Automatic Detection of Character Encoding and Language

Seungbeom Kim, Jongsoo Park
{sbkim,jongsoo}@stanford.edu

CS 229, Machine Learning
Autumn 2007
Stanford University

1 Introduction

The Internet is full of textual contents in various languages and character encodings, and their communication across the linguistic borders is ever increasing. Since the different encodings are not compatible with one another, communication protocols such as HTTP, HTML, and MIME are equipped with mechanisms to tag the character encoding (a.k.a. *charset*) used in the delivered content. However, as native speakers of a language whose character set does not fit in US-ASCII, we have encountered a lot of web pages and e-mail messages that are encoded in or labeled with a wrong character encoding, which is often annoying or frustrating. Many authors, especially of e-mail messages, are not aware of the encoding issues, and their carelessness in choosing correct encodings can result in messages that are illegible to the recipient. Therefore, automatically detecting the correct character encoding from the given text can serve many people using various character encodings, including their native one, and has a good practical value.

The importance of automatic charset detection is not restricted to web browsers or e-mail clients; detecting the charset is the first step of text processing. Therefore, many text processing applications should have automatic charset detection as their crucial component; web crawlers are a good example.

Due to its importance, automatic charset detection is already implemented in major Internet applications such as Mozilla or Internet Explorer. They are very accurate and fast, but the implementation applies many domain specific knowledges in case-by-case basis. As opposed to their methods, we aimed at a simple algorithm which can be uniformly applied to every charset, and the algorithm is based on well-established, standard machine learning techniques. We also studied the relationship between language and charset detection, and compared byte-based algorithms and character-based algorithms. We used Naïve Bayes (NB) and Support Vector Machine (SVM).

Using the documents downloaded from Wikipedia [6], we evaluated different combinations of algorithms and compared them with the universal charset detector in Mozilla. We found two promising algorithms. The first one is a simple SVM whose feature is the frequency of byte values. The algorithm is uniform and very easy to implement. It also only needs maximum 256 table entries per each charset and the detection time is much shorter than other algorithms. Despite of its simplicity, it achieves 98.22% accuracy, which is comparable to that of Mozilla (99.46%). The second one is a character-based NB whose accuracy is 99.39%. It needs a larger table size and a longer detection time than the first algorithm, but it also detects the language of document as a byproduct.

2 Previous Work - Mozilla

Kikui [2] proposed a simple statistical detection algorithm. He used different algorithms for multi-byte and single-byte charsets: a character unigram for multi-byte charsets and a word unigram for single-byte charsets. (N-gram is the sequence of linguistic units with length N. A word unigram means that the unit of statistics is one word.) Using different algorithms for multi-byte and

single-byte charsets is a fairly common approach shared by almost all previous works. This is because multi-grams are too long for multi-byte charsets and unigrams are too short to gather meaningful statistics of single-byte charsets. Kikui's algorithm is essentially NB except that it has a pre-step which distinguishes between multi-byte charsets and single-byte charsets.

Russell and Lapalme [5] advocated the simultaneous detection of charset and language. Their probability model is also NB, but they interpolated trigram and unigram models.

There are not many academic papers about automatic charset detection, and they are neither theoretically sound nor supported by a thorough empirical evaluation. At the same time, open source implementations such as Mozilla [4] and ICU [1] work extremely well in practice. Since their approaches are similar, we focus on the universal charset detector in Mozilla.

The universal charset detector in Mozilla [4] uses three different methods in a complementary manner (Figure 1). It first classifies the document by finding a special byte sequence. For example, every UTF series character stream can start with a special byte sequence byte-order mark (BOM). If the input byte sequence has an escape character `'\033'` (ESC) or `'~{'`, we know that the input is encoded by one of the escape charsets (GB2321, HZ Simplified Chinese, ISO-2022-CN/JP/KR). If the input satisfies this escape character condition, the detector executes a state machine for each possible charset and pick the one which arrives at the "itsMe" state first. If the input neither satisfies the conditions above and nor has any byte bigger than `0x7f`, we know that the input is US-ASCII.

The second and third methods use a known statistical distribution of each charset. Similar to Kikui [2], the universal charset detector in Mozilla uses different methods for multi-byte charsets and single-byte charsets: one character as the unit for multi-byte charsets and two bytes as the unit for single-byte charsets. However, to fully utilize the encoding rules for each charsets, it also runs a state machine. The state machine identifies the charset which is surely correct or which is surely incorrect, according to the encoding rules. For single-byte charsets whose language does not use Latin characters, it excludes Latin characters from the statistics to reduce the noise.

In summary, Mozilla's implementation smartly exploits various aspects of character encoding rules which are accumulated from years of experience. However, the implementation is not easy to understand by novices.

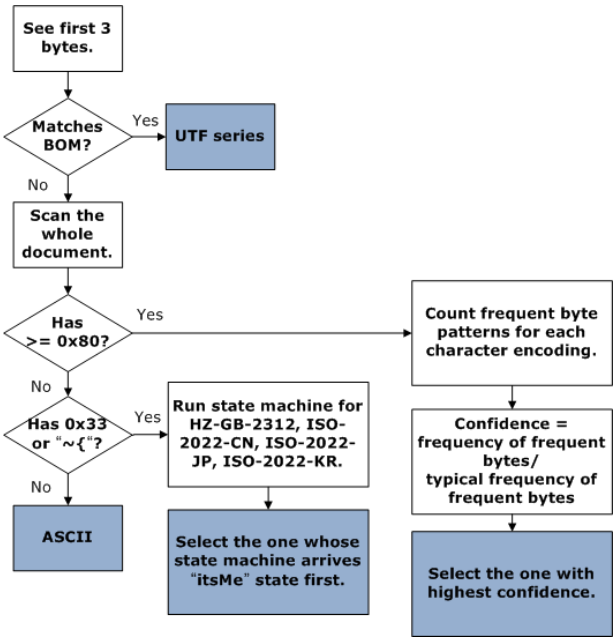

Figure 1: The flow chart of the universal character encoding detector in Mozilla. BOM means Byte-order mark.

3 Design

3.1 Scope

For the purpose of this project, we want to confine the set of target languages and charsets. To be representative among single-byte and multi-byte encodings, we select a few from both classes as follows:

- single-byte:
  - US-ASCII [en]
  - ISO-8859-1 [en,fr]
- multi-byte:
  - Shift-JIS, EUC-JP [ja]
  - EUC-KR [ko]

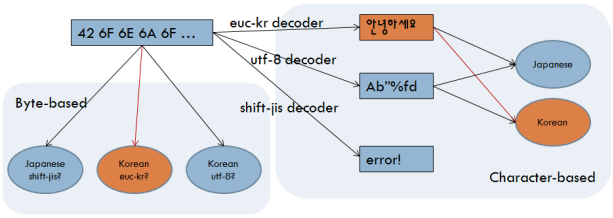

Figure 2: Byte-based method vs. character-based method

- ISO-2022-{JP,KR} [ja,ko]
- UTF-8 [universal]

This yields 12 pairs of (language, charset).

It is a trivial task to add more charsets and languages as needed in the future. In particular, our algorithm does not depend on any previous knowledge of any particular language or charset, except the fact that US-ASCII is the common subset of the other charsets, notably ISO-8859-1 and UTF-8.

3.2 Feature

We define our feature as a sequence of "units." Then the two axes of the feature are formed by the definition of the "unit" and the length of the sequence.

We have two different approaches regarding the definition of the "unit": one is *byte-based*, which means treating the raw bytes from the document as the units, and the other *character-based*, which assumes each charset and decodes the byte sequence into a sequence of characters (represented by the code points of the Universal Character Set) to be considered as the units. (When the byte sequence cannot be decoded under the assumption of a charset, that charset is dropped from the set of candidates. See Figure 2) The latter is inspired by our intuition that it will give a better representation of the character distribution in each language than the former, in which byte sequences spanning the boundary between adjacent characters will perturb the distribution. The character-based algorithm only needs to keep one parameter set for each language, while the byte-based algorithm needs one for each (language, charset) pair.

The other axis is the sequence length, represented by N (in N-grams). A simple model can use individual bytes or characters (unigrams) as the features; more elaborate models can use two or three consecutive bytes or characters (bigrams, trigrams), or even more. The size of the parameter space is exponential to N, so there is a tradeoff between the accuracy of the model and the cost of training, computation and storage. This becomes more important as the size of the set of units becomes large, as in Asian charsets with character-based approach.

3.3 Algorithm

Our first choice was Naïve Bayes (NB) under the multinomial event model, which is similar to the spam filtering example discussed in class. This algorithm is simple to implement and expected to work reasonably well.

Later, Support Vector Machine (SVM) was also tried. Our own implementation of the SMO algorithm was used first, but a library implementation (called `libsvm` [3]) outperformed our version. The lengths of the documents were normalized to 1, thus, for example, the value at the feature vector index `0x68` represents the number of appearances of `0x68` divided by the total number of bytes of the document. Since the feature space is already large, we used the linear kernel. We used "one-vs-the-rest" multiclass strategy: the class with the highest decision values was selected. We also tuned the regularization constant C and the tolerance value.

Based on the result from these two algorithms, we devised a hybrid algorithm, which first uses the character-based NB to detect only the charset and then uses the character-based SVM to further increase the accuracy of the language detection.

4 Result

1000 documents per (language, charset) pair have been downloaded from Wikipedia [6]. (Since Wikipedia is in UTF-8, the documents for other charsets can be converted from UTF-8 by using `iconv`.) They are used for training and testing, using a cross-validation method that uses 70% of the data for training and the rest 30% for testing.

Figure 3 shows the result for the best cases for each detection goal and algorithm. For detecting the charset only, the Mozilla implementation gives the best accuracy of 99.46%, but our SVM method with byte unigrams

achieves a similar accuracy of 98.22% with a much less table size of 4.1 KB and a much less detection time of 0.08 ms per document.

For detecting the charset and the language at the same time, our hybrid method combining unigram NB and unigram SVM gives the best pair accuracy of 98.69% with the table size of 1.6 MB and the detection time of 2.23 ms per document.

Figure 4 shows the histogram of the correct charsets vs. the detected charsets when we use the byte-based SVM to detect the charsets only. Note that the majority of the errors are from detecting UTF-8 or ISO-8859-1 documents as US-ASCII. Since US-ASCII is the subset of all the other character sets and many documents with (English, UTF-8) and (English, ISO-8859-1) mainly use US-ASCII except for very few characters, it is hard to distinguish between them merely by using statistical information. This is the place where the domain-specific knowledge plays an important role and the universal charset detector in Mozilla uses them very well.

The character-based approach does a better job of distinguishing between US-ASCII English documents and non-US-ASCII English documents. If the document is US-ASCII, then UTF-8, ISO-8859-1, and US-ASCII decoders should produce the same character sequence, and thus give the same likelihoods. By giving the precedence on the subset charset (US-ASCII in our case), we can correctly detect a US-ASCII document. (Note, however, that detecting a US-ASCII document as ISO-8859-1 or UTF-8 is not harmful at all and thus can be considered as a non-error.) If the document is almost US-ASCII but has exceptional few characters, then the US-ASCII decoder fails, thus it is detected as another charset.

We expected that the character-based SVM would work very well, because both of the character-based NB and the byte-based SVM gave a better result than the byte-based NB. However, the accuracy dropped from that of the byte-based SVM. This may share the common reason with the fact that the SVM with byte bigrams has a lower accuracy than the SVM with byte unigrams. Finding the reason of the decreased accuracy of the SVM algorithm in a larger feature space could be an interesting future work.

5 Conclusion

We have seen that:

1. the character-based approach with the Naïve Bayes algorithm works well for detection between charsets sharing a large code space (e.g. English US-ASCII, UTF-8, and ISO-8859-1 documents),

2. the byte-based approach with the Support Vector Machine works well with a smaller memory usage and at a faster speed, and

3. the hybrid method of character-based NB for charset detection and character-based SVM for language detection gives the best results.

References

[1] International components for unicode. http://www.icu-project.org.

[2] G. Kikui. Identifying the coding system and language of online documents on the internet. In *International Conference on Computational Linguistics*, 1996.

[3] A library for support vector machines. http://www.csie.ntu.edu.tw/ cjlin/libsvm/.

[4] A composite approach to language/encoding detection. http://www.mozilla.org/projects/intl/UniversalCharset Detection.html, 2002.

[5] G. Russell and G. Lapalme. Automatic identification of language and encoding. 2003.

[6] Wikipedia. http://www.wikipedia.org/.

algorithm			optimal meta-parameters	pair accuracy	charset accuracy	table size	detection time (per document)
charset only	NB	byte	2gram		92.35%	30 KB	3.04 ms
	SVM	byte	1gram, tol=1e-5, C=1e25		98.22%	**4.1 KB**	**0.08 ms**
	Mozilla				**99.46%**	14 KB	1.67 ms
charset & lang.	NB	byte	3gram	81.78%	82.31%	3.4 MB	25.80 ms
		char	3gram	98.01%	99.33%	2.4 MB	8.33 ms
	SVM	byte	1gram, tol=1e-5, C=1e9	90.74%	90.94%	6.1 KB	0.53 ms
		char	1gram, tol=1e-4, C=1e25	89.60%	90.19%	1.6 MB	2.02 ms
	hybrid		1gram NB + 1gram SVM	**98.69%**	99.33%	1.6 MB	2.23 ms

Figure 3: Accuracy, required table size and detection time of each algorithm. Pair accuracy is counted only if the both of charset and language are correct. The best meta-parameters such as N, C and tolerance are chosen for each algorithm.

		utf-8	us-ascii	iso-8859-1	shift-jis	euc-jp	iso-2022-jp	euc-kr	iso-2022-kr
correct charset	utf-8	293	**5**	1	0	0	0	0	0
	us-ascii	1	298	1	0	0	0	0	0
	iso-8859-1	0	**20**	176	0	0	0	0	0
	shift-jis	0	0	0	300	0	0	0	0
	euc-jp	0	0	0	0	300	0	0	0
	iso-2022-jp	0	1	1	0	0	298	0	0
	euc-kr	1	0	0	0	1	0	298	0
	iso-2022-kr	0	0	0	0	0	0	0	300

Figure 4: The histogram of correct charsets vs. detected charsets from byte-based SVM (charset detection only)