# 第6章

# 文本分类

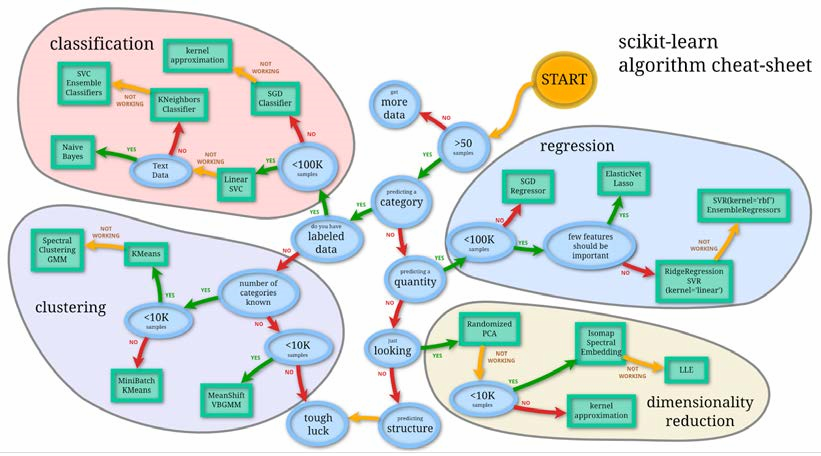
我们在上一章中对NLP领域中一些最常见的工具和预处理步骤进行了详细的讨论。在本章，我们将会利用之前所学习的大部分知识来构建一种复杂度最高的NLP应用。我们将会给出一个解决文本分类问题的通用方法，并带领您从零开始用尽可能简短的代码来构建一个文本分类器。除此之外，我们还将给出一份适用于文本分类问题的分类算法清单。

虽然我们会对部分最常见的文本算法进行一些讨论，但这些也都只是蜻蜓点水式的介绍，对于那些想要了解具体细节和相关数学北京的读者，我们也列出了许多在线资源和相关书籍，以供参考。我们将会尽力帮助读者了解他们所要了解的知识，使他们能够着手使用本章提供的代码片段。尽管，文本分类问题是NLP领域中一个很典型的用例，但在这里我们并不打算使用NLTK来做这件事，因为scikit-learn库中包含了更为广泛的分类算法，使用该库来执行文本挖掘会更为有效。

在阅读完本章之后，我们希望：

* 您应该学会所有的文本分类算法并理解它们。
* 您应该学会如何使用点对点的管道来构建文本分类器，并用scikit-learn和NLTK来实现它

下面我们来看一下scikit-learn库在机器学习方面的功能列表：



credit : scikit-learn

我们可以将上面这个功能列表当成一个流程图来走。这样一来我们就等于有了一个明确的方向，譬如知道哪种方法对应的是哪个问题、分类器之间的迁移依赖于多大规模的标记样本等。对于构建实用程序来说，从这张流程图入手是一个不错的开始，它在大多数情况下都是适用的。在本章，我们大多数时候关注的是文本数据，尽管scikit-learn库也可以处理其他类型的数据，但我们（为了降低维度）在这里将只探讨文本中的文本分类，文本聚类以及主题检测问题，并带您构建一些炫酷的NLP应用。当然，本章不会对机器学习、分类和聚类的概念进行详细说明，因为对于这些内容，我们可以在Web上找到充足的可用资料。我们会在谈到相关语料库时给出这些概念的更多细节，不过在此之前，我们先来做个复习。

## 机器学习

机器学习技术可以被分成两大类型：监督学习和无监督学习：

* **监督型学习**：该技术基于若干预先标记的历史样本，它用来预测未知测试样本的算法主要有以下两类：
  + **分类算法** ：该算法主要用于预测测试样本是否属于某些类型当中的一个。如果算法中只有两个类，这就是一个二元分类问题; 否则就是一个多元分类问题。
  + **回归算法**：该算法主要用于预测某种连续性的变量，例如房价和股票指数等。
* **无监督学习**：当我们没有任何标签数据却仍需要预测类标签时，我们就会用到这种被称之为无监督学习的技术。如果我们需要基于相关项之间的相似性来对它们进行分组，这就是在解决聚类问题。而如果我们是需要在较低维度上表示高维数据的话，那就更多的是一个降维问题。
* **半监督学习**：它在分类上应该属于监督学习任务和技术，但同时也会使用未标记的数据来进行训练。从名称上就可以看出，这更像是一种介于监督学习和无监督学习之间的技术，我们会基于少量标记数据和大量未标记数据来构建具有预测能力的机器学习模型。
* **增强学习**：这是一种利用奖罚机制来实现的机器学习形式，它没有指定的完成任务方式。

如果我们认为自己已经理解了这些不同的机器学习算法，就可以来猜猜看下面这些都属于哪种机器学习问题：

* 下个月的天气预报
* 从数百万笔交易中检测出欺诈行为
* Google的优先收件箱
* 亚马逊的推荐机制
* Google新闻
* 自动驾驶汽车

## 文本分类

对于文本分类，最简单的定义就是要基于文本内容来对该文本进行分类。通常情况下，目前所有的机器学习方法和算法都是根据数字/变量特征来编写的。所以这里最重要的问题之一，就是我们如何在语料库中用数字特征的形式来表示文本。各种技术文献提出了各种不同的转换方式，下面我们从最简单，使用最广泛使用的转换方式着手。

现在，为了帮助读者理解文本分类的具体过程，我们来看看垃圾邮件这个现实问题。 在WhatsApp和SMS的世界中，我们难免会收到许多垃圾邮件。下面，我们就来想想如何借助文本分类算法来解决垃圾邮件检测这个现实问题。 我们会用这个运行实例来贯穿本章的内容。首先我们要求读者手动标记一下这几则真实的SMS例文：

SMS001 ['spam', 'Had your mobile 11 months or more? U R entitled to

Update to the latest colour mobiles with camera for Free! Call The Mobile

Update Co FREE on 08002986030']

SMS002 ['ham', "I'm gonna be home soon and i don't want to talk about this stuff anymore tonight, k? I've cried enough today."]

|  |
| --- |
| 读者也可以从下面链接中下载到一份类似的已完成标注的数据集。当然，请确保您创建的是一个和上述例子显示内容相同的CSV文件。下面代码中'SMSSpamCollection'所对应的就是这个文件。  https://archive.ics.uci.edu/ml/datasets/ SMS+Spam+Collection |

我们现在要做的第一件事是按照之前几章所学到的数据清理、标识化处理以及词干提取等知识来对SMS进行清理，使其内容更简洁一些。 下面我们就来写一个基本的、用于文本清理的函数：

>>> import nltk

>>> from nltk.corpus import stopwords

>>> from nltk.stem import WordNetLemmatizer

>>> import csv

>>> def preprocessing(text):

>>> text = text.decode("utf8")

>>> # tokenize into words

>>> tokens = [word for sent in nltk.sent\_tokenize(text) for word in nltk.word\_tokenize(sent)]

>>> # remove stopwords

>>> stop = stopwords.words('english')

>>> tokens = [token for token in tokens if token not in stop]

>>> # remove words less than three letters

>>> tokens = [word for word in tokens if len(word) >= 3]

>>> # lower capitalization

>>> tokens = [word.lower() for word in tokens]

>>> # lemmatize

>>> lmtzr = WordNetLemmatizer()

>>> tokens = [lmtzr.lemmatize(word) for word in tokens]

>>> preprocessed\_text= ' '.join(tokens)

>>> return preprocessed\_text

我们在*第3章：词性标注*中已经讨论过了与标记化处理、词形还原以及停用词相关的知识。在上述代码中[[1]](#footnote-1)，我只是对SMS进行了解析并对其内容做了清理，获得了较为简洁的SMS文本。在接下来的几行代码中，我们将会创建两个列表，分别用以获取被清理之后的所有SMS内容以及类标签。用**ML（Machine learning）**术语来说就是获取所有的X和Y：

>>> smsdata = open('SMSSpamCollection') # check the structure of this file!

>>> smsdata\_data = []

>>> sms\_labels = []

>>> csv\_reader = csv.reader(sms,delimiter='\t')

>>> for line in csv\_reader:

>>> # adding the sms\_id

>>> sms\_labels.append( line[0])

>>> # adding the cleaned text We are calling preprocessing method

>>> sms\_data.append(preprocessing(line[1]))

>>> sms.close()

在继续任何下一步动作之前，我们要确保自己所用的系统中已经安装了scikit-learn库。

>>> import sklearn

|  |
| --- |
| 如果这句代码除了错，或者您在安装scikit的过程中遇到了一些错误，可以按照下面链接中的内容来安装scikit：  <http://scikit-learn.org/stable/install.html> |

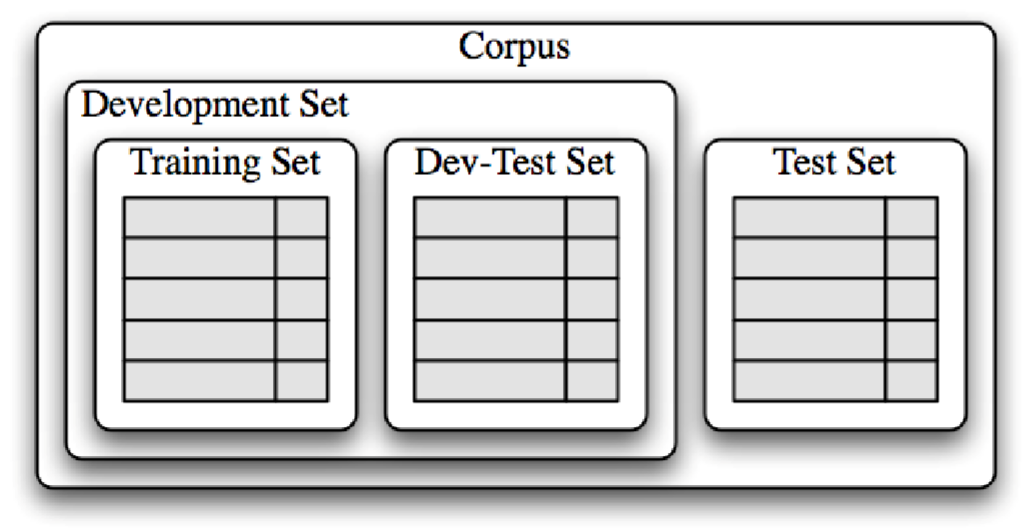
## 取样操作

Once we have the entire corpus in the form of lists, we need to perform some form of sampling. Typically, the way to sample the entire corpus in development train sets, dev-test sets, and test sets is similar to the sampling shown in the following figure.

一旦我们以列表的形式拥有整个语料库，我们需要执行某种形式的抽样。 通常，在开发训练集，开发测试集和测试集中对整个语料库进行抽样的方式与下图所示的抽样类似。

The idea behind the whole exercise is to avoid overfitting. If we feed all the data points to the model, then the algorithm will learn from the entire corpus, but the real test of these algorithms is to perform on unseen data. In very simplistic terms, if we are using the entire data in the model learning process the classifier will perform very good on this data, but it will not be robust. The reason being, we have to tune it to perform the best on the given data, but it doesn't learn how to deal with unknown data.

整个练习背后的想法是避免过度拟合。 如果我们将所有数据点馈送到模型，则算法将从整个语料库学习，但是这些算法的真实测试是对不可见数据执行。 在非常简单的术语中，如果我们在模型学习过程中使用整个数据，分类器将对此数据执行非常好，但它不会是鲁棒的。 原因是，我们必须调整它以对给定数据执行最佳，但它不会学习如何处理未知数据。



To solve this kind of a problem, the best way is to divide the entire corpus into two major sets. The development set and test set are kept away for the modeling exercise. We just use the dev set to build and tune the model. Once we are done with the entire modeling exercise, the results are projected based on the test set that we put aside. Now, if the model performs well on this set, we are sure that it's accurate and robust for any new data sample.

为了解决这种问题，最好的办法是将整个语料库分成两大类。 开发集和测试集被保留以用于建模操作。 我们只是使用dev设置来构建和调整模型。 一旦我们完成了整个建模练习，结果将基于我们放在一边的测试集进行预测。 现在，如果模型在这个集合上表现良好，我们确信它对任何新数据样本都是准确和稳健的。

Sampling itself is a very complicated and well-researched stream in the machine learning community, and it's a remedy for many data **skewness** and overfitting issues. For simplicity, will use the basic sampling, where we just divide the corpus into a split of 70:30:

抽样本身是一个非常复杂和经过深入研究的流在机器学习社区，它是一个补救措施许多数据偏差和过度拟合的问题。 为了简单起见，将使用基本抽样，其中我们只是将语料库分成70:30的分割：

>>> trainset\_size = int(round(len(sms\_data)\*0.70))

>>> # i chose this threshold for 70:30 train and test split.

>>> print 'The training set size for this classifier is ' + str(trainset\_ size) + '\n'

>>> x\_train = np.array([''.join(el) for el in sms\_data[0:trainset\_size]])

>>> y\_train = np.array([el for el in sms\_labels[0:trainset\_size]])

>>> x\_test = np.array([''.join(el) for el in sms\_data[trainset\_ size+1:len(sms\_data)]])

>>> y\_test = np.array([el for el in sms\_labels[trainset\_size+1:len(sms\_ labels)]])or el in sms\_labels[trainset\_size+1:len(sms\_labels)]])

>>> print x\_train

>>> print y\_train

* So what do you think will happen if we use the entire data as training data?
* What will happen when we have a very unbalanced sample?
* 那么，如果我们使用整个数据作为训练数据，你认为会发生什么？
* 当我们有一个非平衡样本时会发生什么？

|  |
| --- |
| To understand more about the available sampling techniques, go through  要了解更多有关可用的抽样技术，请访问  http://scikit-learn.org/stable/modules/classes.html#module-sklearn.cross\_validation. |

Let's jump to one of the most important things, where we transform the entire text into a vector form. The form is referred to as the **term-document matrix**. If we have to create a term-document matrix for the given example, it will look somewhat like this:

让我们跳到最重要的事情之一，我们将整个文本转换为向量形式。 形式被称为术语 - 文档矩阵。 如果我们必须为给定的示例创建一个term-document矩阵，它将看起来像这样：

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **TDM** | **Anymore** | **Call** | **camera** | **color** | **cried** | **enough** | **entitled** | **free** | **gon** | **had** | **latest** | **Mobile** |
| SMS1 | 0 | 1 | 1 | 1 | 0 | 0 | 1 | 2 | 0 | 1 | 0 | 3 |
| SMS2 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |

The representation here of the text document is also known as the **BOW** (**Bag of Word**) representation. This is one of the most commonly used representation in text mining and other applications. Essentially, we are not considering any context between the words to generate this kind of representation.

To generate a similar term-document matrix in Python, we use scikit vectorizers:

>>> from sklearn.feature\_extraction.text import CountVectorizer

>>> sms\_exp=[ ] >>>for line in sms\_list:

>>> sms\_exp.append(preprocessing(line[1]))

>>> vectorizer = CountVectorizer(min\_df=1)

>>> X\_exp = vectorizer.fit\_transform(sms\_exp)

>>> print "||".join(vectorizer.get\_feature\_names()) >>>print X\_exp.toarray() array([[

1, 0, 1, 1, 1, 0, 0, 1, 2, 0, 1, 0, 1, 3, 1, 0,

0, 0, 1, 0, 0, 2, 0, 0], [ 0, 1, 0, 0, 0, 1, 1, 0,

0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 1, 1, ]])

The count vectorizer is a good start, but there is an issue that you will face while using it: longer documents will have higher average count values than shorter documents, even though they might talk about the same topics.

|  |
| --- |
| To avoid these potential discrepancies, it suffices to divide the number of occurrences of each word in a document by the total number of words in the document. This new feature is called tf (Term frequencies). |

Another refinement on top of tf is to downscale weights for words that occur in many documents in the corpus, and are therefore less informative than those that occur only in a smaller portion of the corpus.

This downscaling is called **tf–idf** (**term frequency–inverse document frequency**). Fortunately, scikit also provides a way to achieve the following:

>>>from sklearn.feature\_extraction.text import TfidfVectorizer

>>>vectorizer = TfidfVectorizer(min\_df=2, ngram\_range=(1, 2), stop\_ words='english', strip\_accents='unicode', norm='l2')

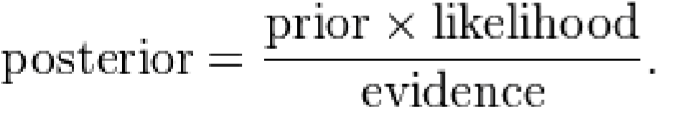
>>>X\_train = vectorizer.fit\_transform(x\_train)

>>>X\_test = vectorizer.transform(x\_test)

We now have the text in a matrix format the same as we have in any machine learning exercise. Now, X\_train and X\_test can be used for classification using any machine learning algorithm. Let's talk about some of the most commonly used machine learning algorithms in context of text classification.

### 朴素贝叶斯方法

Let's build your first text classifier. Let's start with a Naive Bayes classifier. Naive Bayes relies on the Bayes algorithm and essentially, is a model of assigning a class label to the sample based on the conditional probability class given by features/attributes. Here we deal with frequencies/bernoulli to estimate prior and posterior probabilities.



The naive assumption here is that all features are independent of each other, which looks counter intuitive in the case of text. However, surprisingly, Naive Bayes performs quite well in most of the real-world use cases.

Another great thing about NB is that it's too simple and very easy to implement and score. We need to store the frequencies and calculate the probabilities. It's really fast in case of training as well as test (scoring). For all these reasons, in most of the cases of text classification, it serves as a benchmark.

Let's write some code to achieve this classifier:

>>>from sklearn.naive\_bayes import MultinomialNB

>>>clf = MultinomialNB().fit(X\_train, y\_train)

>>>y\_nb\_predicted = clf.predict(X\_test)

>>>print y\_nb\_predicted

>>>print ' \n confusion\_matrix \n '

>>>cm = confusion\_matrix(y\_test, y\_pred)

>>>print cm

>>>print '\n Here is the classification report:'

>>>print classification\_report(y\_test, y\_nb\_predicted) confusion\_matrix [[1205 5] [26 156]]

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  | | --- | | Classified | | |  | | --- | | A c t u a l s | | |  |  |  | | --- | --- | --- | | True Positive |  | False Negarive | | | | |  |  |  | | --- | --- | --- | | False Positive |  | True Negarive | | | |

The way to read the confusion matrix is that from all the 1,392 samples in the test set, there were 1205 true positives and 156 true negative cases. However, we also predicted 5 false negatives and 26 false positives. There are different ways of measuring a typical binary classification.

We have given definitions of some of the most common measures used in classification measures:

Accuracy = *tp*+*tn tp*+ +*tn fp*+ *fn*

Precision = *tp tp*+ *fp*

Recall = *tp tp*+ *fn*

*F* = 2⋅ precision⋅recall precision+recall

Here is the classification report:

Precision recall f1-score support ham 0.97 1.00 0.98 1210 spam 1.00 0.77 0.87 182 avg / total 0.97 0.97 0.97 1392

With the preceding definition, we can now understand the results clearly. So, effectively, all the preceding metrics look good, which means that our classifier is performing accurately, and is robust. I would highly recommend that you look into the module metrics for more options to analyze the results of the classifier. The most important and balanced metric is the f1 measure (which is nothing but the harmonic mean of precision and recall), which is used widely because it gives a better picture of the coverage and the quality of the classification algorithms. Accuracy intuitively tells us how many true samples have been covered from all the samples. Precision and recall both have significance, while precision talks about how many true positives it got and what else got covered, hand recall gives us details about how accurate we are from the pool of true positives and false negatives.

For more information on various scikit classes visit the following link: [http://scikit-learn.org/stable/modules/classes. html#module-sklearn.metrics](http://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics)

The other more important process we follow to understand our model is to really look deep into the model by looking at the actual features that contribute to the positive and negative classes. I just wrote a very small snippet to generate the top *n* features and print them. Let's have a look at them:

>>>feature\_names = vectorizer.get\_feature\_names()

>>>coefs = clf.coef\_

>>>intercept = clf.intercept\_

>>>coefs\_with\_fns = sorted(zip(clf.coef\_[0], feature\_names))

>>>n = 10

>>>top = zip(coefs\_with\_fns[:n], coefs\_with\_fns[:-(n + 1):-1]) >>>for (coef\_1, fn\_1), (coef\_2, fn\_2) in top:

>>> print('\t%.4f\t%-15s\t\t%.4f\t%-15s' % (coef\_1, fn\_1, coef\_2, fn\_2))

-9.1602 10 den -6.0396 free

-9.1602 15 -6.3487 txt

-9.1602 1hr -6.5067 text

-9.1602 1st ur -6.5393 claim

-9.1602 2go -6.5681 reply

-9.1602 2marrow -6.5808 mobile

-9.1602 2morrow -6.5858 stop

-9.1602 2mrw -6.6124 ur

-9.1602 2nd innings -6.6245 prize

-9.1602 2nd ur -6.7856 www

In the preceding code, I just read all the feature names from the vectorizer, got the coefficients related to the given feature, and then printed the first-10 features. If you want more features, just modify the value of *n*. If we look closely just at the features, we get a lot of information about the model as well as more suggestions about our feature selection and other parameters, such as preprocessing, unigrams/bigrams, stemming, tokenizations, and so on. For example, if you look at the top features of ham you can see that 2morrow, 2nd innings, and some of the digits are coming very significantly. We can see on the positive class (spam ) term "free" comes out a very significant term which is intuitive while many spam messages will be about some free offers and deal. Some of the other terms to note are prize, www, claim.

|  |
| --- |
| For more details, refer to http://scikitlearn.org/stable/ modules/naive\_bayes.html. |

### 决策树

Decision trees are one of the oldest predictive modeling techniques, where for the given features and target, the algorithm tries to build a logic tree. There are multiple algorithms that exist for decision trees. One of the most famous and widely used algorithm is **CART**.

CART constructs binary trees using this feature, and constructs a threshold that yields the large amount of information from each node. Let's write the code to get a CART classifier:

>>>from sklearn import tree

>>>clf = tree.DecisionTreeClassifier().fit(X\_train.toarray(), y\_train)

>>>y\_tree\_predicted = clf.predict(X\_test.toarray())

>>>print y\_tree\_predicted

>>>print ' \n Here is the classification report:'

>>>print classification\_report(y\_test, y\_tree\_predicted)

The only difference is in the input format of the training set. We need to modify the sparse matrix format to a **NumPy** array because the scikit tree module takes only a NumPy array.

Generally, trees are good when the number of features are very less. So, although our results look good here, people hardly use trees in text classification. On the other hand, trees have some really positive sides to them. It is still one the most intuitive algorithms and is very easy to explain and implement. There are many implementations of tree-based algorithms, such as ID3, C4.5, and C5. scikit-learn uses an optimized version of the CART algorithm.

### 随机梯度下降

**Stochastic gradient descent** (**SGD**) is a simple, yet very efficient approach that fits linear models. It is particularly useful when the number of samples (and the number of features) is very large. If you follow the cheat sheet, you will find SGD to be the one-stop solution for many text classification problems. Since it also takes care of regularization and provides different losses, it turns out to be a great choice when experimenting with linear models.

SGD, also known as **Maximum entropy** (**MaxEnt**), provides functionality to fit linear models for classification and regression using different (convex) loss functions and penalties. For example, with loss = log, fits a logistic regression model, while with loss = hinge, it fits a linear support vector machine (SVM).

An example of SGD is as follows:

>>>from sklearn.linear\_model import SGDClassifier

>>>from sklearn.metrics import confusion\_matrix

>>>clf = SGDClassifier(alpha=.0001, n\_iter=50).fit(X\_train, y\_train)

>>>y\_pred = clf.predict(X\_test)

>>>print '\n Here is the classification report:'

>>>print classification\_report(y\_test, y\_pred)

>>>print ' \n confusion\_matrix \n '

>>>cm = confusion\_matrix(y\_test, y\_pred)

>>>print cm

Here is the classification report:

precision recall f1-score support

ham 0.99 1.00 0.99 1210

spam 0.96 0.91 0.93 182

avg / total 0.98 **0.98 0.98 1392**

Most informative features:

-1.0002 sir 2.3815 ringtoneking

-0.5239 bed 2.0481 filthy

-0.4763 said 1.8576 service

-0.4763 happy 1.7623 story

-0.4763 might 1.6671 txt

-0.4287 added 1.5242 new

-0.4287 list 1.4765 ringtone

-0.4287 morning 1.3813 reply

-0.4287 always 1.3337 message

-0.4287 and 1.2860 call

-0.4287 plz 1.2384 chat

-0.3810 people 1.1908 text

-0.3810 actually 1.1908 real

-0.3810 urgnt 1.1431 video

### 逻辑回归

Logistic regression is a linear model for classification. It's also known in the literature as logit regression, maximum-entropy classification (MaxEnt), or the log-linear classifier. In this model, the probabilities describing the possible outcomes of a single trial are modeled using a logit function.

As an optimization problem, the L2 binary class' penalized logistic regression minimizes the following cost function:

#### *min w*

*w c*,

Similarly, L1 the binary class' regularized logistic regression solves the following optimization problem:

*miw c*, *n* 12 *w* 1 +*C*∑*i*=*n*1 log e( xp(−*yi* (*X wiT* + +*c*)) 1)

### 支持向量机

**Support vector machines** (**SVM**) is currently the-state-of-art algorithm in the field of machine learning.

SVM is a non-probabilistic classifier. SVM constructs a set of hyperplanes in an infinite-dimensional space, which can be used for classification, regression, or other tasks. Intuitively, a good separation is achieved by a hyperplane that has the largest distance to the nearest training data point of any class (the so-called functional margin), since in general, the larger the margin, the lower the size of classifier.

Let's build one of the most sophisticated supervised learning algorithms with scikit:

>>>from sklearn.svm import LinearSVC

>>>svm\_classifier = LinearSVC().fit(X\_train, y\_train)

>>>y\_svm\_predicted = svm\_classifier.predict(X\_test)

>>>print '\n Here is the classification report:'

>>>print classification\_report(y\_test, y\_svm\_predicted)

>>>cm = confusion\_matrix(y\_test, y\_pred)

>>>print cm

Here is the classification report for the same:

precision recall f1-score support

ham 0.99 1.00 0.99 1210

spam 0.97 0.90 0.93 182

avg / total 0.98 0.98 0.98 1392

confusion\_matrix [[1204 6] [ 17 165]]

The most informative features:

-0.9657 road 2.3724 txt

-0.7493 mail 2.0720 claim

-0.6701 morning 2.0451 service

-0.6691 home 2.0008 uk

-0.6191 executive 1.7909 150p

-0.5984 said 1.7374 www

-0.5978 lol 1.6997 mobile

-0.5876 kate 1.6736 50

-0.5754 got 1.5882 ringtone

-0.5642 darlin 1.5629 video

-0.5613 fullonsms 1.4816 tone

-0.5613 fullonsms com 1.4237 prize

These are definitely the best results so far from all the supervised algorithms we have tried. Now with this, I will stop with supervised classifiers. There are millions of books available related to the different machine learning algorithms; even for individual algorithms, there are many books that are available for you. I would highly recommend you to have a deep understanding of any of the preceding algorithms before you use them for any of the real-world applications.

## 随机森林算法

A random forest is an ensemble classifier that estimates based on the combination of different decision trees. Effectively, it fits a number of decision tree classifiers on various subsamples of the dataset. Also, each tree in the forest built on a random best subset of features. Finally, the act of enabling these trees gives us the best subset of features among all the random subsets of features. Random forest is currently one of best performing algorithms for many classification problems.

An example of Random forest is as follows:

>>>from sklearn.ensemble import RandomForestClassifier

>>>RF\_clf = RandomForestClassifier(n\_estimators=10)

>>>predicted = RF\_clf.predict(X\_test)

>>>print '\n Here is the classification report:'

>>>print classification\_report(y\_test, predicted)

>>>cm = confusion\_matrix(y\_test, y\_pred)

>>>print cm

|  |
| --- |
| People who still want to work with NLTK for text classification. Please go through the following link:  <http://www.nltk.org/howto/classify.html> |

## 文本聚类

The other family of problems that can come with text is unsupervised classification. One of the most common problem statements you can get is "I have these millions of documents (unstructured data). Is there a way I can group them into some meaningful categories?". Now, once you have some samples of tagged data, we could build a supervised algorithm that we talked about, but here, we need to use an unsupervised way of grouping text documents.

Text clustering is one of the most common ways of unsupervised grouping, also known as, clustering. There are a variety of algorithms available using clustering. I mostly used **k-means** or **hierarchical** clustering. I will talk about both of them and how to use them with a text corpus.

### K-means

Very intuitively, as the name suggest, we are trying to find k groups around the mean of the data points. So, the algorithm starts with picking up some random data points as the centroid of all the data points. Then, the algorithm assigns all the data points to it's nearest centroid. Once this iteration is done, recalculation of the centroid happens and these iterations continue until we reach a state where the centroids don't change (algorithm saturate).

There is a variant of the algorithm that uses mini batches to reduce the computation time, while still attempting to optimize the same objective function.

Mini batches are subsets of the input data randomly sampled in each training iteration. These options should always be tried once your dataset is really huge and you want less training time.

An example of K-means is as follows:

>>>from sklearn.cluster import KMeans, MiniBatchKMeans

>>>true\_k=5

>>>km = KMeans(n\_clusters=true\_k, init='k-means++', max\_iter=100, n\_ init=1)

>>>kmini = MiniBatchKMeans(n\_clusters=true\_k, init='k-means++', n\_init=1, init\_size=1000, batch\_size=1000, verbose=opts.verbose)

>>># we are using the same test,train data in TFIDF form as we did in text classification >>>km\_model=km.fit(X\_train)

>>>kmini\_model=kmini.fit(X\_train)

>>>print "For K-mean clustering "

>>>clustering = collections.defaultdict(list) >>>for idx, label in enumerate(km\_model.labels\_):

>>> clustering[label].append(idx)

>>>print "For K-mean Mini batch clustering "

>>>clustering = collections.defaultdict(list) >>>for idx, label in enumerate(kmini\_model.labels\_):

>>> clustering[label].append(idx)

In the preceding code, we just imported scikit-learn's kmeans / minibatchkmeans and fitted the same training data that we were using in the running examples. We can also print a cluster for each sample using the last three lines of the code.

## 文本中的主题建模

The other famous problem in the context of the text corpus is finding the topics of the given document. The concept of topic modeling can be addressed in many different ways. We typically use **LDA** (**Latent Dirichlet allocation**) and LSI (Latent semantic indexing) to apply topic modeling text documents.

Typically, in most of the industries, we have huge volumes of unlabeled text documents. In case of an unlabeled corpus to get the initial insights of the corpus, a topic model is a great option, as it not only gives us topics of relevance, but also categorizes the entire corpus into number of topics given to the algorithm.

We will use a new Python library "gensim" that implements these algorithms for us. So, let's jump to the implementation of LDA and LSI for the same running SMS dataset. Now, the only change to the problem is that we want to model different topics in the SMS data and also want to know which document belongs to which topic. A better and more realistic use case could be to run topic modeling on the entire Wikipedia dump to find different kinds of topics that have been discussed there, or to run topic modeling on billions of reviews/complaints from customers to get an insight of the topics that people discuss.

### Installing gensim

One of the easiest ways to install gensim is using a package manager:

>>>easy\_install -U gensim

Otherwise, you can install it using:

>>>pip install gensim

Once you're done with the installation, run the following command:

>>>import gensim

If there is any error, go to

<https://radimrehurek.com/gensim/install.html>.

Now, let's look at the following code:

>>>from gensim import corpora, models, similarities

>>>from itertools import chain

>>>import nltk

>>>from nltk.corpus import stopwords

>>>from operator import itemgetter

>>>import re

>>>documents = [document for document in sms\_data]

>>>stoplist = stopwords.words('english')

>>>texts = [[word for word in document.lower().split() if word not in stoplist] \ for document in documents]

We are just reading the document in our SMS data and removing the stop words. We could use the same method that we did in the previous chapters to do this. Here, we are using a library-specific way of doing things.

Gensim has all the typical NLP features as well provides some great way to create different corpus formats, such as TFIDF, libsvm, market matrix. It also provides conversion of one to another.

In the following code, we are converting the list of documents to a BOW model and then, to a typical **TF-IDF** corpus:

>>>dictionary = corpora.Dictionary(texts)

>>>corpus = [dictionary.doc2bow(text) for text in texts]

>>>tfidf = models.TfidfModel(corpus)

>>>corpus\_tfidf = tfidf[corpus]

Once you have a corpus in the required format, we have the following two methods, where given the number of topics, the model tries to take all the documents from the corpus to build a LDA/LSI model:

>>>si = models.LsiModel(corpus\_tfidf, id2word=dictionary, num\_topics=100)

>>>#lsi.print\_topics(20)

>>>n\_topics = 5

>>>lda = models.LdaModel(corpus\_tfidf, id2word=dictionary, num\_topics=n\_

topics)

Once the model is built, we need to understand the different topics, what kind of terms represent that topic, and we need to print some top terms related to that topic:

>>>for i in range(0, n\_topics):

>>> temp = lda.show\_topic(i, 10)

>>> terms = [] >>> for term in temp:

>>> terms.append(term[1])

>>> print "Top 10 terms for topic #" + str(i) + ": "+ ",

".join(terms)

Top 10 terms for topic #0: week, coming, get, great, call, good, day, txt, like, wish

Top 10 terms for topic #1: call, ..., later, sorry, 'll, lor, home, min, free, meeting

Top 10 terms for topic #2: ..., n't, time, got, come, want, get, wat, need, anything

Top 10 terms for topic #3: get, tomorrow, way, call, pls, 're, send, pick, ..., text

Top 10 terms for topic #4: ..., good, going, day, know, love, call, yup, get, make

Now, if you look at the output, we have five different topics with clearly different intent. Think about the same exercise for Wikipedia or a huge corpus of web pages, and you will get some meaningful topics that represent the corpus.

## 参考资料

* http://scikit-learn.org/
* https://radimrehurek.com/gensim/
* https://en.wikipedia.org/wiki/Document\_classification

## Summary

The idea behind this chapter was to introduce you to the world of text mining. We want to give you a basic introduction to some of the most common algorithms available with text classification and clustering .We know how some of these concept will help you to build really great NLP applications, such as spam filters, domain centric news feeds, web page taxonomy, and so on. Though we have not used NLTK to classify the module in our code snippets, we used NLTK for all the preprocessing steps. We highly recommend you to use scikit-learn over NLTK for any classification problem. In this chapter, we started with machine learning and the types of problems that it can address. We discussed some of the specifics of ML problems in the context of text. We talked about some of the most common classification algorithms that are used for text classification, clustering, and topic modeling. We also give you enough implementation details to get the job done. I still think you need to read a lot about each and every algorithm separately to understand the theory and to gain in-depth understanding of them.

We also provided you an entire pipeline of the process that you need to follow in case of any text mining problem. We covered most of the practical aspects of machine learning, such as sampling, preprocessing, model building, and model evaluation.

The next chapter will also not be directly related to NLTK/NLP, but it will be a great tool for a data scientist/NLP enthusiast. In most of NLP problems, we deal with unstructured text data, and the Web is one of the richest and biggest data sources available for this. Let's learn how to gather data from the Web and how to efficiently use it to build some amazing NLP applications.

1. 译者注：此处的原文是In the following code，但从文本实际布局来看，这里指的应该是这段文字上面的代码。疑为作者笔误。 [↑](#footnote-ref-1)