Error-Correcting Codes and Applications to Large Scale Classification Systems

by

Jeremy Scott Hurwitz

S.B., Massachusetts Institute of Technology (2008)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2009

Certified by .....
Ahmad Abdulkader, Staff Software Engineer
VI-A Company Thesis Supervisor

Certified by .....
Tomas Lozano-Perez, Professor
M.I.T. Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Error-Correcting Codes and Applications to Large Scale Classification Systems

by

Jeremy Scott Hurwitz

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2009, in partial fulfillment of the
requirements for the degree of
Masters of Engineering in Computer Science and Engineering

Abstract

In this thesis, we study the performance of distributed output coding (DOC) and error-Correcting output coding (ECOC) as potential methods for expanding the class of tractable machine-learning problems. Using distributed output coding, we were able to scale a neural-network-based algorithm to handle nearly 10,000 output classes. In particular, we built a prototype OCR engine for Devanagari and Korean texts based upon distributed output coding. We found that the resulting classifiers performed better than existing algorithms, while maintaining small size. Error-correction, however, was found to be ineffective at increasing the accuracy of the ensemble.

For each language, we also tested the feasibility of automatically finding a good codebook. Unfortunately, the results in this direction were primarily negative.

Thesis Supervisor: Ahmad Abdulkader
Title: Staff Software Engineer

Thesis Supervisor: Tomas Lozano-Perez
Title: Professor

Acknowledgments

There are many, many people I need to thank for their help throughout the thesis process. On the MIT end: Professor Tomas Lozano-Perez, Professor John Guttag, and the VI-A program. On the Google end: Ahmad Abdulkader, Russell Smith, Michele Covell, Shumeet Baluja, and the entire Google OCR Team. And finally, on the "bug me to write the thesis" end, my mother and the beautiful Anya.

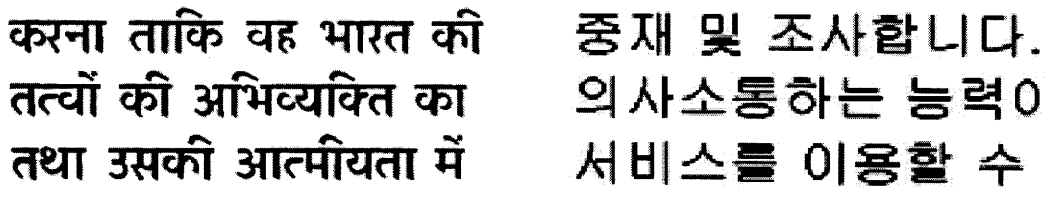
Chapter 1

Introduction

Optical Character Recognition (OCR) is the task of recognizing text in an image, or more recently, in a video. Clearly a large problem, it can be broken down into a series of independent tasks. First, the text within the image must be located and the layout and character set determined. Once the text has been linearized, it is segmented into individual words, which are then segmented into fundamental units (ie, letters or glyphs). Finally, the characters corresponding to the individual graphemes are identified. A post-processing step may also be used to fix errors, at the word or higher levels of granularity.

Each of these steps must be tackled by any OCR engine. Thankfully, each is largely (although not completely) independent from the others, allowing the different pieces to be improved independently. In this thesis, we concern ourselves with the character recognition step, which formally is a classification problem, mapping bitmaps to unicode symbols (or possibly a set of unicode symbols). That we are actually functioning within a larger OCR engine manifests itself in two ways. First, the character segmentation may not have worked correctly, meaning that the input does not correspond to any of the outputs and should be rejected as such. Secondly, a short list of alternative characters, along with the relative confidences, should be returned to allow for post-processing at the word level.

The primary approach to character classification is via machine learning. Machine learning algorithms can be generally grouped into two categories: discriminative and



(a) Devanagari (b) Hangul

Figure 1-1: Sample text in Hindi, written using Devanagari, and Korean, written using Hangul.

generative/density-based. Discriminative models, whose complexity depends on the number of free parameters, scale well with respect to the dimension of the input space but tend to perform badly as the number of output classes increases. Generative algorithms are the opposite, scaling well with the number of output classes, but suffering from the curse of dimensionality as the dimension of the input increases. Trying to bring together the best parts of each model is an active area of research in AI, with a wide variety of suggested approaches.

In this thesis, we investigate two ensemble-based approaches to scaling discriminative models to large languages. Distributed Output Coding (DOC) attempts to limit the number of parameters which must be learned by dividing the problem into independent subproblems. Error-Correcting Output Coding (ECOC) attempts to improve the resilience of the ensemble to random errors by introducing redundancy into the system. In both, instead of using a single classifier, a group of classifiers are used instead, whose results are combined in some fashion to yield the final result. If the ensemble is well designed, then erroneous results can often be out-voted, allowing the system to return the correct answer, even in the presence of noise.

To test the effectiveness of these methods, we considered two very large languages - Hindi (Devanagari) and Korean (Hangul). Each language contains approximately 10,000 distinct graphemes which must be differentiated. In Chapters 2 and 3, we present the background on the languages and ECOC. Then in Chapter 4, we study the effectiveness of a various types of codebooks. Finally, in Chapter 5, we turn our attention to the challenge of finding a codebook.

DOC was found to be very effective, resulting in small ensembles with good overall

accuracy, and the resulting OCR-engine prototype performed competitively against commercial engines. ECOC, however, was found to be ineffective. Ensembles designed solely based upon error-correction capabilities performed very poorly, and appending error-correcting capabilities to a good DOC ensemble did not improve the ensemble accuracy.

Chapter 2

Notation and Definitions

In this section, we present the formal mathematical framework used by DOC and ECOC.

2.1 Classifiers

A classifier takes as training data a set of labeled examples (x_i, y_i), with the samples drawn from a domain X and labels drawn from the set of possible classes $Y = [1, \ldots, k]$. Implicit in the classifier is also a hypothesis class $\mathcal{F}$, which is a subset of all possible functions $X \rightarrow Y$. Based on the training data, the classifier determines a function $f \in \mathcal{F}$ such that $f(x_i) = y_i$ for all i, if such a function exists. If no such function exists, the classifier instead determines the "best" function f, where "best" is defined according to some loss function L. A classifier is binary, or a dichotomizer, if there are exactly two output classes, which are then generally denoted ± 1.

A dichotomizer is a *margin-based classifier* if, instead of determining a function $f : X \rightarrow \pm 1$, it determines a function $f : X \rightarrow \mathbb{R}$. The output class is determined by $sign(f(x))$, while $|f(x)|$ provides a measure of confidence. Many standard binary classifiers, particularly support-vector machines and neural nets with sigmoid output, are margin-based. Since I will be focusing exclusively on margin-based classifiers in my thesis, all classifiers will be assumed to be margin-based unless otherwise stated.

2.1.1 Ensemble Classifiers

Given a dichotomizer, we wish to extend it to a k-class setting by using an ensemble of ℓ classifiers. To do so, we first determine ℓ encoding functions $E_1, \ldots, E_\ell$ that $[1, \ldots, n] \to \pm 1$, creating ℓ dichotomies. We then train ℓ classifiers on the induced binary problems, resulting in functions $f_1, \ldots, f_\ell$. Finally, we decode the ℓ results via a function D mapping $(f_1(x), \ldots, f_\ell(x))$ to a class y. In general, the E_j and D will be closely related, although neither strictly determines the other.

When designing an ensemble-based classifier, there are many ways to structure the ensemble. Each method tries to optimize a number of features, including robustness, conceptual simplicity, simplicity of the induced dichotomies, and size. Since it is generally impossible to optimize all of these features at one time, we must instead choose which aspects to focus on.

One-Versus-All Coding The simplest ensemble structure for multi-class learning is the one-vs-all approach. Given k possible classes, we train k classifiers, each of which is responsible for recognizing a single class. The final answer is then the class corresponding to the classifier which gives the most resounding "yes" for a given input. More formally, the encoding and decoding functions are given by

$$E_j(y) = \begin{cases} 1 & y = j \\ 0 & y \neq j \end{cases} \quad \text{and} \quad D(y^1, \ldots, y^k) = \arg\max_j y^j$$

One-vs-all possesses two primary advantages. The first is that it is fairly resilient to errors. Whether or not each binary classifier erred on a particular input is irrelevant, provided that the correct classifier had the largest margin. The second is that it is simple, both conceptually and in terms of the induced dichotomies. The downside is the number of classifiers required is linear in the size of the problem.

All-Versus-All Coding For completeness, we briefly mention all-vs-all coding here. In this scheme, $\Theta(k^2)$ classifiers are used, each distinguishing between a pair

of output classes. Although this method is competitive with most other ensemble schemes (see, for example, [1] and [10]), it is infeasible for domains with very large numbers of classes. With 1000 classes, almost 1 million classifiers would be required. Therefore, we will not address this scheme again.

Distributed-Output Coding To deal with the number of classes required in the one-vs-all approach, distributed-output coding was introduced. In this formulation, each of the k classes is assigned a codeword of size $n = \lceil \lg k \rceil$. We then train one classifier for each bit-position in the codewords. To decode simply requires running the new sample through each classifier, and returning the class that corresponds to the generated code word. More formally, if the code word for class y_i is $c_i^1 c_i^2 \ldots c_i^n$, the encoding functions are $E_j(y_i) = c_i^j$ and the decoding function maps the string $sign(f_1(x))sign(f_2(x)) \ldots sign(f_n(x))$ to the class corresponding to the nearest code.

DOC has the obvious advantage of only requiring a logarithmic number of classifiers, as compared to the linear number required in one-vs-all. However, it is very susceptible to errors. If even a single classifier errs, the final answer will be wrong. Therefore, each bit of the code must be very easy to learn.

2.1.2 Error-Correcting Output Codes

Error-correcting codes were introduced to try to get the resilience of one-vs-all, while achieving the space gains of DOC. This approach was first pioneered in [6], and has been the subject of a large body of research since. As in distributed-output representations, each output class is assigned a codeword of size n. However, in this case n is chosen to be larger than $\lceil \lg k \rceil$, so as to increase the minimum Hamming distance between code words. Intuitively, this allows the ensemble to recover from errors made by individual binary classifiers, since incorrect bits can be identified and corrected. This intuition can be formalized, if we assume that the errors made by the individual classifiers are independent. According to [13], if each classifier has

error-rate e, the worst-case error-rate for one-vs-all coding is

$$\Pr[\text{ensemble error}] = \min\{ke, 1\}$$

while for codes with error-correction the worst-case error-rate is

$$\Pr[\text{ensemble error}] = 2\frac{ne}{d},$$

where d is the minimum Hamming distance between code words. [13] further shows that the bound on one-vs-all coding is tight, while the bound on error-correcting codes is within a factor of 2 from being tight.

It should be noted, however, that this does not necessarily translate into actual performance improvements, since the error-rate e may be drastically different between the two methods. Indeed, [6] show that when using a decision tree as the underlying binary classifier, the number of leaves per classifier is often an order of magnitude larger in ECOC than in a one-vs-all ensemble. This issue is discussed in more detail in section 3.1.4.

One of the primary challenges in constructing an error-correcting ensemble is the construction of the encoding. Although the error-correcting code should have the Hamming Distance properties required by standard communications theory, use for classification places an additional restraint on the code. Each bit-position should be highly different from the other bit-positions, so as to hopefully reduce error-correlation between classifiers. This is due to the fact that most binary classifiers are symmetric with respect to the positive and negative classes, and therefore perfectly complimentary dichotomies will have perfectly correlated errors.

2.1.3 Unification of the Ensemble Methods

In [1], these methods were shown to be special instances of a more general framework. Given k output classes and ℓ binary classifiers, let $M \in \{\pm 1, 0\}^{k \times \ell}$ be a $k \times \ell$ matrix over $\{\pm 1, 0\}$. Each row of the matrix corresponds to a class, while each column

corresponds to a binary classifier in the ensemble. An entry of $+1$ corresponds to positive samples, an entry -1 corresponds to negative samples, and an entry of 0 means that that classifier makes no statement about that class.

In other words, given a matrix M mapping k output classes to ℓ classifiers, we have

$$E_j(y_i) = M_{ij} \quad \text{and} \quad D(y^1, \ldots, y^\ell) = \arg\min_i L(M_i, y^1, \ldots, y^{\{\ell\}})$$

where L is some loss function, such as the Hamming distance or inner product.

Although both ECOC and DOC are special cases of this unified framework, we maintain a distinction between the two approaches. When considering a codebook based on easy dichotomies, we refer to the approach as DOC. When attempting to use error-correction in the ensemble, we consider the approach to be ECOC.

2.2 Devanagari and Korean

Devanagari is the script used by many Indian dialect, particularly Hindi, Marathi, and Nepali. It consists of 36 consonants and 14 vowels, occupying Unicode points U+0900 to U+097F. When written, however, these letters are not simply arranged linearly as in most Latin scripts. Instead, the letters combine to form graphemes, as shown in figure 2-1a. Although the distortions do follow a pattern, the rules are generally too complicated and subject to variation to reliably encode directly into a computer; such an approach certainly does not scale to multiple languages. Furthermore, because the letters are distorted and merged when becoming a grapheme, a computer cannot easily segment out the individual letters from a grapheme. As a result, the number of distinct output classes a character classifier must recognize grows combinatorially. Since a syllable may have two or more consonants and a vowel, there are theoretically over $36 \cdot 36 \cdot 14 = 18144$ distinct graphemes possible. Thankfully, not every possible grapheme occurs in practice, greatly reducing the number of output classes. Based on a combination of books and webpages, Google estimates that roughly 1500 graphemes are needed to achieve 90% coverage and roughly 8500 graphemes are needed to achieve 99% coverage.

(a) The Devanagari grapheme *ddhrya*. The left half consists of the letter *da* (द) with the letters *dha* (ध) and *ra* (र) attached below. The right half consists of the single letter *ya* (य).

(b) The Korean grapheme *han*. The lead is *hieut*, the vowel is *a*, and the tail is *nieun*.

Figure 2-1: Sample graphemes for Devanagari and Hangul.

Hangul, the script used for Korean, also has a compositional structure. Each letter is composed of a lead consonant, a middle vowel, and a final consonant, as shown in figure 2-1b. There are 19 leads, 21 vowels, and 27 tails. In practice, only approximately 2000 graphemes are used. However, Unicode supports all possible combinations, spanning code points U+AC00 to U+FFDC. A specialized segmenter can, in principle, separate a Hangul grapheme into its constituent letters. However, most OCR engines do not contain such a specialized segmenter, and for this paper we treat the graphemes are indivisible.

In both languages, humans do not learn all possible graphemes by rote (as opposed to other Asian languages, such as Chinese). Instead, they learn the basic letters and the general rules for composition. Given a grapheme, it is generally fairly easy for a human to understand which letters form that symbol. This will be key to training a computer to do the same.

Chapter 3

Previous Work

It is crucial to note up front that the literature strongly disagrees on the performance of error-correcting coding versus one-vs-all coding. Many papers claim that error-correcting coding performs noticeably better (see, for example, [1] and [6]), while many others claim that the two methods are comparable (see, for example, [10] and [15]). In particular, [19] performs a detailed survey of the field, and reaches the conclusion that one-vs-all performs as well as ECOC. According to [19], papers stating otherwise have two flaws. The first is that the binary classifiers used in the experiments were not sufficiently well-tuned. The second is that the results, while technically statistically significant, were not actually practically significant, especially considering the increased conceptual and computational difficulty.

It is worth noting that in all of these papers, the number of classes under consideration rarely exceeded 30. Indeed, [12] specifically attempts to apply error-correcting coding to situations with a "large" number of classes, and yet only considers 105 classes.

3.1 Construction of the Coding Matrix

Against this backdrop of inconclusive results, research has attempted to optimize each facet of ECOC. Much of the research has obviously focused on the construction of the encoding used for error-correction.

3.1.1 Column Separation and Error Correlation

In [13], error-correlation is established on a theoretical basis as one of the most important factors for ensemble error-rate. As a result, many authors (for example, [4], [6], and [20]) have attempted to maximize the Hamming distance between the columns of the encoding matrix. Furthermore, since classifiers tend to be symmetric with respect to dichotomies, the Hamming distance between columns and the complements of columns is also maximized. These approaches have focused on producing codes based solely on their error-correcting properties, without taking a specific underlying domain into account.

3.1.2 Designing *A Priori* Codes

Depending on the properties required, a variety of methods exist for constructing error-correcting codes. The primary ones, particularly in early papers, generated the encoding based solely on their error-correcting properties and without taking the training data into consideration. Such methods include complete codes, random codes, and BCH codes. Complete codes consist of using all possible combinations, in the hopes of maximizing error-correcting capabilities and the information available with which to make a final decision. However, for k classes, the number of distinct columns is $O(2^k)$, which quickly becomes infeasible.

[4] showed that random codes achieve good row and column separation as the length of the code increases. In that approach, a large number of random matrices are generated, and the best one is selected. Multiple papers empirically support this effectiveness of this approach, although it is challenged in [12]. Two more deterministic approaches involve using BCH codes ([6]) and Hadamard matrices ([14]). However, these codes do not appear to work any better than random codes, and so may not be worth the complexity of generating them.

3.1.3 Designing *A Posteriori* Codes

Rather than construct the encoding based on coding theory, more recent research has focused on designing codes around the underlying domain. In [11] and [7], the code is designed so as to focus on distinguishing classes which are similar to each other. In [7], the authors suggest iteratively improving the coding-matrix by appending a classifier which is good at distinguishing the pair of classes with the highest mutual error-rate. In [11], it is suggested that the Hamming distance between the code words should depend on the distance between the corresponding classes (as measured by some metric on the input space). However, the authors of [11] do not actually implement such an algorithm, and instead attempt to minimize the dependence on such issues.

3.1.4 Simplifying Induced Dichotomies

In [6], the authors note that when using a decision-tree for the underlying classifier, each individual classifier tended to have roughly the same number of leaves as a decision-tree that solved the entire multi-class classification problem directly. This was roughly an order of magnitude more per tree than in the one-vs-all approach. These results showed that the induced dichotomies in ECOC tend to be much more complicated than the dichotomies of one-vs-all, and were in fact often almost as complicated as the original problem.

Since complicated dichotomies tend to increase error-rate, [18], [2], and [16] attempt to simplify the induced dichotomies by machine-learning the encoding dynamically. In all three approaches, a static code is used to generate an initial guess, and then some form of relaxation or gradient descent is used to improve the encoding. They found that the complexity of the induced dichotomies can be "drastically reduced . . . without deteriorating the overall generalization" ([16]). [10] uses this idea to instead try and improve the performance of the resulting classifier. However, the resulting classifiers still performed comparably to ECOC.

More recently, researchers have begun investigating how to dynamically construct

the code from the ground up, instead of relaxing a complete matrix. In [17] and [9], mutual information is used to find classes which relate to each other. These groupings are then used to determine the dichotomies in the coding matrix by a greedy algorithm.

Unfortunately, the problem-dependent encoding algorithms tend to be hierarchical in nature, and as a result tend to result in a linear number of classifiers. In fact, [16], which results in a codebook of polylogarithmic size, is the only method I could find in the literature which resulted in a codebook of size $o(n)$.

We also note in passing that designing optimal codes has been shown to be NP-hard in general ([5]). [5] also shows that this problem can be overcome if the encoding matrix is relaxed to allow real-valued entries. However, since sufficiently-good codes can be generated without using such machinery, real-valued encodings are generally not investigated, even in papers which cite their existence (for example, [15]).

3.2 Structure of the Ensemble

In terms of implementing the individual classifiers, [15] establishes that the individual classifiers should be implemented as separate, parallel classifiers, rather than as multiple outputs of a single monolithic classifier. Although this issue does not appear to be specifically addressed elsewhere, most other papers simply use parallel classifiers. We note that a parallel implementation is also desirable given the Google framework, since a monolithic architecture is much harder to distribute across machines.

As to which classifier to use for the binary classifiers, margin-based classifiers are preferable to non-margin-based, since they allow for fancier decoding. As to which margin-based classifier should be used, no consensus has been reached. in fact, mixed ensembles have been explicitly suggested as a means of reducing error-correlation.

3.3 Decoding

The most basic method for decoding is based solely on Hamming distance, and was first used in [6]. This idea has since been improved by the use of loss-based encoding, introduced in [1] and [13]. In loss-based decoding, the returned class is the codeword which minimizes

$$\sum_{j=1}^{\ell} L(y^j, M_{ij}),$$

where L is the loss-function minimized by the underlying binary classifiers. This approach or a variation of it is used in all of literature occurring after [1].

If the encoding matrix is taken over $\{\pm 1, 0\}$, the decoding becomes trickier. [8] provides a fairly comprehensive treatment of the topic.

Chapter 4

Codebook Styles

Given the many styles of codebook proposed throughout the literature, I began by testing their effectiveness when applied to thousands of possibly output classes.

For each codebook to be tested, I trained an ensemble of neural nets. The nets were had two primary architectures - fully-connected and convolutional. A fully-connected net consisted of two hidden layers, in addition to the input layer and output node. Every node was then connected to every node of the previous layer. Convolutional nets consisted of two convolutional layers, the second of which was then fully-connected to the output node. The exact layout of the classifiers is given in Appendix 6.

Training data consisted of artificially generated samples from a variety of fonts and sizes. These samples were then degraded using various heuristics modeling the degradation normally experienced during OCR. Depending on the situation, either single graphemes were generated or entire words were generated and then split by the segmenter from an existing OCR engine. The resulting training set contained 1 million samples, for an average of approximately 100 samples per grapheme, distributed according to the natural frequency of the graphemes. The testing set was generated by the same process on demand, again following the natural frequencies.

The obvious control to test against would be a monolithic multi-output neural net. Four such nets were trained, each containing two hidden layers of size 25, 50, 75, and 100. The three smallest converged, after a few months, to an accuracy well-below

10%. The largest, after four months, still had not converged and was terminated.

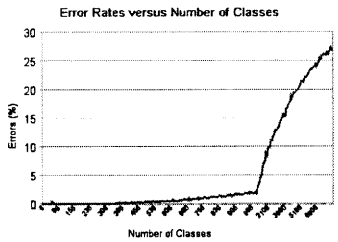
4.1 *A Priori* Codebooks

A priori codebooks, as discussed in section 3.1.2, are constructed specifically to have good error-correcting properties. However, intuitively they should have a high error-rate. After all, if the dichotomies have no structure to them, solving a single bit becomes nearly as difficult as solving the entire original problem.

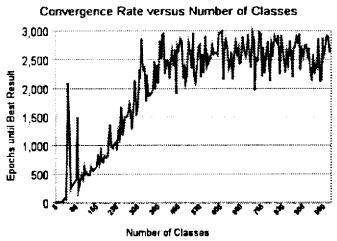
To test this intuition, I randomly split the first k output classes of Devanagari into two groups, for k ranging from 2 to 8000. Then, for each of these dichotomies, I trained a single-output neural net (two hidden layers, each of size 30). The results are shown in figure 4-1. For $k > 1000$, the net appears to have reached capacity, and performance begins to deteriorate quickly. The convergence rate is shown in figure 4-1b. Again, the graph contains a corner after which the convergence rate appears to have flattened. Interestingly, the critical points in the two graphs do not appear at the same value of k.

For comparison, figure 4-1c shows the convergence rate for dichotomies constructed specifically to be easy to learn (see section 4.2 for how these dichotomies were constructed). The net converged onto the easy dichotomies roughly twice as fast as it did for the *a priori* dichotomies. The error rates for the easy dichotomies is not shown since it never rose above 1%.

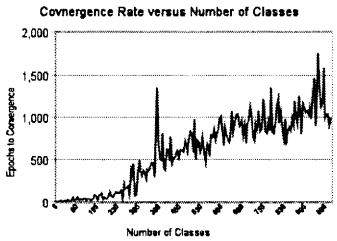
With an error rate above 25%, as it was for the full *a priori* dichotomy, correcting that many errors would require an impractically large number of extra classifiers, even if the information-theoretic optimal correction rate could be achieved in practice. Given these results, combined with the difficulties with *a priori* codes described in previous work, such encodings appear to not be a feasible approach.



(a) The error rate versus the number of classes for an *a priori* code.



(b) The convergence rate versus the number of classes for an *a priori* code.



(c) The convergence rate versus the number of classes for an *a posteriori* code.

Figure 4-1: The difficulty in learning an *a priori* dichotomy. For each graph, the training and testing samples were drawn uniformly from a fixed subset of the output classes. The classifier was a neural net with two fully-connected hidden layers of 30 nodes each.

4.2 Compositional Codebook and Results

Given that oblivious codebooks yielded such poor performance, any output coding clearly must take the structure of the input domain into account. We therefore turn our attention to distributed-output coding, with error-correction be introduced later.

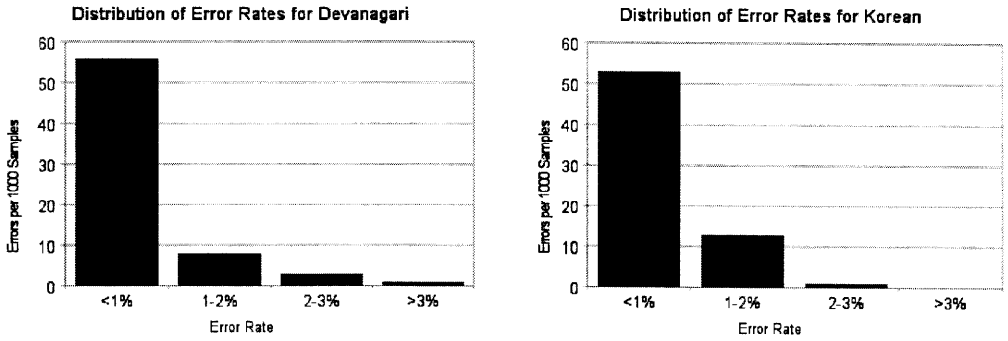
Hindi and Devanagari both contain an obvious compositional structure, which allows for the easy creation of a DOC code. Specifically, the graphemes of each language are created by composing a small set of fundamental units. This suggests the use of a *compositional codebook*, in which each classifier is responsible for deciding the presence/absence of a single fundamental unit within the grapheme.

Formally, let $\Sigma = \{\sigma_j\}$ be the alphabet of the language. For Devanagari, $|\Sigma| = 70$, while for Korean $|\Sigma| = 67$.[1] Let $G = \{g_i\}$ be the set of indivisible graphemes in the language. The codebook C will then have size $|G| \times |\Sigma|$. We set $C_{ij} = 1$ if g_i contains σ_j, and $C_{ij} = 0$ otherwise.

This codebook has the obvious advantage of being comprised of very simple dichotomies. As figure 4-2 shows, many bits had 0% error. On the other hand, there is essentially no error-correction, since flipping almost any bit of a codeword (particularly from a 1 to a 0) results in another valid codeword.

For Devanagari, this codebook cannot differentiate graphemes containing the same

[1]We consider the alphabet to be the set of Unicode points, not the size of the actual alphabet.



(a) Devanagari (b) Korean

Figure 4-2: The error-rate per bit for the compositional codebooks, using a neural net with two fully-connected hidden layers of size 30 each.

letters in a different order. For this thesis, I treated these graphemes as interchangeable, and counted as correct any output which erred only in the ordering of the letters within the grapheme. To handle this case in a full-fledged character classifier, a specialized second-level classifier can be dispatched which differentiates between those graphemes which differ only in ordering. Such a classifier would have at most four outputs, and these four graphemes are all drastically different, allowing for very high accuracy.

The compositional codebooks resulted in reasonably good performance, even without any pruning or other incremental improvements. As is shown in figure 4-3, using two hidden layers of size 30, we achieved an accuracy for Devanagari of almost 80%. For Korean, the accuracy was over 90%.

These accuracies, although promising, are clearly not good enough for a commercial OCR engine. However, a look at how the errors were distributed suggests multiple ways to boost the accuracy. According to figure 4-2, for the vast majority of bits, no or almost no errors occurred, even using a weak dichotomizer, emphasizing how easy these dichotomies were to detect. Certain other bits, however, had rather high error rates. Indeed, only 3 bits accounted for over a third of the total errors. These bits corresponded to letters which drastically changed their shape depending on their location, and therefore contained too much variety for the classifier to handle, as evidenced by the drastic improvement as the classifier was strengthened. By

Language	Network Layout	Average Bit Errors (%)		Accuracy (%)	
		Arithmetic	Geometric	Predicted	Actual
Devanagari	Hidden 10-10	14.64	10.98	36.62	(n/a)
Devanagari	Hidden 20-20	8.97	6.18	56.87	(n/a)
Devanagari	Hidden 30-30	6.65	4.57	65.89	78.64
Devanagari	Convolve 2-2	10.98	7.57	50.08	62.18
Devanagari	Convolve 3-3	7.71	5.29	61.73	72.52
Devanagari	Convolve 4-4	5.87	4.03	69.27	76.36
Korean	Hidden 10-10	6.87	4.47	66.51	81.92
Korean	Hidden 30-30	5.08	3.31	73.97	87.80

Figure 4-3: Summary of Results for the Compositional Codebook. This table shows the results of using a compositional codebook. The neural nets used either a fully-connected net with hidden layers of the specified size or a convolutional architecture with the two convolutional layers replicated the specified number of times.. The geometric mean of the errors is calculated as $1 - GeoMean(success)$, to ensure that it is well-defined. The predicted accuracy is based on assuming no error correction and uncorrelated errors.

splitting such cases into multiple bits, each corresponding to one form of the letter, the accuracy can be easily boosted without having to use large, and therefore slow, dichotomizers.

4.2.1 Optimizing the Ensemble

As a very basic optimization beyond the uniform ensemble originally tested, I trained multiple ensembles on the codebook, using multiple network layouts. For each bit, I then selected the best classifier from the trials. This allowed for some pruning, as well as avoiding some local minima from the training phase. I did not split complicated bits into simpler dichotomies, since that required a slow manual step[2].

The resulting classifier for Devanagari achieved a nearly 90% accuracy at the grapheme level, while a single multi-output neural net only achieved 85% accuracy. For Korean, the grapheme accuracy was almost 98%. At the word level, the Korean classifier was 96% accurate, as opposed to the commercial engine used by Google, which only achieved 88% accuracy. These results could most likely be further im-

[2]A native speaker would be able to perform this task fairly quickly.

proved by more agressive pruning, modifications to the codebook against hard dichotomies, and other standard optimizations.

It should be noted that these results may be slightly exaggerated, particularly for Korean. The training data given to the DOC ensemble was fairly similar to the testing data. The commercial OCR engine, on the other hand, was likely trained on different data. Therefore, these results should be interpreted as saying that DOC yields a competitive classifier, but not necessarily a superior one.

4.2.2 Effects of Error-Correction

Given the strong results achieved by distributed output coding, we now turn our attention to error-correction.

For the compositional codebook, slightly over 33% of the ensemble errors were caused by a single bit-error. Therefore, error-correction seems as though it could significatly boost the accuracy of the ensemble. Indeed, to correct against such errors, only $\lceil \lg(67) \rceil = 8$ error-correcting bits are required in theory.

To achieve this, I appended 8 bits to the compositional codebook, forming a BCH code with 1 bit of error correction. Unfortunately, the error-correcting bits had a 25% error rate. Assuming that the errors were indepenedent, this means that with probability $1 - 0.75^8 = 0.90$, an error occured in the error-correcting bits, exhausting the error-correcting capabilities of the code. In fact, with probability 0.63, more than one error occured, lowering the overall accuracy of the system. For only 10% of the samples were the extra bits able to contribute useful information.

This prediction closely matches the empirical results. The BCH-extended ensemble had an overall accuracy identical to the pure compositional ensemble; the two error rates were within each other's margin of error.

Interestingly, the accuracy of the error-correcting bits was nearly independent from the accuracy of the compositional bits. This leaves open the possibility that error-correcting bits could be appended in a useful way. However, such a system would most likely need to be domain specific and the number of extra bits needed before the signal became usable over the noise is likely to be prohibitively high.

Chapter 5

Designing a Codebook

Korean and Devanagari both contained a natural decomposition, thereby allowing a good codebook to be easily constructed semi-automatically. Given the unicode for the output class, simple scripts and existing libraries could determine the codeword, allowing automatic construction of the codebook. However, such behavior cannot be generally assumed, since in many domains simple dichotomies may not be readily apparent. We therefore would like to machine-learn the codebook, with minimal manual involvement. Such a codebook should yield good performance, while maintaining a small size.

5.0.3 Forgiving Hash Functions

Locality-sensitive hash functions are hash functions which approximately preserve distances, in that inputs which are close in the original space remain close in the compressed space, while inputs which are far apart remain far apart. Forgiving Hash Function are a specialized version of locality-sensitive hashing which were first designed as a means for doing quick database lookup of a song. The goal is to learn a locality-sensitive hash function which is easy to compute and resilient to noise. [3] used the hash function as the key to a hash table.

Similarly, we could use the forgiving hash function to generate the encoding for an ECOC ensemble. This approach does result in a linear-sized codebook. However,

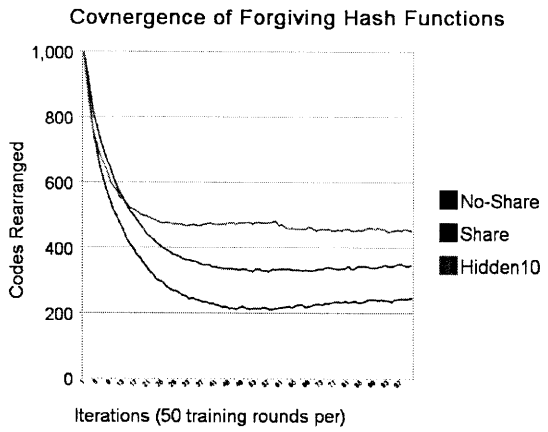


Figure 5-1: The convergence rate for Forgiving Hash Functions. In addition to not stabilizing, the ensemble seems to regress as the number of iterations increases.

the constant-factor is, in practice, much less than 1, so the resulting codebook would be feasibly sized.

Unfortunately, although this approach seems very promising and most likely deserves further study, I was unable to get the learning algorithm to converge, as shown figure 5-1.

5.0.4 Expanding ECOC

Originally introduced in [16], Expanding ECOC works by repeatedly executing the algorithm shown in figure 1. This algorithm constructs the codebook by repeatedly appending bits to the encoding until all classes have minimum distance Δ. While a pair of classes exist which are not sufficiently-well separated, a weak-classifier is trained to separate those two classes. A sampling of instances from every class is then presented to the classifier. If most samples from a class are classified as positive, that entry of the codebook gets a 1. If most samples are classified as negative, that entry of the codebook gets a 0.

This algorithm can also be generalized to allow for a "Don't Care" option. In this case, new bits are only assigned if they have a sufficiently high confidence. More formally, given a classifier and a class, the corresponding entry is labeled a 1 if at least a $1 - \epsilon$ fraction of the samples are classified as positive and is labeled a 0 if

Algorithm 1 The Expanding ECOC Algorithm. While a pair of classes exist which are not well separated, append a bit which separates those classes.

$M \leftarrow []$
$k \leftarrow 0$.
while there exist classes i, j such that $d(M_i, M_j) < \Delta$ **do**
  Train a classifier $\mathcal{C}$ with class i as positive examples and j as negative examples.
  for $\ell = 1 \ldots n$ **do**
    Run $\mathcal{C}$ on samples from class ℓ.
    Assign $M_{\ell,k} = \pm 1$ accordingly.
  end for
  $k \leftarrow k + 1$
end while

at least a $1 - \epsilon$ fraction of the samples are classified as negative. If neither case is satisfied, the entry is marked "Don't Care."

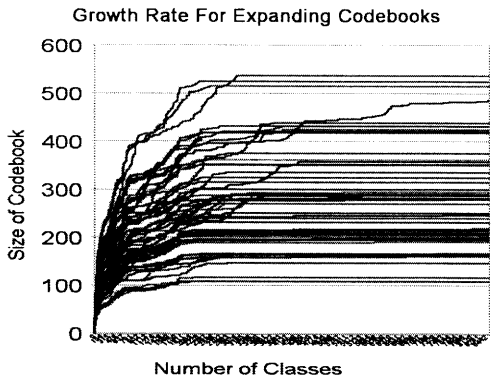
Since we want high distance between the columns of the codebook, we can further improve the algorithm by only including columns which are sufficiently different from the columns already created. Formally, if the new column differs from an existing column in less than Γ locations, we discard it.

To test the behavior of this system, I trained an ensemble using all combinations of

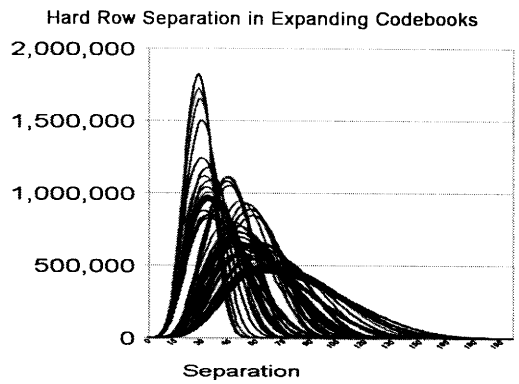
1. Minimum row distance - 1, 2, 3
2. Minimum column distance - 0, 100, 300, 500
3. Minimum bit confidence - 0.75, 0.8, 0.85, 0.9

Size of the Codebook Given a confidence parameter of ϵ, suppose that a given class is marked as a 0 with probability $f(\epsilon)$, a 1 with probability $f(\epsilon)$, and "Don't Care" otherwise, each independently at random. Then the expected size of the codebook is $\Theta(\lg n)$.

Experimentally, this prediction holds. As shown in figure 5-2a, the number of bits does grow roughly logarithmically. Furthermore, the separation between two classes obeys a roughly binomial distribution, as shown in figure 5-2b. Unfortunately, the



(a) The size versus number of classes for an ExpandingECOC codebook.



(b) The Hamming Distance between rows of an ExpandingEcoc codebook. If either row contained a "Don't Care" in a particular bit, that bit is not counted towards the Hamming Distance.

Figure 5-2: The growth rate and row separation for ExpandingECOC codebooks.

constant factor hidden in the $\Theta(\lg n)$ is very large; every codebook was well over one order of magnitude above the theoretical size necessary.

Accuracy Due to limits in available computation power, I only fully trained a single Expanding ECOC ensemble. I used a row separation of 1, a column separation of 300, and a bit-confidence of 0.9.

Since every bit should, by construction, have accuracy at least 0.9, we can use the distance distribution form figure 5-2b, along with an assumption of error independence, to estimate the performance of the ensemble. According to such an estimate, the overall ensemble should have an accuracy of roughly 92%, since the average bit error-rate was 0.07 with a maximum of 0.15.

However, in practice, the accuracy was much, much lower - the overall ensemble accuracy was just 15%. This was a result of two factors. First and primarily, the extreme number of "Don't Care" bits resulted in a fragile ensemble. Second, the bit errors were not uncorrelated, making the theoretical prediction nearly meaningless.

Chapter 6

Conclusions and Future Work

On both Devanagari and Hangul, a DOC ensemble of discriminative classifiers achieved excellent results relative to existing approaches. For Hindi, comparing the performance of ensemble to other approaches was difficult, since there was no previous result to compare it to (that I am aware of). However, I was able to show that its error rate was comparable to that achieved by classifiers working with a much-reduced output space. For Korean, the ensemble classifier appeared to work at least as well as the commercially-available alternative.

Given these promising results, there are numerous avenues for future work. The first is to implement an industrial-grade version of the compositional ensemble for both languages, and see how well a finely-tuned version performs. In particular, I focused on character-level accuracy, although OCR engines actually care about word-level accuracy. It is hopeful that on instances where the classifier errs, the correct answer will have a sufficiently small penalty that it will be corrected at the word level.

It also remains to compare the speed of the ensemble classifier to the speed of other approaches, both discriminative- and generative-based. Ideally, the ensemble approach will result in both higher accuracy and better speeds. However, if the accuracy is better while the speed is worse, then it may be possible to use the ensemble as a "second-pass" correction, in which only characters with a confidence below a given threshold are tested. Such a hybrid approach may result in getting the best of all worlds.

Another avenue is to use a hybrid approach within the ensemble itself. Such an approach may be useful, almost certainly for speed and probably for accuracy. In this work, I required the base-level classifier to always be a dichotomizer. However, the base levels could easily be a multi-output neural net or other multi-class classifier. For example, a single classifier could detect and identify all the Hindi vowels. Since many classifiers are likely performing similar internal calculations, this could greatly reduce the computation cost, and may also improve the accuracy and generalization. At a more extreme level, there could be a weak classifier performing a very simple classification, perhaps into only a few dozen general classes. These general classes are then solved using conventional techniques. In other words, the output coding is only used until the problem becomes tractable. Qualitatively, the marginal effectiveness of each successive bit appeared to diminish, so this approach seems very promising.

Focusing on construction of the codebook, the major open problem is obviously to do so with minimal human intervention. Running a clustering algorithm on the input space and using the resulting clusters to seed the codebook seems fruitful. Allowing some manual intervention, it would be useful to start with a human-generated codebook and attempt to refine that codebook. Regardless of the approach, it may be useful to only require the encoding to shrink the problem into tractable subproblems, since splitting the final few classes from each other is far harder than finding broad distinctions.

An automated approach to learning the codebook would also allow the approach to be applied to situations which lack a compositional structure. Given a language such as Chinese, manually constructing a codebook could be tedious even if good features can be identified. It remains to determine if this approach can be generalized effectively to less structured domains.

Regardless of the results of such future work, ensemble-based approaches have been shown to be a viable approach for efficiently scaling discriminative classifiers to structured domains with very large numbers of output classes, without sacrificing an undue amount of accuracy or time.

Appendix: Structure of the Neural Networks

Below are the formal descriptions of the layout for the various neural networks used in this thesis. The basic connections are

- **fullconnect** *inputs outputs*
- **convolve** *inputs outputs in_size_x in_size_y window_x window_y step_x step_y out_size_x out_size_y copies_x copies_y*

Hidden 10-10

```
input in[0,2303]
output out[0,0]
fullconnect in[0,2303] hidden_1[1,10]
fullconnect hidden_1[1,10] hidden_2[1,10]
fullconnect hidden_2[1,10] out[0,0]
```

Hidden 20-20

```
input in[0,2303]
output out[0,0]
fullconnect in[0,2303] hidden_1[1,20]
fullconnect hidden_1[1,20] hidden_2[1,20]
fullconnect hidden_2[1,20] out[0,0]
```

Hidden 30-30

```
input in[0,2303]
output out[0,0]
fullconnect in[0,2303] hidden_1[1,30]
fullconnect hidden_1[1,30] hidden_2[1,30]
fullconnect hidden_2[1,30] out[0,0]
```

Convolve 2-2

```
input in[0,2303]
output out[0,0]
convolve in[0,2303] layer_1[1,900] 48 48 6 6 3 3 15 15 2 2
convolve layer_1[1,900] layer_2[1,324] 30 30 6 6 3 3 9 9 2 2
fullconnect layer_2[1,324] out[0,0]
```

Convolve 3-3

```
input in[0,2303]
output out[0,0]
convolve in[0,2303] layer_1[1,2025] 48 48 6 6 3 3 15 15 3 3
convolve layer_1[1,2025] layer_2[1,784] 45 45 6 6 3 3 14 14 2 2
fullconnect layer_2[1,784] out[0,0]
```

Convolve 4-4

```
input in[0,2303]
output out[0,0]
convolve in[0,2303] layer_1[1,3600] 48 48 6 6 3 3 15 15 4 4
convolve layer_1[1,3600] layer_2[1,1444] 60 60 6 6 3 3 19 19 2 2
fullconnect layer_2[1,1444] out[0,0]
```

Bibliography

[1] Erin L. Allwein, Robert E. Schapire, and Yoram Singer. Reducing multiclass to binary: A unifying approach for margin classifiers. In *Proc. 17th International Conf. on Machine Learning*, pages 9–16. Morgan Kaufmann, San Francisco, CA, 2000.

[2] E. Alpaydin and E. Mayoraz. Learning error-correcting output codes from data. *Artificial Neural Networks, 1999. ICANN 99. Ninth International Conference on (Conf. Publ. No. 470)*, 2:743–748 vol.2, 1999.

[3] Shumeet Baluja and Michele Covell. Learning to hash: forgiving hash functions and applications. *Data Mining and Knowledge Discovery*, 17(3):402–430, December 2008.

[4] A. Berger. Error-correcting output coding for text classification, 1999.

[5] Koby Crammer and Yoram Singer. On the learnability and design of output codes for multiclass problems. In *Computational Learing Theory*, pages 35–46, 2000.

[6] Thomas G. Dietterich and Ghulum Bakiri. Solving multiclass learning problems via error-correcting output codes. *Journal of Artificial Intelligence Research*, 2:263–286, 1995.

[7] Sergio Escalera and Oriol Pujol. Ecoc-one: A novel coding and decoding strategy. In *ICPR '06: Proceedings of the 18th International Conference on Pattern*

Recognition, pages 578–581, Washington, DC, USA, 2006. IEEE Computer Society.

[8] Sergio Escalera, Oriol Pujol, and Petia Radeva. Decoding of ternary error correcting output codes. In Jos Francisco Martnez Trinidad, Jess Ariel Carrasco-Ochoa, and Josef Kittler, editors, *CIARP*, volume 4225 of *Lecture Notes in Computer Science*, pages 753–763. Springer, 2006.

[9] Sergio Escalera, David M.J. Tax, Oriol Pujol, Petia Radeva, and Robert P.W. Duin. Subclass problem-dependent design for error-correcting output codes. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 30(6):1041–1054, June 2008.

[10] Eibe Frank and Stefan Kramer. Ensembles of nested dichotomies for multi-class problems.

[11] R. Ghaderi and T. Windeau. Circular ecoc. a theoretical and experimental analysis. *Pattern Recognition, 2000. Proceedings. 15th International Conference on*, 2:203–206 vol.2, 2000.

[12] Rayid Ghani. Using error-correcting codes for text classification. In Pat Langley, editor, *Proceedings of ICML-00, 17th International Conference on Machine Learning*, pages 303–310, Stanford, US, 2000. Morgan Kaufmann Publishers, San Francisco, US.

[13] Venkatesan Guruswami and Amit Sahai. Multiclass learning, boosting, and error-correcting codes.

[14] Shilei Huang, Xiang Xie, and Jingming Kuang. Using hybrid hadamard error correcting output codes for multi-class problem based on support vector machines. *Computational Intelligence and Security, 2006 International Conference on*, 1:7–10, Nov. 2006.

[15] F. Masulli and G. Valentini. Effectiveness of error correcting output coding methods in ensemble and monolithic learning machines. *Pattern Anal. Appl.*, 6(4):285–300, 2003.

[16] Eddy Mayoraz and Miguel Moreira. On the decomposition of polychotomies into dichotomies. In *ICML '97: Proceedings of the Fourteenth International Conference on Machine Learning*, pages 219–226, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[17] O. Pujol, P. Radeva, and J. Vitria. Discriminant ecoc: a heuristic method for application dependent design of error correcting output codes. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(6):1007–1012, June 2006.

[18] Gunnar Rätsch, Alexander J. Smola, and Sebastian Mika. Adapting codes and embeddings for polychotomies.

[19] Ryan Rifkin and Aldebaro Klautau. In defense of one-vs-all classification. *J. Mach. Learn. Res.*, 5:101–141, 2004.

[20] T. Windeatt and T. Ghaderi. Coding and decoding strategies for multi-class learning problems, 2003.