), and that’s why you’ll see a common representation of these matrix as sparse matrices.

* **Count Vectorizer:**

The **CountVectorizer** already uses as default “analyzer” called **WordNGramAnalyzer**, which is responsible to convert the text to lowercase, accents removal, token extraction, filter stop words, etc… you can see more information by printing the class information. Since we already defined our small train/test dataset before, let’s use them to define the dataset

**train\_set** = ("The sky is blue.", "The sun is bright.")

**test\_set** = ("The sun in the sky is bright.","We can see the shining sun, the bright sun.")

what we have presented as the term-frequency, is called **CountVectorizer**, so we need to import it and create a news instance:

from sklearn.feature\_extraction.text import CountVectorizer

Creating the vocabulary index:

the vocabulary created is the same as E(t) (except because it is zero-indexed)

Let’s use the same vectorizer now to create the sparse matrix of our **test\_set** documents:

Note that the sparse matrix created called **smatrix** is a [Scipy sparse matrix](http://www.scipy.org/doc/api_docs/SciPy.sparse.sparse.coo_matrix.html" \o "Scipy API Docs :: COOrdinate format" \t "_blank) with elements stored in a [Coordinate format](http://en.wikipedia.org/wiki/Sparse_matrix#Coordinate_list_.28COO.29). But you can convert it into a dense format:

Note that the sparse matrix created is the same matrix M_{|D| \times F} we cited earlier in this post, which represents the two document vectors \vec{v_{d_3}} and \vec{v_{d_4}}.

We learned how to use the **term-frequency** to represent textual information in the vector space. However, the main problem with the term-frequency approach is that it scales up frequent terms and scales down rare terms which are empirically more informative than the high frequency terms. The basic intuition is that a term that occurs frequently in many documents is not a good discriminator, and really makes sense (at least in many experimental tests); the important question here is: why would you, in a classification problem for instance, emphasize a term which is almost present in the entire corpus of your documents?

The tf-idf weight comes to solve this problem. What tf-idf gives is how important is a word to a document in a collection, and that’s why tf-idf incorporates local and global parameters, because it takes in consideration not only the isolated term but also the term within the document collection. What tf-idf then does to solve that problem, is to scale down the frequent terms while scaling up the rare terms; a term that occurs 10 times more than another isn’t 10 times more important than it, that’s why tf-idf uses the logarithmic scale to do that.

But let’s go back to our definition of the \mathrm{tf}(t,d) which is actually the term count of the term t in the document d. The use of this simple term frequency could lead us to problems like keyword spamming, which is when we have a repeated term in a document with the purpose of improving its ranking on an IR (Information Retrieval) system or even create a bias towards long documents, making them look more important than they are just because of the high frequency of the term in the document.

To overcome this problem, the term frequency \mathrm{tf}(t,d) of a document on a vector space is usually also normalized. Let’s see how we normalize this vector.

### Vector normalization: Suppose we are going to normalize the term-frequency vector \vec{v_{d_4}}that we have calculated in the first part of this tutorial. The document d4 from the first part of this tutorial had this textual representation:

d4: We can see the shining sun, the bright sun.

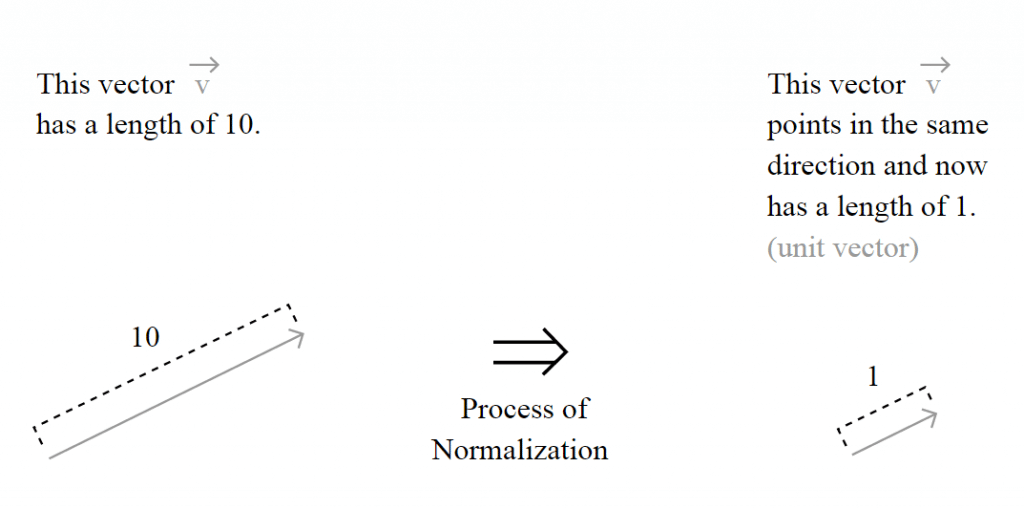
And the vector space representation using the non-normalized term-frequency of that document was:

\vec{v_{d_4}} = (0,2,1,0)

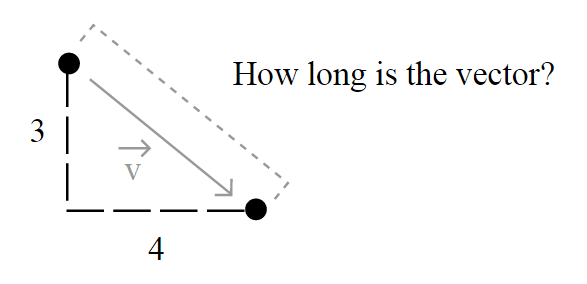
To normalize the vector, is the same as calculating the [Unit Vector](http://en.wikipedia.org/wiki/Unit_vector) of the vector, and they are denoted using the “hat” notation: \hat{v}. The definition of the unit vector \hat{v} of a vector \vec{v} is:

  \displaystyle \hat{v} = \frac{\vec{v}}{\|\vec{v}\|_p}  

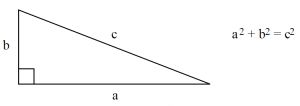
Where the \hat{v} is the unit vector, or the normalized vector, the \vec{v} is the vector going to be normalized and the \|\vec{v}\|_p is the norm (magnitude, length) of the vector \vec{v} in the L^p space. The unit vector is actually nothing more than a normalized version of the vector, is a vector for which the length is 1.



But the important question here is how the length of the vector is calculated and to understand this, you must understand the motivation of the L^p spaces, also called [Lebesgue spaces](http://en.wikipedia.org/wiki/Lp_space).



Usually, the length of a vector \vec{u} = (u_1, u_2, u_3, \ldots, u_n) is calculated using the [Euclidean norm](http://mathworld.wolfram.com/L2-Norm.html) – a norm is a function that assigns a strictly positive length or size to all vectors in a vector space -, which is defined by:



  \|\vec{u}\| = \sqrt{u^2_1 + u^2_2 + u^2_3 + \ldots + u^2_n}  

But this isn’t the only way to define length, and that’s why you see (sometimes) a number p together with the norm notation, like in \|\vec{u}\|_p. That’s because it could be generalized as:



and simplified as:

  \displaystyle \|\vec{u}\|_p = (\sum\limits_{i=1}^{n}\left|\vec{u}_i\right|^p)^\frac{1}{p}  

So when you read about a **L2-norm**, you’re reading about the [**Euclidean norm**](http://en.wikipedia.org/wiki/Norm_%28mathematics%29#Euclidean_norm), a norm with p=2, the most common norm used to measure the length of a vector, typically called “magnitude”; actually, when you have an unqualified length measure (without the pnumber), you have the **L2-norm** (Euclidean norm).

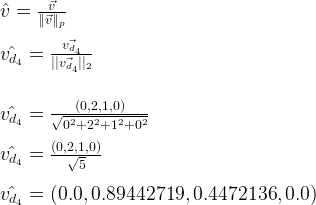
When you read about a**L1-norm**, you’re reading about the norm with p=1, defined as:

  \displaystyle \|\vec{u}\|_1 = ( \left|u_1\right| + \left|u_2\right| + \left|u_3\right| + \ldots + \left|u_n\right|)  

Which is nothing more than a simple sum of the components of the vector, also known as [Taxicab distance](http://en.wikipedia.org/wiki/Taxicab_geometry), also called Manhattan distance.

### Back to vector normalization

Now that you know what the vector normalization process is, we can try a concrete example, the process of using the L2-norm (we’ll use the right terms now) to normalize our vector \vec{v_{d_4}} = (0,2,1,0) in order to get its unit vector \hat{v_{d_4}}. To do that, we’ll simple plug it into the definition of the unit vector to evaluate it:



And that is it ! Our normalized vector \hat{v_{d_4}} has now a L2-norm \|\hat{v_{d_4}}\|_2 = 1.0.

**Note that here we have normalized our term frequency document vector, but later we’re going to do that after the calculation of the tf-idf.**

### The term frequency – inverse document frequency (tf-idf) weight

Now you have understood how the vector normalization works in theory and practice, let’s continue our tutorial. Suppose you have the following documents in your collection (taken from the first part of tutorial):

**Train Document Set:**

d1: The sky is blue.

d2: The sun is bright.

**Test Document Set:**

d3: The sun in the sky is bright.

d4: We can see the shining sun, the bright sun.

Your document space can be defined then as D = \{ d_1, d_2, \ldots, d_n \} where n is the number of documents in your corpus, and in our case as D_{train} = \{d_1, d_2\} and D_{test} = \{d_3, d_4\}. The cardinality of our document space is defined by \left|{D_{train}}\right| = 2 and \left|{D_{test}}\right| = 2, since we have only 2 two documents for training and testing, but they obviously don’t need to have the same cardinality.

Let’s see now, how idf (inverse document frequency) is then defined:

  \displaystyle \mathrm{idf}(t) = \log{\frac{\left|D\right|}{1+\left|\{d : t \in d\}\right|}}  

where \left|\{d : t \in d\}\right| is the **number of documents** where the term tappears, when the term-frequency function satisfies \mathrm{tf}(t,d) \neq 0, we’re only adding 1 into the formula to avoid zero-division.

The formula for the tf-idf is then:

  \mathrm{tf\mbox{-}idf}(t) = \mathrm{tf}(t, d) \times \mathrm{idf}(t)  

and this formula has an important consequence: a high weight of the tf-idf calculation is reached when you have a high term frequency (tf) in the given document (local parameter) and a low document frequency of the term in the whole collection (global parameter).

Now let’s calculate the idf for each feature present in the feature matrix with the term frequency we have calculated in the first tutorial:

  M_{train} =  \begin{bmatrix}  0 & 1 & 1 & 1\\  0 & 2 & 1 & 0  \end{bmatrix}  

Since we have 4 features, we have to calculate \mathrm{idf}(t_1), \mathrm{idf}(t_2), \mathrm{idf}(t_3), \mathrm{idf}(t_4):

  \mathrm{idf}(t_1) = \log{\frac{\left|D\right|}{1+\left|\{d : t_1 \in d\}\right|}} = \log{\frac{2}{1}} = 0.69314718  



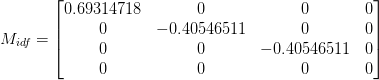


  \mathrm{idf}(t_4) = \log{\frac{\left|D\right|}{1+\left|\{d : t_4 \in d\}\right|}} = \log{\frac{2}{2}} = 0.0  

These idf weights can be represented by a vector as:

  \vec{idf_{train}} = (0.69314718, -0.40546511, -0.40546511, 0.0)  

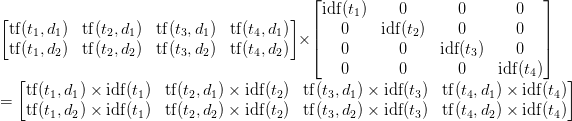
Now that we have our matrix with the term frequency (M_{train}) and the vector representing the idf for each feature of our matrix (\vec{idf_{train}}), we can calculate our tf-idf weights. What we have to do is a simple multiplication of each column of the matrix M_{train} with the respective \vec{idf_{train}} vector dimension. To do that, we can create a square [diagonal matrix](http://en.wikipedia.org/wiki/Diagonal_matrix) called M_{idf} with both the vertical and horizontal dimensions equal to the vector \vec{idf_{train}} dimension:



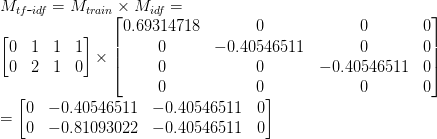
and then multiply it to the term frequency matrix, so the final result can be defined then as:

  M_{tf\mbox{-}idf} = M_{train} \times M_{idf}  

Please note that the matrix multiplication isn’t commutative, the result of A \times B will be different than the result of the B \times A, and this is why the M_{idf} is on the right side of the multiplication, to accomplish the desired effect of multiplying each idf value to its corresponding feature:



Let’s see now a concrete example of this multiplication:



And finally, we can apply our L2 normalization process to the M_{tf\mbox{-}idf}matrix. Please note that this normalization is **“row-wise”** because we’re going to handle each row of the matrix as a separated vector to be normalized, and not the matrix as a whole:

   M_{tf\mbox{-}idf} = \frac{M_{tf\mbox{-}idf}}{\|M_{tf\mbox{-}idf}\|_2}       = \begin{bmatrix}   0 & -0.70710678 & -0.70710678 & 0\\   0 & -0.89442719 & -0.4472136 & 0   \end{bmatrix}   

And that is our pretty normalized tf-idf weight of our testing document set, which is actually a collection of unit vectors. If you take the L2-norm of each row of the matrix, you’ll see that they all have a L2-norm of 1.

**In this section I’ll use Python to show each step of the tf-idf calculation using the [Scikit.learn](http://scikit-learn.sourceforge.net/" \o "Scikit.learn" \t "_blank) feature extraction module.**

The first step is to create our training and testing document set and computing the term frequency matrix:

from sklearn.feature\_extraction.text import CountVectorizer

train\_set = ("The sky is blue.", "The sun is bright.")

test\_set = ("The sun in the sky is bright.",

"We can see the shining sun, the bright sun.")

count\_vectorizer = CountVectorizer()

count\_vectorizer.fit\_transform(train\_set)

print "Vocabulary:", count\_vectorizer.vocabulary

Now that we have the frequency term matrix (called **freq\_term\_matrix**), we can instantiate the **TfidfTransformer**, which is going to be responsible to calculate the tf-idf weights for our term frequency matrix:

from sklearn.feature\_extraction.text import TfidfTransformer

tfidf = TfidfTransformer(norm="l2")

tfidf.fit(freq\_term\_matrix)

print "IDF:", tfidf.idf\_

# IDF: [ 0.69314718 -0.40546511 -0.40546511 0. ]

Note that I’ve specified the norm as L2, this is optional (actually the default is L2-norm), but I’ve added the parameter to make it explicit to you that it it’s going to use the L2-norm. Also note that you can see the calculated idf weight by accessing the internal attribute called **idf\_**. Now that **fit()** method has calculated the idf for the matrix, let’s transform the **freq\_term\_matrix** to the tf-idf weight matrix:

tf\_idf\_matrix = tfidf.transform(freq\_term\_matrix)

print tf\_idf\_matrix.todense()

# [[ 0. -0.70710678 -0.70710678 0. ]

# [ 0. -0.89442719 -0.4472136 0. ]]

And that is it, the **tf\_idf\_matrix** is actually our previous M_{tf\mbox{-}idf}matrix. You can accomplish the same effect by using the **Vectorizer**class of the Scikit.learn which is a vectorizer that automatically combines the **CountVectorizer** and the **TfidfTransformer** to you. See [this example](http://scikit-learn.sourceforge.net/stable/auto_examples/document_classification_20newsgroups.html#example-document-classification-20newsgroups-py) to know how to use it for the text classification process.