SANDIA REPORT

SAND2013-9344

Unlimited Release

Printed September 30, 2013

Network and Ensemble Enabled Entity Extraction in Informal Text (NEEEEIT) Final Report

Philip Kegelmeyer
Timothy M. Shead
Danny Dunlavy

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: <http://www.osti.gov/bridge>

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>

Network and Ensemble Enabled Entity Extraction in Informal Text (NEEEEIT) Final Report

Philip Kegelmeyer
Computer Sciences and Information Systems Center
Sandia National Laboratories
P.O. Box 969
Livermore, CA, 94551
wpk@sandia.gov

Timothy M. Shead
Data Analysis and Visualization Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1326
tshead@sandia.gov

Daniel (Danny) Dunlavy
Data Analysis And Informatics Department
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1326
dmdunla@sandia.gov

Abstract

This SAND report summarizes the activities and outcomes of the Network and Ensemble Enabled Entity Extraction in Informal Text (NEEEEIT) LDRD project, which addressed improving the accuracy of *conditional random fields* for *named entity recognition* through the use of *ensemble methods*.

Conditional random fields (CRFs) are powerful, flexible probabilistic graphical models often used in supervised machine learning prediction tasks associated with sequence data. Specifically, they are currently the best known option for named entity recognition (NER) in text. NER is the process of labeling words in sentences with semantic identifiers such as “person”, “date”, or “organization”.

Ensembles are a powerful statistical inference meta-method that can make most supervised machine learning methods more accurate, faster, or both. Ensemble methods are normally best suited to “unstable” classification methods with high variance error. CRFs applied to NER are very stable classifiers, and as such, would initially seem to be resistant to the benefits of ensembles.

The NEEEEIT project nonetheless worked out how to generalize ensemble methods to CRFs, demonstrated that accuracy can indeed be improved by proper use of ensemble techniques, and generated a new CRF code, “pyCrust” and a surrounding application environment, “NEEEEIT”, which implement those improvements.

The summary practical advice that results from this work, then, is:

- When making use of CRFs for label prediction tasks in machine learning, use the pyCrust CRF base classifier with NEEEEIT’s bagging ensemble implementation. (If those codes are not available, then de-stablize your CRF code via every means available, and generate the bagged training sets by hand.)
- If you have ample pre-processing computational time, do “forward feature selection” to find and remove counter-productive feature classes.
- Conversely, if pre-processing time is limited, use NEEEEIT’s “edited clone” pyCrust mechanism, along with a more modest use of bagging, to generate ensembles much more quickly.

Acknowledgment

This work was funded under LDRD Project Number 151292 and Title “Network and Ensemble Enabled Entity Extraction in Informal Text (NEEEEIT)”.

Contents

1	Introduction	11
1.1	Why Entity Extraction?	11
1.2	The Original Goals and Proposed Methods	11
1.3	An Overview of the Technical Issues and Approaches	11
1.4	A Summary Of Technical Accomplishments And Issues	14
2	Related Work and Conceptual Challenges	17
2.1	Named Entity Extraction	17
2.2	Conditional Random Fields	17
2.3	Ensemble Methods	17
2.3.1	CRFs are Stable Classifiers	17
2.3.2	Token Data Versus Sequence Data	18
2.3.3	Stratification	18
2.3.4	Sequence Labeling	19
2.3.5	Performance Metrics	20
2.4	CRFs and Ensembles	21
3	Data	23
3.1	Corpus Interface	23
3.1.1	Tokens	23
3.1.2	Sequences	23
3.1.3	Labels	24
3.1.4	Features	24
3.1.5	Metadata	28
3.1.6	Threads	28
3.1.7	Implementations	29
3.1.8	Frozen Data	29
3.1.9	Manual Labeling	30
3.2	Data Sources	31
4	pyCrust	33
4.1	The CRF Model	33
4.2	Existing CRF Model Implementations	35
4.3	pyCrust Implementation Details	37
4.3.1	pyCrust Classes	38
4.3.2	CRF Model Training	39
4.3.3	CRF Model Inference	40
4.3.4	Open Questions and Future Work	41
5	NEEEEIT	43
5.1	Experiments	43
5.1.1	Growing Ensembles	43
5.1.2	Evaluating Ensembles	45

5.2	Postprocessing	46
5.2.1	Voting	46
5.2.2	Performance Metrics	48
5.2.3	Analyses	48
6	Experiments, Speculations, Conclusions	51
6.1	Ensemble Experiments	51
6.1.1	Single CRFs and Basic Bagging	51
6.1.2	Ensemble Size	53
6.1.3	Diversity Via Small Bags	53
6.1.4	Diversity Via Feature Sampling	56
6.1.5	The Effect of Voting Scheme	59
6.2	The Effect of Feature Selection	61
6.2.1	Simple Feature Selection	61
6.2.2	Greedy Feature Class Selection for Single CRFs	63
6.2.3	Greedy Best Feature Class Selection for Ensembles	64
6.2.4	Focusing On Features That Are Individually Useful	65
6.2.5	Greedy Worst Feature Class Avoidance for Ensembles	66
6.2.6	Light, Middle, and Heavyweight Features for Ensemble CRF Improvement	67
6.3	Feature and Factor Graph Visualization	69
6.3.1	Feature Class Similarity	69
6.3.2	Factor Graph Visualization	74
6.4	Lambda Manipulation	75
6.4.1	Lambda Cleaning	75
6.4.2	Knockout Ensembles	78

Appendix

A	Loose Ends	84
A.1	Ideas That Were Under Active Investigation When The Project Stopped	84
A.1.1	Advanced Feature Selection	84
A.1.2	CRF Creation Via Lambda Manipulation	85
A.1.3	Combining Paths to Diversity	85
A.1.4	Mallet Destablization	86
A.1.5	Feature-Class/Label Transition Pools	86
A.2	Miscellaneous Open Issues	87
A.3	Miscellaneous Research ideas	87

Figures

1	Manually labeling corpus tokens in a web browser.	30
2	A taxonomy of the current family of graphic models (excerpted from [2]).	34
3	Accuracy vs Pairwise Kappa as a Function of Random Subspace Size	49

4	Bagging pyCrust with Newswire Data, as a Function of Size	54
5	Bagging pyCrust with NGC, as a Function of Bag Size	55
6	Bagged Random Subspaces Applied to CRFs and Feature Classes	57
7	Unbagged Random Subspaces Applied to CRFs and Feature Classes	58
8	Bagging pyCrust with NGC, as a Function of Size and Feature Sampling	61
9	SimpleMajority voting vs UnanimousOther-0.4 for a Large Ensemble.	62
10	Single CRF Accuracy as Feature Classes are Added	64
11	Feature Sampling with Focused Features	65
12	Ensemble Size Vs. Number of Useful Feature Classes	66
13	Subset: Ensemble Size Vs. Number of Useful Feature Classes	67
14	Accuracy from Uncleaned, Cleaned, or Non-Toxic Features	68
15	Accuracy from Light and Midweight Features Only	69
16	Accuracy from Lightweight Features Only	70
17	Hierarchical Clustering of Cosine Distance between Feature Class Population Vectors	71
18	KS Feature Class Similarity for “O to B-PER” Label Transition	72
19	KS Feature Class Similarity for All Label Transitions	73
20	A Conceptual Factor Graph	74
21	The Feature Classes Activated by “Scott and”.	76
22	Feature Selection Accuracy after Lambda Cleaning	77
23	Ensemble Accuracy with Cleaned and Uncleaned Lambdas	78

1 Introduction

1.1 Why Entity Extraction?

Much of the world’s actionable information is locked up in increasingly unmanageable volumes of text. This has inspired work in “entity extraction” (EE), which is the detection of meaningful terms in text: persons, places, dates, etc. Robust, accurate entity extraction is the crucial first step for all information extraction from text; you can’t “connect the dots” unless you identify the dots in the first place.

The original work on entity extraction focused on clean, edited, “formal” text. Unfortunately, much of the information of interest is in personal, *informal* text, such as email or blogs. These differ from the Medline abstracts and Reuters news stories commonly studied in that they tend to be riddled with errors of spelling and grammar, abbreviations, and a terseness and obliquity of reference borne of the context shared by sender and receiver. Prior to the work reported on in this SAND Report, the informal text domain had received scant study; just enough to show that it causes the performance of existing methods to drop precipitously.

1.2 The Original Goals and Proposed Methods

We thus proposed to improve the accuracy of entity extraction in informal text via application of Sandia-specific expertise in “ensemble” machine learning methods. We further intended to exploit new high performance computing network capabilities to permit the integrated analysis of a full conversational thread. (The thread idea remains promising, but ultimately we did not fully explore it in this project.)

By creating methods to make accurate entity extraction in informal text possible, and by leveraging Sandia’s prior investment in text analysis tools and methods, we aimed to deliver a system that starts with raw email and produces high quality entity extraction, suitable for search, linking, visualization, and analysis.

1.3 An Overview of the Technical Issues and Approaches

The Challenge To illustrate the challenges posed by informal text, note the difference between finding the two people (in boldface) mentioned in [22]

Sherron Watkins, the Enron Corp. executive who warned longtime chairman **Kenneth L. Lay** last August that the company might collapse because of accounting scandals, offered him public relations advice last fall on how to blame others for Enron’s swiftly escalating financial problems.

vs the four in [15]

Susan I would love to - but as usualy I cant - I have a Dinner Scheduled with **greg** at APB Monday - he needs to vent about **KL** - and I am trying not to schedule more than one night per week for outings. . .

sb

The informal text domain had previously received scant study; just enough to definitively show that it causes the performance of existing methods to drop precipitously. “F1” is the standard measure of performance for entity extraction. It ranges on $[0, 1]$, and values of around 0.90 are generally considered to be a practical minimum: any lower and the results are so inaccurate or incomplete as to make it impossible to get useful information from the text. Current commercial and academic methods reliably achieve F1 of around 0.98 for formal text. F1 for the same methods drops to around 0.72 when applied to email [25].

Background So, to address the additional challenges posed by informal text, we investigated the application of “ensemble” methods to the standard entity extraction algorithm. All ensemble methods have “statistical machine learning” at their core. As such, they require:

1. *Training data*, to serve as examples of the idea being learned. An example in this case would be a large number of emails in which every proper name has been identified.
2. A set of *features* to describe each data point. Example feature classes for a word in an email might be IS-CAPITALIZED, PART-OF-SPEECH, or PERSONAL-PRONOUN.
3. A *classifier* algorithm, which looks at all of the training data and deduces the patterns in the features which indicate the likelihood of, for instance, a name in text.

Through statistical analysis, the classifier produces a set of rules that can be applied to each word in a new email, to predict whether that word is part of a named entity.

Ensembles are a relatively new development in classifier theory. There are various ensemble methods; they all involve looking at multiple, structured, *subsets* of the information available, and building a separate classifier for each subset [18]. This results in each classifier having a slightly different perspective. When a new word in an email must be classified, all of the classifiers are consulted, and their opinions blended together. Sandia has already successfully exploited ensembles to scalably handle huge and disjointed volumes of *non-text* data, and further, to handle data that is noisy, incomplete, skewed, and in general riddled with error.

Challenges for Ensembles in Entity Extraction Text application examples were used in the above discussion, to be concrete, but in fact this ensemble idea has not yet been commonly applied to text entity extraction. This is likely due to the fact that ensemble research to date has largely focused on scientific data that can be largely described numerically. For instance, to determine

whether a node in a structural simulation is part of a “crumple zone”, sensible features would be LOCAL-AVG-STRESS, TOTAL-DISPLACEMENT, LOCAL-AVG-TEMP, and so on. The use of numerical features means that every data point exists in an N-dimensional space with an associated distance metric; this concept of distance is central to most classifiers.

The proper feature classes for entity extraction are very different. They are not numerical. They tend to be binary (IS-CAPITALIZED) or to be “categorical”, which means they take on one of set of distinct, non-numerical values. This can be a small number of values (PART-OF-SPEECH) or a huge number (PREVIOUS-TOKEN).

Since non-numeric features do not permit a distance metric, standard classifiers cannot be used. Current state of the art entity extraction therefore uses an entirely different classifier method called Conditional Random Fields (CRFs) [33]. CRFs are slow and unwieldy, but they don’t require a distance metric, and they do gracefully handle categorical features.

However, because CRFs are known as “stable” classifiers, they resist standard ensemble methods. We believed that there was nothing, in principle, which forbade the use of ensemble methods with CRFs, though we anticipated that there would be a number of subtle and difficult issues to resolve. For example, ensembles depend on diversity of opinion in the base classifiers. For standard classifiers, it is fairly well understood how to achieve diversity while maintaining accuracy, but this question has not been studied with CRFs.

Core Idea: Apply Ensembles to CRFs The core technical contribution of this LDRD, then, was to be to bring the power of ensembles to CRFs and categorical features.

There has been considerable prior work, much of it conducted by Sandia, in using ensembles to dramatically improve machine learning methods [1] as applied to non-text data, but ensembles for CRFs have received little attention, at Sandia or elsewhere. Still, ensembles held out great promise for CRFs and entity extraction. In addition to increasing accuracy, ensembles are beautifully suited for high performance computing, as they can trade off the number of processors for accuracy in a quantitative fashion. CRFs are indeed slow, but Sandia has previously done groundbreaking work [7] to develop a technique called “ensembles of bites” that handled the issue of slow base classifiers in non-text data. By working with very small data subsets — tiny “bites” of the original data — each classifier can be relatively speedy to build. And by structuring those small subsets correctly, no accuracy will be lost.

Integrated Analysis of a Conversation There is one other aspect of informal text that is challenging, yet which may be potentially exploited. Formal text is generally intended to be self-contained. Informal text, being personal, is usually part of a *chain* of communication; email threads are a good example. So any one email can be impossibly cryptic; but the entire thread, considered as a whole, can perhaps be coherent enough to help disambiguate its pieces.

This fact is understood by the research community, and there is even a proposed formalism, “skip-chain” CRFs [33], for linking information across an email thread or across a blog post and its

offspring. Unfortunately CRFs are already computational demanding, with skip-chain CRFs even more so, and thus have been unusable to date even though they demonstrably improve accuracy (on suitably small data).

We expected the “ensembles of bites” approach to allow us to scale and apply skip-chains CRF against text corpora of useful size. Further, skip-chain CRFs are, behind the scenes, essentially network analysis methods, and so we also expected Sandia’s network expertise, in hardware and in the algorithmic advances from the Networks Grand Challenge LDRD, to aid in making skip-chain CRFs into a practical method.

Ultimately, we didn’t closely investigate either idea, as we did not extend our new CRF code, “pyCrust”, to handle skip-chain CRFs. We did, however, investigate integrated analysis by building and testing “thread aware” voting strategies; see Section 5.2.

1.4 A Summary Of Technical Accomplishments And Issues

A brief summary of the main findings of this project, all of which are further elaborated later:

- *There are interesting conceptual difficulties in applying ensemble methods to CRFs.*

Standard ensemble methods were developed for “point” data. That is, they operate on base classifiers which are labeling *individual* samples of data. CRFs operate on “sequence” data; they simultaneously assign the best *set* of labels to all tokens in a sequence. This complicates basic machine learning mechanisms such as stratified partitioning (Section 2), has led to a research community with a surprising lack of uniformity in metrics and terminology around performance (Section 2), and required new and creative thinking in order to conceive of ensemble methods specifically suited to CRFs (Section 6.4.2).

- *CRF stability was indeed a barrier to useful ensemble use.*

One of the reluctant conclusions of our investigations was that no existing CRF code permitted the degree of instrumentation and internal manipulation required to support our research. So we made a lengthy investment in building a new code: pyCrust (Section 4).

- *Despite these first two points, we were ultimately able to construct ensemble methods that did improve CRF accuracy.*

One happy result of the pyCrust investment was the ability to demonstrate that prior CRF stability is (at least partially) designed, rather than innate. Altering and undermining that design permitted ensemble methods to improve both on prior stable methods and on single-CRF unstable performance (Section 6.1.1).

- *Ensembles of decision trees do not degrade in accuracy as ensemble size grows; this seems also to be true of ensembles of CRFs.*

Ensembles of decision trees are more or less the dominant practical machine learning method for point data. One of the reasons is that they don’t degrade in performance as ensemble size

grows, which removes one point of design stress in their practical use. We were able to empirically show that this attractive property holds as well for ensembles of CRFs (Section 6.1.2).

- *Ensembles of decisions trees are robust to the inclusion of weak or irrelevant features; this is not true for ensembles of CRFs.*

Ensembles of decision trees are easy to use because there is no need for careful feature selection, normalization, de-correlation, or whatnot. This, unfortunately, is *not* true of ensembles of CRFs, or even single CRFs. Indeed, we were able to identify the existence of “toxic” features that reduced CRF performance. Further, we generated anecdotal evidence that careful feature selection in CRFs might, by itself, achieve much of the performance gain observed with proper use of ensemble methods (Section 6.2).

- *The factor graph formalism underlying CRFs permits a novel “edited clone” ensemble algorithm which is both unique to CRFs and much faster than ensemble methods developed for non-CRF base classifiers.*

The process of building a single CRF is basically the process of inferring, from data, the λ values on the edge of a complex factor graph (Section 4). Ensemble methods typically generate diverse base classifiers by changing the training data, which results in a factor graph with altered λ .

We created a new “knock-out” ensemble method which takes a single constructed CRF and generates many diverse variations *directly*, by altering or removing λ weights. This is *much* faster than inferring each new CRF, and demonstrably recovers the same performance achieved by bagging, a standard and straightforward, but much slower, method (Section 6.4.2).

2 Related Work and Conceptual Challenges

2.1 Named Entity Extraction

There has been considerable prior work, both academic and commercial, in named entity extraction (NER) from text. There have even been open contests, with provided data [8]. However, the contest data has all been formal text, particularly Reuters news stories and abstracts from technical journals. There has been very little work on entity extraction in informal text. And the work which exists either does not make use of the ensemble ideas proposed here [25] or mainly serves to document how precipitously performance plunges when standard entity extraction methods are applied to informal text [6]. There are commercial tools (Inxight is an example) that claim to handle entity extraction in email, but upon inspection, their accuracy claims are based on processing the semi-structured email *header* information, not the unstructured body text.

Accordingly, NER continues to be an active field of study. A useful overview of the concepts is in *Semi-Supervised Named Entity Recognition: Learning to Recognize 100 Entity Types with Little Supervision*, a Ph.D thesis from David Nadeau [27]. This treatment is particularly strong on describing example features and methods for evaluating NER.

2.2 Conditional Random Fields

There is, unfortunately, still no single reference for conditional random fields which is both comprehensive and clear. The seminal paper is likely *Conditional random fields: Probabilistic models for segmenting and labeling sequence data*, Lafferty et.al [19]. Useful tutorial treatments are in *Conditional Random Fields: An Introduction*, by Hannah Wallach [36] and *An introduction to conditional random fields for relational learning* by Sutton and McCallum [33].

2.3 Ensemble Methods

Ensembles [18], and in particular ensembles of decision trees [5], are a robust, reliably near-optimal method for practical, robust, accurate supervised machine learning — for *non-sequence* data. The robust reliability of ensembles of decisions trees is what, in part, inspired the NEEEEIT research, as ensembles of trees provide many capabilities that are challenging to replicate when applied to CRFs in support of NER.

2.3.1 CRFs are Stable Classifiers

Ensemble methods work best when the individual “base” classifiers in the ensemble have low bias error, and thus have high variance error. In other words, ensemble methods are useful when base classifiers are very responsive to the training data, and indeed, tend to overfit the data. In a *single*

classifier, overfitting the data is a bad thing, but in standard ensemble methods such as bagging [5], the data varies slightly from base classifier to base classifier. The overfitting means that the resulting classifiers tend to be diverse, and diverse classifiers lead to more accurate ensembles.

CRFs, however, tend to be very stable classifiers with very low variance error. This stability is inherent in their nature. A decision tree, for instance, assigns a label only to one token at a time, and so can focus tightly only on the features describing that one token. A CRF, on the other hand, finds the assignment of labels that is *simultaneously* the most likely set for *all* of the tokens in the current sequence. This simultaneous consideration of all of the tokens smoothes out the response to any one of them, lowering the variance error and diversity, and thus lowering the utility of ensembling. And indeed, it proved in this project that increasing the variance error of the base classifiers algorithmically was the main key to making ensembles work with CRFs.

2.3.2 Token Data Versus Sequence Data

Another major difference between CRFs as applied to NER of text and more classic machine learning methods is the “token” vs “sequence” dichotomy, alluded to above.

In classic supervised machine learning, each training *and* test sample stands alone. Each sample, each language token in the text context, has its own individual set of features, and its own class label. The classifier is built by considering each training token one at a time, slowly building a model for how to predict the class label from the features. The classifier is used by considering each test token one at a time, assigning it a class, and each individual test token is “right” or “wrong” according to whether it was assigned the correct class label.

As will be discussed in detail in Section 4, this is considerably more complicated with CRFs. CRFs operate on tokens linked in some graphical structure. In the case of NER on text studied here, that graphical structure is simply a sequence. So each token may have “unigram” features that apply to it alone, but unlike the classic model just described, it also has “bigram” features which link it and the previous token in sequence. Further, when assigning class labels to a test sequence, the sequence is considered as whole, with the bigram features helping to link the classification of the individual tokens in the sequence. However, the final “accuracy” of the sequence is assessed according to the accuracy of the tokens.

2.3.3 Stratification

This token vs sequence tension introduces a number of complications; one is stratification of data.

Many data sets are unbalanced in the proportions of class labels observed. This is very true of NER text data, for instance, as the vast bulk of the tokens tend to be of class “other”. When partitioning data (automatically, as for N-fold cross-validation, or deliberately, as when setting up stable train, test, and validation data sets), one typically tries to maintain the original class proportions in the partitions, so that each partition can resemble, statistically, the parent partition.

With pure token data, this is more or less straightforward. Just take each class in turn and split it into the requisite number of pieces before accumulating the pieces into the partition you are growing.

With sequence data, this is *not* straightforward. You can not simply break everything down to tokens and partition the tokens, as the whole point of a CRF is to learn from a *sequence*, that is, from the sequential ordering of the tokens. However, assigning a sequence to a partition means assigning an irreducible mix of class labels all at once, which might make it impossible to balance all of the labels across all of the partitions.

So in practice, when partitioning data for N-fold cross validation, we simply assigned sequences at random, and hoped for the best, depending on the brevity (relative to corpus size) of the sequences in our corpus.

When building semi-stable partitions by hand, as in the NEWSWIRET/NEWSWIREV data described in Section 3.2, we built the partitions in a greedy fashion, incrementally assigning each sequence to whichever partition provided, at that moment, the best match to the overall class proportions.

As a side issue, a related stratification difficulty is that much published work fails to mention what level of stratification is in play. As an anecdotal example, twelve out of nineteen related papers that were careful enough to do cross-validation investigations nonetheless failed to at all mention whether they were partitioning at the token, sequence, or document level.

2.3.4 Sequence Labeling

Another, and unexpected, complication induced by the “token vs sequence” tension is a lack of standardization around assessing the accuracy of labels, and indeed, of how to describe labeled “ground truth” in the first place.

Again, with token data, this is straightforward: every token has a label.

With NER text sequences, there is more complexity. Consider a simple case where the *only* named entity you are trying to label is PERSON. In this context, you might think to simply label every token individually as OTHER or PERSON, as being part of a name or not. But the point of labeling entities is usually to extract them for some other purpose. So if

The author was cited as Kegelmeyer, William.

was labeled as

The/O author/O was/O cited/O as/O Kegelmeyer/PER ,/O William/PER ./O

the “Kegelmeyer/PER ,/O William/PER” sequence refers to a *single* person.

However, in

I/O saw/O Peter/PER ,/O Paul/PER and/O Mary/PER ./O

the structurally identical “Peter/PER ,/O Paul/PER” sequence refers to two distinct people.

Thus many people use distinct labels for the beginning and interior of a specific label; this is often called IOB format, for “Inside Outside Beginning”. So the cases above would be correctly and distinctly truthed as

The/O author/O was/O cited/O as/O Kegelmeyer/B-PER ,/I-PER William/I-PER ./O

I/O saw/O Peter/B-PER ,/O Paul/B-PER and/O Mary/B-PER ./O

A final nuance to look out for is the literature that doesn’t explicitly label the “OTHER” tokens. They are simply ignored, left unlabeled, in a sparse description of the sequence labels. This can be *logically* equivalent to an explicit “OTHER” label, but it can also lead to ambiguities in accuracy evaluation (Section 2.3.5).

For this project, we adopted the IOB formalism with explicit labeling of “OTHER” tokens, including the labeling of punctuation as being just another token. This is partly because it can be surprisingly difficult to automatically distinguish between punctuation and non-punctuation, especially in informal text. Another reason is that we adopted the practice of truthing punctuation embedded in a named entity as being I-XXX, as opposed to OTHER. Examples:

Tim/B-PER ’/I-PER s/I-PER message/O to/O

Tina/B-PER Eliassi/I-PER -/I-PER Rad/I-PER said/O ...

2.3.5 Performance Metrics

A complication induced by IOB labeling, however, is what counts as “correctly” detecting an entity. Unfortunately, much published work is not clear on this. It would seem unambiguous — was the correct label assigned or not? — but there are possible subtleties. In particular, there were a series of “Message Understanding Contests” [8] devoted to assessing progress in NER, and they eventually implemented a complex metric that separately assessed the ability of an NER system simply to spot the right sort of entity (and thus ignored beginning/internal distinctions) and the ability to get the entity boundaries right.

In this project we avoided that distinction as unhelpful, and instead assessed our accuracy (via overall accuracy, precision, recall, or F1) on whether we assigned the correct label to each token.

However, a final ambiguity remains. In almost every text data set, the OTHER label vastly outnumbers all of the other labels. As a result, one could score deceptively well simply by assigning the OTHER label to everything.

Accordingly, we settled on a default performance metric of “aggregate Not-O F1”. That is, we count the number of true positives, true negatives, false positives, and false negatives induced by all of the labels except “OTHER”: the “Not-O” labels. Then we *sum* (not average) them, compute precision and recall, and thus F1, as per usual¹.

2.4 CRFs and Ensembles

Given the complications reviewed above, it is perhaps unsurprising that there was scant prior work combining ensembles and CRFs. One approach looked at combining small numbers of base NER classifiers, treated as black boxes, by estimating and exploiting their respective conditional probability distributions as a function of local text [21]. Another paper took a similar approach, but instead conditioned on the label [11]. Both were interesting, but not useful to NEEEEEIT, in that we were interested in the internals of CRFs, and understanding them and ensembles well enough to adapt them directly to ensemble methods.

In a non-text image analysis context, CRFs are sometimes used to model and detect various image features. One investigation [37] looked into building a variety of CRFs, each dedicated to finding a specific image feature, and then combined them by building a master CRF to interpret their feature recognition outputs. This would be distantly akin to our building a very small number of linear CRFs, each devoted to detecting only one label type, and each thus working from a distinct set of token features. So again, not directly relevant to our goal.

¹http://en.wikipedia.org/wiki/Precision_and_recall#Definition_.28classification_context.

3 Data

3.1 Corpus Interface

Because we needed to work with a wide variety of data sources, one of the earliest tasks for the project was the creation of a *corpus* API that would provide a consistent abstract interface when accessing the contents of a labeled document corpus, regardless of source. We used that API to create various concrete implementations of the corpus interface. These acted as data-set-specific “adapters”, making the underlying data accessible in a consistent way. Over time the interface grew to provide a rich set of accessor functions for every facet of a corpus, including tokens, sequences, labels, features and metadata, all discussed in the sections to follow.

3.1.1 Tokens

Content to be used for training and evaluation from the underlying data source was represented by the corpus as a sequence of string *tokens*, where each token was a collection of one or more characters that were significant as a group. For the English-language content used in this project, a token was typically — though not necessarily — a single word or a single punctuation mark.

In most cases the corpus implementations were responsible for tokenizing the content of the underlying data sources using separate tokenization strategy objects, allowing the caller to experiment with alternate tokenization algorithms. Two algorithms were used for the experiments in this report: a Penn Treebank tokenizer [28] from the Python NLTK toolkit [3] and a custom tokenizer for email documents [13], designed for use with NLTK.

For many of our corpora, the tokens represented a *subset* of the full content of the underlying data source: for example, with email corpora, the tokens were extracted from the body of each email message, leaving out any email header information (though the headers were used elsewhere, see 3.1.5). Similarly, some of our corpora that were based on newspaper articles omitted article metadata and headlines from their token content, including just the body text.

3.1.2 Sequences

As outlined in 2.3.2, a key feature of CRF classifiers is that they are trained and evaluated, not on observations, but on *sequences* of observations. This meant that the corpus needed to represent the underlying data source as a collection of *sequences* of tokens, not just a collection of tokens. Further, the *types* of sequences could vary, depending on the needs of the experimenter and the underlying data. For example, the documents in a corpus naturally partition the tokens into a set of contiguous sequences. Those document sequences might be further split into chapters, paragraphs, sentences, phrases, and so on. To accomodate this potential diversity of sequences, each corpus stored one or more sequence types, identified by name. Each sequence type contained a collection of sequences, each of which was a half-open range of contiguous token indices.

In practice, most of our corpora provided per-document and per-sentence sequence types, with the per-sentence sequences used for a majority of our experiments. To generate per-sentence sequences, we used a Punkt sentence tokenizer [30], also part of NLTK.

3.1.3 Labels

Each corpus contained a set of per-token “ground-truth” labels, used to train new classifiers and evaluate the performance of existing classifiers. The labels were stored in two parts: a sequence of N zero-based integer label indices that represented the label for each of the N tokens in the corpus, and a *label dictionary* that mapped the label indices to more semantically meaningful label strings.

The set of available labels was determined by the contents of the underlying data source: some sources defined labels for a variety of named entity types (Facility, Geopolitical Entity, Location, Organization, Person) while others were more limited (Person). In all cases a special “Other” label was assigned to tokens that would otherwise be unlabeled.

In most cases, the labels in the underlying data were supplemented with Inside-Outside-Beginning (IOB) formatting to highlight the boundaries between adjacent label, for the reasons discussed earlier in Section 2.3.4.

3.1.4 Features

As with tokenization, each of our corpus implementations was responsible for extracting *features* from every observation (token). To accomodate the wide variety of possible features that could be extracted from a corpus, we implemented a broad set of *feature extraction strategies* that could be mixed and matched to suit the needs of any given corpus or experiment.

Conceptually, each feature extraction strategy was a function that accepted a sequence of tokens and a sequence of already-extracted feature vectors as inputs; each strategy implementation was responsible for iterating over the tokens and growing the feature vectors accordingly. This approach made it possible to write feature extraction strategies that incorporated previously-extracted features into their algorithms, if desired.

During feature extraction we represented the feature vector for an observation as a dictionary of key-value pairs, combining a key to identify the *feature instance*, with a potentially arbitrary *feature value*. In practice, we extracted only sparse boolean features for this project, so the feature value for every extracted feature instance was “True”. Using sparse boolean features in this way was a natural fit for features such as “IS-CAPITALIZED” which were simply true or false for a given token, and it was still possible to extract categorical (multi-instance) features. For example, a feature “TOKEN-LENGTH” — which could take on multiple values, one for every possible token length in a corpus — was handled by extracting a distinct feature instance for every possible feature value. In this case, we incorporated the value into the feature instance key using string substitution: so we extracted a boolean feature “TOKEN-LENGTH(3)” for all length-3 tokens, a

separate feature “TOKEN-LENGTH(4)” for all length-4 tokens, and so on. Thus, a single feature extraction strategy could extract a family of multiple related feature instances. In this case we referred to the collection of related instances as a *feature class*.

Many of our feature extraction strategies were parameterized, allowing a single implementation to extract multiple feature classes. For example the “inside” strategy, which extracted a boolean feature for every token that contained a parameterized character of interest, was used to extract feature classes “INTERNAL-PERIOD”, “INTERNAL-APOSTROPHE”, “INTERNAL-HYPHEN”, and “INTERNAL-AMPERSAND” by simply specifying the feature instance key and search character when instantiating the strategy. A strategy using regular expression matches was used to extract several distinct feature classes, as was a strategy that matched tokens to lists of values.

A special “combination” strategy allowed us to create strategies that were compositions of individual strategies. Using this mechanism, we created a “standard” collection of strategies that was used for a majority of our experiments. This standard set of strategies extracted the following features:

IS-CAPITALIZED Tokens that were capitalized (first character is upper-case, followed by one or more lower-case characters).

IS-UPPER-CASE Tokens that contained all upper-case characters.

IS-LOWER-CASE Tokens that contained all lower-case characters.

IS-MIXED-CASE Tokens that contained one or more upper-case and one or more lower-case characters.

ENDS-WITH-PERIOD Tokens that ended with a period.

INTERNAL-PERIOD Tokens that contained (but did not begin or end with) a period.

INTERNAL-APOSTROPHE Tokens that contained (but did not begin or end with) an apostrophe.

INTERNAL-HYPHEN Tokens that contained (but did not begin or end with) a hyphen.

INTERNAL-AMPERSAND Tokens that contained (but did not begin or end with) an ampersand.

ENDS-WITH-IST Tokens that ended with “ist”.

ENDS-WITH-ISH Tokens that ended with “ish”.

ENDS-WITH-AN Tokens that ended with “an”.

ENDS-WITH-EX Tokens that ended with “ex”.

ENDS-WITH-TECH Tokens that ended with “tech”.

ENDS-WITH-SOFT Tokens that ended with “soft”.

ALL-DIGITS Tokens that only contained digits.

TWO-DIGITS Tokens that contained exactly two digits.

FOUR-DIGITS Tokens that contained exactly four digits.

CARDINAL Tokens that matched a list of cardinal numbers (“one”, “two”, “twenty”, etc).

ORDINAL Tokens that matched a list of ordinal numbers (“first”, “second”, “twentieth”, etc).

ROMAN Tokens that were Roman numerals.

WORD-WITH-DIGITS Tokens that contained one or more digits and one or more alpha characters.

TITLE Tokens that matched a list of standard English titles (“Mr”, “Mrs”, “Dr”, “Cmdr”, etc).

PROPER-NAME Tokens that matched a list of popular proper names.

POSSESSIVE-PRONOUN Tokens that matched a list of English possessive pronouns (“mine”, “yours”, “hers”, etc).

PERSONAL-PRONOUN Tokens that matched a list of basic English personal pronouns (“I”, “you”, “she”, etc).

FIRST-PERSON-PRONOUN Tokens that contained the first person pronoun “I”.

P3(prefix) Extracted the first three characters of a token as a feature. Ignored tokens with less than three characters.

S3(suffix) Extracted the last three characters of a token as a feature. Ignored tokens with less than three characters.

N3(trigram) Extracted trigrams from a token as features. Ignored tokens with less than three characters.

STEM(stem) Extracted a token stem as a feature, using an implementation of the Porter Stemming Algorithm [29] from NLTK.

ALPHA(characters) Created a feature from all of the alpha characters in a token.

NON-ALPHA(characters) Created a feature from all of the non-alpha characters in a token.

UPPER-CASE(characters) Created a feature by converting every character in a token to upper-case.

LOWER-CASE(characters) Created a feature by converting every character in a token to lower-case.

PATTERN(pattern) Converted each token into a pattern feature by mapping every upper-case character to “A”, every lower-case character to “a”, every digit to “0”, and all other characters to “-”. For example “Timothy’s” would become “PATTERN(Aaaaaa-a)”.

SUMMARIZED-PATTERN(*pattern*) Like PATTERN, but with runs of identical pattern characters compressed to a single character, so “Timothy’s” would become “SUMMARIZED-PATTERN(Aa-a)”.

TOKEN-LENGTH(*length*) Created a feature from the length of a token.

PREV-TOKEN-LENGTH(*length*) Created a feature from the length of the previous token.

NEXT-TOKEN-LENGTH(*length*) Created a feature from the length of the following token.

TOKEN(*token*) Created a feature from the token.

PREV-TOKEN(*token*) Created a feature from the previous token.

NEXT-TOKEN(*token*) Created a feature from the following token.

FIRST-TOKEN Matched the first token in the corpus.

LAST-TOKEN Matched the last token in the corpus.

In addition to the standard feature extraction strategies itemized above, most corpus implementations extracted special *sequence features* that identified a token as being at the beginning or end of a particular sequence, e.g: “DOCUMENT-BEGIN”, “DOCUMENT-END”, “SENTENCE-BEGIN”, “SENTENCE-END”, and-so-on for every sequence type supported by the underlying data.

As a special-case for email corpora, we extracted a special set of features identifying tokens that appeared in the email headers for a specific document, along with the header type:

TOKEN-IN-HEADERS(*token*) Created a feature for every token that appears as a value anywhere in the document email header data.

TOKEN-IN-*header*(*token*) Created a feature for every token that appears as a value for that specific document email header.

For example, in an email document with a subject line of “Birthday party for Tim”, the token “Tim” in the body of the document would include the features “TOKEN-IN-SUBJECT(Tim)” and “TOKEN-IN-HEADERS(Tim)”.

Once the feature extraction process was complete, we computed a *Feature Dictionary* containing the set of all unique feature instances, replacing the string keys in our feature vectors with integer indices referencing the feature dictionary.

3.1.5 Metadata

Each corpus provided a mechanism to access metadata extracted from the underlying data source. The metadata mechanism was used to store arbitrary sets of named metadata on a per-sequence basis, for any of the sequences in the corpus. As an example, some corpora — whose underlying data consisted of one file for each corpus document — stored the name of each file as per-document metadata.

Our email corpora relied heavily on this mechanism to store data contained in email headers — which was *not* treated as token content — using per-document metadata. Examples of email metadata included common headers such as “FROM”, “TO”, and “SUBJECT”, along with a sparse set of less-common headers specific to particular email servers and clients.

3.1.6 Threads

Some of our experiments involved training classifiers using methods that were aware of document relationships, such as threaded conversations of email messages. For those data sets that supported the notion of threaded conversations, the corpus interface incorporated support for storing optional *Thread Graphs*. Each thread graph in a corpus was named, associated with a specific sequence type, and represented a directed acyclic graph of parent-child relationships between sequences of that type. Thread graphs were stored as $N \times N$ sparse matrices containing nonzero values for every element ij where sequence i was a child of sequence j .

For example, a corpus of email messages might have a thread graph that stored relationships between document sequences (messages), representing a reply message as a child of an original message. Using the graph in this fashion, we were able to model reply messages, forwarded messages, replies-to-replies, etc. Although we did not work with other threaded conversation types, this mechanism could have been used to represent a wide variety including blog postings, hierarchies of online comments, twitter conversations, etc.

Once we had a concrete representation for threaded conversations, we could explore how to make our classifiers “thread aware”. One approach was to use the thread graph to supplement our “normal” features with a set of *thread aware features*. Note that because our thread aware features applied to entire sequences rather than individual tokens (i.e. “document 5 is a child of document 2”), we added each thread aware feature for a sequence to the feature vectors for every token within the sequence:

THREAD-ROOT Documents that began a thread, i.e. had no parent.

THREAD-REPLY Documents that were replies to another document, i.e. had a parent.

THREAD-LEAF Documents that ended a conversational thread, i.e. had no children.

THREAD-SIZE(count) Created a feature based on the size of (number of messages within) a thread.

THREAD-PATH(*path*) Created a feature that encapsulated a document’s path relative to the root of the conversation. For example, the first reply to the second reply to the beginning of a conversation would have feature “THREAD-PATH(/2/1)”.

See 5.2.1 for an alternate mechanism that we used to incorporate threaded conversation information into our experiments.

3.1.7 Implementations

Over the course of the project, we developed several implementations of the corpus API, to read the following types of data:

TaggedSentence Converted a sequence of tagged sentences into a corpus, typically for testing and debugging.

LDC Converted labeled data in LDC SGML format [35] to a corpus.

MinorThird Converted labeled data in MinorThird format [9] to a corpus.

UnlabelledEmail Converted a filesystem directory containing email messages into an unlabeled corpus.

3.1.8 Frozen Data

Although the corpus interface provided a powerful and flexible abstraction for accessing a wide variety of data sets, performance was still a concern. Because our experiments on ensembles explicitly meant generating tens of thousands of classifiers over the course of the project, we knew it would be important to amortize the substantial computation needed to tokenize, identify sequences, and and extract features from a corpus. Working with corpus instances that repeated those computations every time we generated a classifier or re-started an experiment would have been a significant waste of resources.

Instead, we implemented a process to *freeze* the contents of a corpus to a filesystem, along with a special corpus implementation that could load the the frozen data. The layout and format of the frozen data on disk was chosen to maximize load performance, so the data could be loaded and re-loaded quickly, as often as necessary, with a minimum of computation.

Thus, in practice we created corpus implementations for each of our data sets, used them once to extract the required data, froze the contents, then used the frozen corpora for the majority of our experiments. Because all of our corpus implementations were parameterized to some degree, we often created more than one frozen corpus based on a single corpus implementation, run with different parameters.

Figure 1. Manually labeling corpus tokens in a web browser.

3.1.9 Manual Labeling

In some cases we needed to work with data sets that didn't already include ground truth labels. To facilitate adding our own labels, we created a simple client-server tool based on web technologies for the task. Users would choose a corpus to edit, start a special web server, then point a web browser at the server to view the corpus (Figure 1). After choosing the type of label set and sequences to edit, the user was able to move back and forth through sequences, seeing the sequence tokens, color-coded by label type. Clicking on a token provided a pop-up menu of label choices, and keyboard shortcuts were provided to make the work faster. Using the manual labeling tool, we annotated several otherwise unlabeled email corpora culled from our own inboxes and loaded using the UnlabelledEmail corpus implementation.

As a sanity-checking tool for use in conjunction with the manual labeling tool, we implemented a script to identify unexpected label transitions in a corpus' ground truth. Unexpected label tran-

sitions were transitions that should never occur in correctly-labeled data, such as a transition from “O” to “I-PER” or “B-LOC” to “I-PER” — since every named entity must have a B- label for its first token, and every I- label must be preceded by a B- label of the same type.

3.2 Data Sources

Some notable corpora that we used over the course of the project were as follows:

ENRONRANDOM A collection of 591 internal email messages (with 6502 sentences and 145391 tokens) made public after the collapse of the Enron corporation, with Persons labeled. [15]

LDCENGLISHNWIRE A subset of 105 English-language news articles (with 3000 sentences and 66951 tokens) from the TIDES Extraction 2003 Multilingual Training Data set, with Persons, Locations, Geopolitical Entities, and Organizations labeled. [26]

NGC300 A collection of 300 email messages (with 4489 sentences and 113057 tokens) from an internal project mailing list that were either sent by or addressed to one of three specific people, with Persons labeled using our manual labeling process, dedicated to training.

VNGC92 A collection of 92 different email messages (with 2068 sentences and 45661 tokens) fitting the same criteria as NGC300, and used exclusively for validation.

NEWSWIRET, NEWSWIREV A partitioning of the LDCENGLISHNWIRE corpus into 57 and 48 document subsets, chosen with greedy stratification to balance the counts for each label type as closely as possible for each set.

4 pyCrust

pyCrust is a Python software package for training and evaluating conditional random field (CRF) models. It features a modular design separating the CRF model data structure from the training and evaluation functionality. Furthermore, it is agnostic with respect to the meaning and/or context of the application features that it is modeling. This means that it can be used as a generic CRF model for a variety of domain specific applications, as it does not require any understanding of the information owned or contained in those applications.

This section describes the general CRF model, the use of existing CRF model codes in this project, and our motivation for implementing a new CRF model. It also details the design choices made in implementing pyCrust CRF models.

4.1 The CRF Model

CRFs are a type of discriminative undirected probabilistic graphical model first introduced for segmenting and labeling sequential data [19]. Graphical models are a useful family of models for modeling and analyzing relationships in data [17, 2]. Many well-known models and analysis approaches fall into this family (see Fig. 2 for a depiction of its current extent), including naive Bayes classifiers, logistic regression models, and hidden Markov models (HMMs). Several papers in the literature have identified the relationship among these models, CRFs included (see, e.g., [12, 16]).

The typical application of CRFs for sequence labeling, and the focus of this project, is the *linear chain CRF*. The term “linear chain” refers to the dependency relationship of the label sequence only in the model, as in most general CRF models, probabilities of labels are often conditioned on observations (and associated features of those observations) at any position in the sequence.

For a given sequence of T observations (e.g., words), $\mathbf{x} = \{x_1, x_2, \dots, x_T\}$ and associated labels (e.g., named entity labels), $\mathbf{y} = \{y_1, y_2, \dots, y_T\}$, the linear chain CRF model is given as

$$p_{\lambda}(\mathbf{y}|\mathbf{x}) = \frac{1}{Z_{\lambda}(\mathbf{x})} \exp \left\{ \sum_{t=1}^T \sum_{j=1}^J \lambda_j f_j(y_{t-1}, y_t, x_t) \right\}, \quad (1)$$

where $\{f_j\}_{1 \leq j \leq J}$ is an arbitrary set of J binary feature functions, $\{\lambda_j\}_{1 \leq j \leq J} \in \mathbb{R}$ the set of J model parameters, and $Z_{\lambda}(\mathbf{x})$ is a normalization factor. This normalization to $[0, 1]$ is achieved using a sum of weighted features over all possible label sequences, $\mathcal{Y}$, given by

$$Z_{\lambda}(\mathbf{x}) = \sum_{\mathbf{y} \in \mathcal{Y}} \exp \left\{ \sum_{t=1}^T \sum_{j=1}^J \lambda_j f_j(y_{t-1}, y_t, x_t) \right\}. \quad (2)$$

Note that the feature functions in linear chain CRFs are often binary functions, but could be continuous functions as well; for example, one may wish to include a confidence or importance

Figure 2. A taxonomy of the current family of graphic models (excerpted from [2]).

score for each feature instance, and continuous feature functions could be used for such cases. For linear chain CRFs the feature functions are often chosen to model either transitions or particular states (label/observations pairs). To illustrate the difference, the following are examples of binary transition and state feature functions for the second word in the phrase “Balthazar is funny” with labels being part of speech (NOUN, VERB, etc.):

$$\textbf{transition: } f_j(y_0, y_1, x_1) = \begin{cases} 1, & \text{if } y_0 = \text{NOUN and } y_1 = \text{VERB and } x_1 = \text{IS-LOWERCASE} \\ 0, & \text{otherwise} \end{cases}$$

$$\textbf{state: } f_j(y_0, y_1, x_1) = \begin{cases} 1, & \text{if } y_1 = \text{VERB and } x_1 = \text{IS-LOWERCASE} \\ 0, & \text{otherwise} \end{cases}$$

So this sample transition feature would have value “1” only at the moment that x_1 is the second token, “is”, and if the sentence labels have been correctly assigned as “NOUN VERB ADJ”.

Although the transition and state feature functions have the same functional form, no information regarding the previous state is used in state feature functions. Using the same notation for both types of functions is convenient when defining the CRFs and is prevalent in the literature, and the authors here caution readers of papers on CRFs to be clear about the exact types of functions included, as it is not always apparent. In pyCrust, we refer to transition and state feature functions as *bigram* and *unigram* feature functions, following [20].

Parameter estimation is achieved through the maximum-likelihood method, where training of the CRF model is performed by maximizing the log-likelihood $\mathcal{L}$ on training data $\mathcal{T}$:

$$\mathcal{L}(\mathcal{T}) = \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{T}} \log p_{\lambda}(\mathbf{y}|\mathbf{x}), \quad (3)$$

where $p_{\lambda}(\mathbf{y}|\mathbf{x})$ is defined in Eq. (1).

In the original work on CRF models [19], the authors suggested the use of the Improved Iterative Scaling (IIS) algorithm [10] for fitting the models parameters in Eq. 3, which had been successfully applied to parameter estimation problems in other applications involving other random field models. However, as CRF models were applied to larger data collections, large-scale optimization methods have shown the most promise to date (see descriptions and references in [20] for more information on approaches that have been considered).

The typical approach to parameter estimation is to minimize a regularized version of the loss function in Eq. 3. Below is an example of a loss function with an ℓ^2 regularization term added:

$$\mathcal{L}(\mathcal{T}) = \sum_{(\mathbf{x}, \mathbf{y}) \in \mathcal{T}} \log p_{\lambda}(\mathbf{y}|\mathbf{x}) + \frac{\rho_2}{2} \|\lambda\|_2^2. \quad (4)$$

Here λ is the vector of model parameters and ρ_2 is the regularization parameter controlling the tradeoff between fitting the model exactly to the training data and finding a set of smooth parameters. With the addition of the ℓ^2 regularization term, the loss function becomes a smooth convex function that can be minimized using standard large-scale optimization methods, such as limited memory Quasi-Newton methods (e.g., LBFGS [23]) or Stochastic Gradient Descent [4]. Along with ℓ^2 regularization, there have also been attempts to use ℓ^1 regularization in the loss function in attempts to find sparse sets of parameters that fit the CRF models to training data well [20].

Once a suitable set of parameters have been found to fit a CRF model to training data, the model is often then used to predict the most likely $\mathbf{y}$ given $\mathbf{x}$ for a new sequence of data. This is referred to as model inference or evaluation, and is typically determined using the Viterbi algorithm [31].

4.2 Existing CRF Model Implementations

At the start of the NEEEEEIT project, the team had developed some expertise in using the Stanford Named Entity Recognizer (SNER) [14]. However, some effort was put into assessing different CRF implementations to determine which would be most amenable to 1) research on ensemble methods, and 2) useful in the initial Python-based data processing and experimentation framework developed for the project. In this section, we describe the main CRF model implementations considered for use in NEEEEEIT.

CRF++². The CRF++ package is a C++ implementation of a CRF model. Although it appears efficient and has been demonstrated to perform well on a standard benchmark data set as reported

²<http://crfpp.sourceforge.net>

by the authors, the API for feature specification was onerous and cryptic. Thus the NEEEEEIT team decided against using this package in the project. However, a port of CRF++ to Java was implemented by another SNL project, and the NEEEEEIT team explored the use of that implementation, as it was being used for NER in SNL applications. However, as an implementation for research purposes, it seemed that using CRF++ (either implementation) would have required hard-coding of specific features to be used in the CRF models, and this was deemed too great a limitation when exploring new ideas.

SNER³. SNER is a Java-based implementation of a a CRF model specific to the task of named entity recognition [14]. Although the NEEEEEIT team had prior experience with SNER and it had a rich set of features for use with NER tasks, the API did not support as many of the data processing and transformation methods as the Mallet implementation, discussed next. Thus, we chose not to continue with SNER for the NEEEEEIT project.

Mallet⁴. The MACHine Learning for Language Toolkit (Mallet) is a Java-based package for statistical natural language processing originally developed in Andrew McCallum’s research group at UMass Amherst [24]. Mallet contains an implementation of a linear chain CRF model, and there is a supplemental package called GRaphical Models in Mallet (GRMM) [32] containing an implementation for arbitrary CRF models. As the NEEEEEIT project was interested in both linear chain and more general CRF models from the start, the choice of using Mallet seemed appropriate at the start of the project.

The main challenge for using Mallet in the NEEEEEIT project at first was bridging the gap between Python, which was used for the data processing and experimentation implementations, and Java, which was used for the Mallet implementation. After assessing the use of JPytype and Jython, the NEEEEEIT team decided to use Jython, based on 1) performance, and 2) the ability to directly use Java array data structures (which are heavily leveraged throughout Mallet) in Python.

Of the CRF model implementations explored by the NEEEEEIT team, Mallet seemed to be the most mature code base, contained the richest set of default features for the task of NER, and appeared to be computationally efficient for our initial test data. However, after developing an interface from the NEEEEEIT software framework to Mallet and experimenting with several ideas for extending single CRF models, we discovered that the specific Mallet implementation for single CRF models did not permit the degree of internal instrumentation and manipulation necessary to support an ensemble-focused research agenda. Further, the overhead in calling libraries and the lack of support for C extension modules in Jython (NEEEIT had dependencies on several C extension modules, including NumPy) slowed matters to the point where the parameter studies required for exploring ensemble performance were impossible. These reasons eventually led the team to explore a new CRF model implementation.

Wapiti⁵. Accordingly, about half-way through the NEEEEEIT project, we decided to develop our own Python implementation of a CRF model. Other researchers had begun efforts involving

³<http://crfpp.sourceforge.net>

⁴<http://mallet.cs.umass.edu/>

⁵<http://wapiti.limsi.fr/>

Python implementations of CRF models (e.g., Monte⁶, pyner⁷, CRFsuite⁸, etc.), but those efforts were either not fully developed to date, lacked sufficient documentation, or did not appear to be efforts that would be supported beyond the current funding of the associated project. Furthermore, no other research projects were investigating the use of ensembles, and thus the codes we examined were tuned toward single model generation and use, lacked API or instrumentation for timing and performance testing, and did not provide access to the internal data structures that the NEEEEEIT project needed to explore creating diverse CRF models across an ensemble.

During the development of pyCrust, we benefited from ideas described in [20] for implementing efficient CRF models. A C-based software package called “wapiti” was developed as part of the research efforts described in that paper. Our team used the wapiti code as a reference implementation of the ideas in [20], and for verifying models generated using pyCrust. Wapiti provides more algorithmic capabilities than pyCrust to date, but the feature templating mechanism in wapiti is onerous, and thus we decided to not adopt wapiti into the NEEEEEIT project. Still, future projects requiring Python-based CRF model implementations should consider the costs and benefits of adopting pyCrust versus wrapping wapiti via C extensions.

4.3 pyCrust Implementation Details

In this section, we present the implementation details of pyCrust. This may be important for those readers interested in using or extending pyCrust for their own applications. As mentioned above, the pyCrust implementation follows the CRF model descriptions presented in [20]. Table 1 presents the main components of the current implementation of pyCrust, which are described throughout the next sections. pyCrust is implemented as a Python module with several classes, a C++ extension module, and testing routines. In this section, the pyCrust module is contained in the top level directory called `pycrust`.

<code>crf.py</code>	CRF model
<code>weights.py</code>	model parameters
<code>data.py</code>	data container
<code>training.py</code>	model fitting methods
<code>inference.py</code>	model inference methods
<code>extension/</code>	C++ extension module for training methods
<code>testing/</code>	unit and performance test methods

Table 1. Main components of the pyCrust implementation.

Currently, there are several required package/module dependencies for pyCrust, including the

⁶<http://montepython.sourceforge.net/>

⁷<https://github.com/dat/pyner>

⁸<http://www.chokkan.org/software/crfsuite/>

following (version numbers are those that have been verified to date).

- Python Standard Library: `copy`, `ctypes`, `math`, `os`, `pickle`, `random`, `sys`
- `numpy`, v1.6.2: <http://www.numpy.org>
- `scipy`, v0.11.0: <http://www.scipy.org>

pyCrust testing is optional and requires the following Python Standard Libraries: `time`, `unittest`.

4.3.1 pyCrust Classes

The CRF model class, `crf`, defined in `crf.py`, is a very lightweight class, and simply serves as the primary container for pyCrust CRF models. The parameters of the CRF model are stored in the `weights` class defined in `weights.py`. The main reason for creating a separate class for the weights is so that the model parameters can be easily passed to a (possibly external) optimization method. The remaining main class is the `data` class defined in `data.py`.

CRF Class. This is a simple container class for a CRF model. It contains data members for the number of features and number of labels that are associated with the model, along with the parameters, or weights, of the model as specified in the `weights` class (see below).

There are several methods that take a pyCrust CRF model as input defined in the pyCrust module. For example, there are methods defined in `pycrust/__init__.py` for loading and saving CRF models (i.e., `crf` object), as well as for altering CRF models (e.g., zeroing out model parameters with magnitudes below a threshold value) that were used in support of some of the the NEEEEEIT ensemble experiments.

Weights Class. The `weights` class is a container class for the CRF model parameters. The parameters are stored as a `numpy.array` object to support use of numerical optimization methods in `scipy`.

pyCrust follows [20] in support of unigram and bigram feature functions. Given F unique features (e.g., `TOKEN(Danny)`, `PATTERN(Aaaaa)`, `TOKEN-LENGTH(5)`, etc.) and L labels (e.g., `B-PER`, `I-PER`, `O`, etc.), there are $F \times L$ weights associated with the unigram features and $F \times L^2$ weights associated with the bigram features for a total of $F(L + L^2)$ weights for the linear chain CRF models in pyCrust. Thus, assuming a fixed-indexed feature dictionary, $\{f_0, f_1, \dots, f_F\}$ and a fixed-index label dictionary, $\{l_0, l_1, \dots, l_L\}$, `weights` are indexed by increasing order of the feature dictionary index-label dictionary index (for unigram features) and then feature dictionary

index-label dictionary index-label dictionary index (for bigram features) as follows:

$$\begin{aligned}
w_0 &\equiv \text{feature} = f_0 \wedge \text{current label} = l_0 \\
w_1 &\equiv \text{feature} = f_0 \wedge \text{current label} = l_1 \\
&\vdots \\
w_L &\equiv \text{feature} = f_0 \wedge \text{current label} = l_L \\
w_{L+1} &\equiv \text{feature} = f_0 \wedge \text{previous label} = l_0 \wedge \text{current label} = l_0 \\
w_{L+2} &\equiv \text{feature} = f_0 \wedge \text{previous label} = l_0 \wedge \text{current label} = l_1 \\
&\vdots \\
w_{2L-1} &\equiv \text{feature} = f_0 \wedge \text{previous label} = l_0 \wedge \text{current label} = l_L \\
w_{2L} &\equiv \text{feature} = f_0 \wedge \text{previous label} = l_1 \wedge \text{current label} = l_0 \\
&\vdots \\
w_{(L+1)L-1} &\equiv \text{feature} = f_0 \wedge \text{previous label} = l_L \wedge \text{current label} = l_L \\
w_{(L+1)L} &\equiv \text{feature} = f_1 \wedge \text{current label} = l_0 \\
&\vdots \\
w_{F(L+1)L-1} &\equiv \text{feature} = f_F \wedge \text{previous label} = l_L \wedge \text{current label} = l_L
\end{aligned}$$

In the `weights` class, there are `numpy.array` objects that contain index offsets for each feature for the different types of feature functions (unigram and bigram), that can be used to efficiently traverse `weights` objects over the feature dictionary indices.

Data Class. For each data set to be used with a CRF model, either as part of training the model or inference of labels from a model, a collection of objects is required. This includes the information on sequence starting and ending indices, the list of features for each observation in each sequence, the list of labels for each observation in each sequence, and the numbers of features and labels in the feature and label dictionaries associated with the data set. In an effort toward a simple, extensible API for pyCrust, the required data for use with training and inference methods are bundled together in data objects. Thus, this class is simply a helper class to help simplify the pyCrust API in the long run.

4.3.2 CRF Model Training

Training of CRF models in pyCrust is accomplished using methods from the `scipy.optimize` package. Specifically, the `minimize` method from that package is used for minimizing $-\mathcal{L}(\mathcal{T})$, which is equivalent to maximizing $\mathcal{L}(\mathcal{T})$, where $\mathcal{L}(\mathcal{T})$ is defined in Eq. (3). Currently, only support for the “L-BFGS-B” method via `minimize` is implemented in pyCrust, as that method was

identified in the literature as the most effective of those available in the `scipy.optimize` package for training CRF models (see [20] and references within).

Although there is code in pyCrust to allow for the use of regularized loss functions as in Eq. 4, access to those loss functions is not currently accessible via the API, as testing did not show that the use of such loss functions in pyCrust led to improvement. Future work will attempt to understand if this is due to code correctness or a limitation in our testing.

As much of the CRF model training involves traversing weights, features, and labels for all of the observations in a training set, there are many nested loops in the training methods. The initial implementations of these methods in Python seemed slower than expected, so we implemented the main computational bottleneck, the computation of the loss function (and its gradient, which was required for the `scipy.optimize.minimize` methods), as a C++ extension module. Using the `ctypes` and `numpy.ctypeslib` modules, we were able to avoid deep copies of the model parameters and training data required in the extension implementation of the model loss function. The difference in the computational costs for the interpreted (Python-only) and compiled (C++ extension) implementations of the loss functions is dramatic. Table 2 presents the timing results of computing a single value of the loss function and its gradient for the LDCENGLISHNWIRE data source described in Section 3.2. Note that the compiled extension implementation is more than two orders of magnitude faster than the pure Python implementation. Note also that the small difference between CPU and wall times indicates that the times for the computation of the loss functions do not incur large overhead or latency costs. These results are similar to those for other data sets as well, where improvements in computation time of more than two orders of magnitude were seen.

Time elapsed	Interpreted	Compiled
CPU time	1137.54	3.01
Wall time	1137.60	3.01

Table 2. pyCrust timing results for computing the loss function and its gradient associated with the LDCENGLISHNWIRE data set using interpreted (Python-only) and compiled (C++ extension) implementations.

4.3.3 CRF Model Inference

Model inference is the process of determining the most likely $\mathbf{y}$ given $\mathbf{x}$ for a new sequence of data and parameterized CRF model. The process of inference also involves traversing model weights, along with features of the new sequence data analogous to model training. Thus, compiled C++ extension implementations of the inference methods are available in pyCrust. Although the computation times are still significantly different for the interpreted and compiled versions of the inference methods, the differences were at most one order of magnitude different (as opposed to two

orders for model training). This is most likely due to the extra nested loops in the computation of the loss function gradients, which are not required in model inference. More testing would help identify the specific differences.

4.3.4 Open Questions and Future Work

At the end of the NEEEEEIT project, the pyCrust package included data structures using efficient `numpy.ndarray` storage and methods that are also compatible with `scipy.optimize` optimization methods, included efficient C++ extension implementations of the most computationally expensive part of the CRF model training and inference tasks, and included a unit and performance testing framework based on the `unittest` Python Standard Library module. Although more work could be done to make the code more efficient, the current pyCrust implementation is ready for use on other sequence label tasks. As an example, Andy Wilson has demonstrated use of pyCrust on vehicle tracking stage identification problems for the FY13-FY15 PANTHER Grand Challenge LDRD project.

Several items that could be addressed in follow-up work on pyCrust, and its role in the broader problem of ensemble-based CRF modeling include the following:

- Creating an open source version of pyCrust. This would help serve the data analysis and machine learning research communities using Python by providing a tool that is not currently available, and indeed, this is very likely to happen.
- Consider extending pyCrust to include CRF models other than the linear chain model currently implemented. This would provide benefit to both practitioners as well as researchers interested in NEEEEEIT-like experimentation. Furthermore, it would provide an example of extending pyCrust to other sequential data modeling and analysis problem types besides those explored in the NEEEEEIT project.
- Consider extending pyCrust to include wrappers for wapiti; for example: by using Swig to automate much of the process. Although the future of wapiti is unclear to the NEEEEEIT team, it does contain capabilities that would be of interest for future research following the NEEEEEIT project. Example capabilities include state-of-the-art optimization heuristics designed specifically for probabilistic graphical models, inference methods that predict the k -most likely y rather than just the most likely (which would support an untested idea for introducing model diversity into ensembles by using less likely label sequences in ensemble inference and voting), and several alternatives to the standard exponential models used in CRFs.

5 NEEEEIT

5.1 Experiments

5.1.1 Growing Ensembles

Because nearly every NEEEEIT experiment involved creating ensembles of classifiers, and because training CRF classifiers was time consuming, it was important to have an ensemble generation process that was efficient and scalable. Fortunately, a majority of our experiments allowed us to train classifiers completely independently, making it practical to train them in parallel. To organize the ensemble training process, we chose to implement a coordinator-worker model in which a single coordinator process generated sets of training parameters and distributed those parameters to remote worker processes to train the individual classifiers.

Experimenters would begin by running the coordinator with command-line parameters to choose the number of worker processes, the corpus, the type of corpus sequences to use for training, the number of folds to create for cross validation, the number of classifiers to train per-fold, and a set of sampling strategies to control the various stages of ensemble generation. Using these parameters, the coordinator would run as follows:

First, the coordinator would start a set of (local or remote) running worker processes. For each fold in the experiment, the corpus was partitioned into training and test data, using the experimenter-supplied fold sampling strategy. Examples of fold sampling strategies included *k-fold* and *k-by-2* for cross validation, as well as a strategy to use all available data as training data, i.e. to disable cross validation. Note that at this stage the coordinator was sampling corpus *sequences*, typically sentences. For each classifier to be generated for the current fold, the coordinator would partition the training data into in-bag and out-of-bag sequences using another experimenter supplied sampling strategy, the most common being random sampling with replacement. Next, the coordinator would sample from the *features* for all corpus sequences, using experimenter-supplied feature sampling strategies. A wide variety of feature sampling strategies were available, including:

- retaining a fixed number or a percentage of randomly chosen feature classes across the entire corpus,
- retaining a fixed number or a percentage of randomly chosen feature instances across the entire corpus,
- retaining a fixed number or a percentage of randomly chosen features on a per-token basis,
- retaining individual per-token features with probability P ,
- and including or excluding feature instances whose keys matched a regular expression.

The experimenter could also choose to work with combinations of these strategies. The specification of corpus, in-bag sequences, out-of-bag sequences, training sequences, and selected features constituted a classifier-training *job* that was placed in a queue. The coordinator would remove jobs from the queue and distribute them to available workers in round-robin order until all jobs were distributed. The coordinator then waited for notifications from workers as jobs were finished, shutting down the workers and exiting once all jobs had been completed.

When started, each worker would wait for incoming jobs. Once a job was received, the worker would extract the in-bag sequences and feature samples from the specified corpus and train a new (Mallet or pyCrust) CRF classifier, saving a serialized version of the classifier to the filesystem.

Although the sampling workload for the coordinator process was relatively lightweight, we were concerned that the bandwidth to send jobs to workers and receive results would become a bottleneck. Therefore, we opted to use a shared filesystem to read and write jobs, so that “sending” a job from the coordinator to a worker simply required transmitting a filesystem path over the network. To ensure that workers could operate completely independently, without locking or other synchronization mechanisms, we developed a standardized *experiment directory* layout on disk, where each fold in an experiment had its own subdirectory, and each classifier within the fold had its own job subdirectory, with well-defined files containing the job specification and results. Thus, each worker could manipulate the contents of its assigned job directories without need for locking or fear of race conditions.

Although bagging (random sampling with replacement) was the most common form of sampling that we applied to token sequences, we did run some experiments using *ivoting*, which was an interesting special-case due to the complexity that it added to the system. At a high level, ivoting is a sampling technique where the choice of data to use training classifier n is based, in whole or in part, on how well classifiers 1 through $n - 1$, as an ensemble, were able to make predictions on that data. For example, if the ensemble to date performed particularly poorly when evaluated on specific sequences, ivoting might choose to train subsequent classifiers on those sequences in an attempt to boost the overall ensemble performance. This added significant complexity to the system because it meant that when ivoting was in effect, workers no longer simply trained classifiers, they also had to evaluate their performance. Similarly, the coordinator could no longer simply sample from the corpus and distribute jobs — it had to wait to receive performance results for one job before it could create the next job. This significantly reduced the parallelism in the system: without ivoting, an experiment training 50 classifiers per fold for 10 folds could, given sufficient resources, train every classifier simultaneously for 500-way parallelism. With ivoting, the same experiment could only train one classifier per fold at a time for 10-way parallelism.

Thus, to support ivoting it was necessary for workers to include code for making predictions with newly-trained classifiers. Because training classifiers was a relatively slow process and evaluation was extremely fast, we configured the workers to always evaluate each newly-trained classifier on its in-bag training sequences, on the out-of-bag sequences, if any, and on the test data for the fold, if any. The evaluation results for each set of sequences was stored as sequences of predicted labels as part of the classifier job directory.

Because we wanted to evaluate classifier performance against additional data from other cor-

pora later-on, workers also serialized each classifier to the job directory, so that it could be loaded and used to make predictions at a later time.

Overall, the use of a coordinator process made it easy to centralize the logic that defined the contents of an ensemble, while the worker processes could focus on training and evaluating classifiers in parallel. During development and testing we could run a coordinator and a handful of workers on a single workstation, while larger experiments ran coordinators and large numbers of workers on separate nodes in a computing cluster. We used the Python “execnet” library to handle interprocess communication between the coordinator and workers.

One of the challenges we faced in running experiments was that the majority of our code — including the pyCrust CRF implementation — was written in Python, while the Mallet CRF implementation that we used for baseline experiments was written in Java. Fortunately, we were able to use “Jython”, an implementation of the Python programming language running in a Java Virtual Machine, to integrate the two bodies of code. Using Jython, we were able to call the Mallet Java code directly from our Python experiment framework, the only caveat being that we had to use the *jython* interpreter instead of the normal *python* interpreter to run any Mallet-related code. Our division of labor into coordinator and workers made this process easier, as only the worker code for training and evaluating classifiers needed to interact with Mallet, allowing for a clean separation between the bulk of our experiment code and some narrowly-defined “back-end” code that had to call Mallet and run within Jython.

5.1.2 Evaluating Ensembles

In summary, the process for training a classifier was as follows:

- Load a corpus.
- Select a subset of corpus sequences for training.
- Select a subset of features from the corpus sequences.
- Train a classifier, using the selected sequences, labels, and features.
- Save the classifier to disk.
- Make predictions using the classifier, and save them to disk.

Once the framework was in place for training ensembles of classifiers and serializing the classifiers to disk for later use, it was possible to begin evaluating those classifiers on new data, especially new corpora. Because our worker processes already contained functionality for making predictions with a classifier (to support ivoting, as described in [5.1.1](#)), we were able to treat ensemble evaluation as a special case of ensemble creation, one where classifiers were simply loaded from disk instead of trained from scratch:

- Load a corpus.
- Select a subset of corpus sequences for evaluation.
- Select a subset of features from the corpus sequences.
- Load an existing classifier from disk.
- Make predictions using the classifier, and save them to disk.

Thus, with some additional parameterization of our jobs (i.e. instructions to load an existing classifier instead of training a new one) we were able to implement ensemble evaluation using our existing workers.

One subtlety that cropped-up when evaluating classifiers trained on one corpus with data supplied by a different corpus was *feature alignment*. With the wide variety of features extracted by our system it was a given that different corpora would contain different sets of features. For example, an evaluation corpus might contain features that didn't appear in the training corpus. These features were therefore uninterpretable, and so they were discarded. Even features that did appear in both corpora might have been extracted in a different order, leading to different indices in their respective feature dictionaries. Thus, during evaluation it was necessary to map the features of the corpus being evaluated into the feature space of the corpus used to train the classifiers.

Despite these minor differences, the experiment directories created by evaluation were essentially identical to the experiment directories created during training. Thus, we were able to use a majority of our postprocessing and visualization tools for both types of experiment.

5.2 Postprocessing

Once we had experiment directories containing ensembles of trained classifiers and label predictions, we could perform postprocessing tasks: evaluating the performance of an ensemble, visualizing the structure of the classifier models, and more.

5.2.1 Voting

Evaluating the performance of an ensemble was our most important postprocessing task, one that hinged on our ability to combine label predictions from multiple classifiers into a single ensemble prediction by *voting*. Voting was the process by which predictions from multiple classifiers, each with a different perspective on the training data, were synthesized into an overall ensemble prediction that would (hopefully) provide better performance than a single classifier alone. We created an abstract voting API that allowed us to implement a variety of voting strategies for testing. During some early experiments we found that time spent voting could be a significant bottleneck: because we wanted to see how the performance of an ensemble changed as it grew (as new classifiers were

added), n separate votes were computed for each ensemble of n classifiers; often, this process was repeated three-or-more times to compare the performance of different voting strategies.

Accordingly, we adopted a stateful voting strategy API that improved the performance of our voting implementations by allowing them to cache information and keep running counts as they were called repeatedly with new sets of classifier predictions. Callers could create an instance of a voting strategy, then alternate adding new sets of predictions to the strategy and retrieving the “current” voting results. Typically, we added predictions and retrieved votes for every token in a corpus at-once, so it was necessary for voting strategies to accept sparse predictions and produce sparse results. For example, a given token might receive out-of-bag label predictions from one third of the classifiers in the ensemble, with nonexistent or “null” predictions from the remaining two thirds, and voting strategies had to handle these nulls in a logically consistent way.

By project’s end, we implemented the following straightforward voting strategies:

FirstChoice A debugging and profiling voting strategy that always returned the first prediction for every token, ignoring any subsequent predictions.

SimpleMajority Voting strategy that picked predictions based on a simple majority for every token. In the case of ties, returned the tied prediction with the largest index.

SimpleMajorityRandomTie Voting strategy that picked predictions based on a simple majority for every token. In the case of ties, chose one of the tied predictions at random.

UnanimousOther Voting strategy that chose “Other” only if every prediction was “Other” or the next most popular prediction was below a threshold; otherwise, picked the next most popular prediction.

We also implemented a set of “thread-aware” voting strategies. These strategies, when applied to predictions on a corpus that had information on conversation threads (such as email message replies), were able to use that information to influence voting:

SimpleThreadMajority Voting strategy that forced identical tokens within a conversational thread to receive identical labels, based on a simple majority of predictions. In the case of ties, chose the prediction with the lowest index. Used a case-insensitive test to identify “identical” tokens.

WeightedThreadMajority Voting strategy that picked predictions based on a weighted majority of predictions for identical tokens within a conversational thread. An inverse distance weighting strategy was used to weight a token’s predictions based on its breadth-first-search distance from the token being voted. We used a case-insensitive strategy to identify which tokens in a thread were considered identical.

5.2.2 Performance Metrics

Once we were able to combine individual classifier predictions using voting, we could begin to compute performance metrics on our experiments. The basis for a majority of our performance metrics was a diversity matrix class that, given a set of ground truth labels and a set of predicted labels (which might be predictions from a single classifier, or voted predictions from an entire ensemble), could compute per-label counts of true positive, false positive, and false negative results. These metrics were used in-turn to compute derived metrics such as per-label accuracy, precision, recall, and f-measure, along with twice-derived metrics such as mean accuracy, mean precision, mean recall, and mean f-measure, each computed over arbitrary sets of labels. Finally, we implemented aggregate precision, aggregate recall, and aggregate f-measure, also computed over arbitrary sets of labels. Computing these means and aggregates using caller-supplied label sets allowed us to look at a wide variety of performance scenarios, from individual labels to arbitrary groups of labels, such as the “Aggregate Not-O F1” metric (i.e. the aggregate F1 measure of every label except “Other”) that is used widely throughout the rest of this paper.

In addition to the confusion-based metrics, we also implemented a set of *diversity* measures to capture how much diversity (i.e. how much difference of opinion among classifiers) was being generated by our ensembles. Using a set of ground-truth labels and sets of predicted labels from multiple classifiers, we computed four pair-wise parameters between each pair of classifier predictions. These parameters were combined in-turn to compute mean accuracy, mean error, disagreement measure, double fault, q statistic, correlation coefficient, and pairwise kappa measures [18] for each pair of classifiers. Because the computational cost of computing these measures grew exponentially as the number of classifiers in an ensemble increased, we also provided options to compute the measures on a random subsampling of classifiers.

An example of the analysis permitted by diversity plots is in Figure 3. Each subplot represents building a “random subspace” (Section 6.1.4) ensemble of Mallet CRFs, using the indicated percentage of the feature classes; the top left subplot used only 5% of the feature classes, for instance. The expected behavior is observed: as more and more of the feature classes are incorporated in each individual CRF, the CRFs become less diverse, more similar, and so more tightly grouped on the x-axis. Further, the center of the cluster moves down in accuracy as the diversity decreases, which is again the usual expectation: less diverse ensembles tend to be less accurate. The value in plots such as this is that they permit straightforward visual assessment as to when a parameter study is nudging the ensemble process in a useful direction.

5.2.3 Analyses

With robust support for voting and performance metrics, we could begin to do true analysis of our results. Over the course of the project we created a plethora of postprocessing scripts, many of them highly specialized or one-offs to support short term goals or immediate needs. A few of those scripts become workhorses that we used on a regular basis, two prominent examples of which follow:

Random Subspace: Diversity

Figure 3. Accuracy vs Pairwise Kappa as a Function of Random Subspace Size

Our most widely used postprocessing script was the “print-cross-validation” script, used to summarize the performance of an ensemble across multiple cross validation folds as-follows: for each fold within an experiment, use a user-specified voting strategy to combine the predictions from each classifier within the fold, and compute per-label performance metrics using the voted predictions and ground truth labels. Then, aggregate the per-fold performance metrics to compute per-label mean and standard deviation statistics across the folds. The results were presented to the user in tabular format. As an example, an ensemble of 10 cross validation folds of 50 classifiers each might be summarized by print-cross-validation as follows:

label	mean(precision)	std(precision)	mean(recall)	std(recall)	mean(f1)	std(f1)	mean(accuracy)	std(accuracy)
O	0.990	0.002	0.993	0.002	0.991	0.002	0.983	0.004
B-FAC	0.625	0.518	0.174	0.238	0.484	0.241	0.174	0.238
B-GPE	0.888	0.044	0.892	0.027	0.890	0.030	0.802	0.049
B-LOC	1.000	0.000	0.519	0.268	0.713	0.161	0.519	0.268
B-ORG	0.816	0.033	0.707	0.052	0.756	0.032	0.609	0.041
B-PER	0.888	0.032	0.863	0.041	0.875	0.025	0.778	0.040
I-FAC	0.643	0.476	0.159	0.183	0.416	0.182	0.154	0.181
I-GPE	0.793	0.107	0.860	0.066	0.824	0.080	0.707	0.121
I-LOC	1.000	0.000	0.596	0.258	0.719	0.202	0.596	0.258
I-ORG	0.712	0.096	0.736	0.094	0.717	0.061	0.562	0.071
I-PER	0.832	0.052	0.878	0.079	0.852	0.046	0.744	0.068
Not-O	0.837	0.026	0.810	0.025	0.823	0.025		

Note that print-cross-validation also included aggregate statistics for the combination of all named-entity (“Not-O”) labels. Researchers could also optionally print the per-fold metrics prior to aggregation, along with per-classifier statistics prior to voting. Command-line arguments allowed a choice of prediction type (in-bag, out-of-bag, test, etc) plus a choice of voting strategy. Because print-cross-validation was often run after every experiment, sometimes multiple times to test different prediction types and voting strategies, performance was an important consideration. Eventually, we implemented a coordinator-worker model, much like that of our ensemble growing code, allowing print-cross-validation to load predictions and perform voting in-parallel.

Our next most useful postprocessing tool was the “plot-ensemble-growth” script — which would begin by computing the per-label performance metrics for the first classifier in an ensemble fold, then compute the same metrics for the first *two* classifiers in the fold using voting, then the first *three* classifiers, and-so-on, keeping a running total of the ensemble performance as each classifier in the fold was “added” into the mix. Plotting the resulting curves provided a straightforward indication of how ensemble performance changed as size grew. A majority of the performance plots in this paper were generated in this fashion; Figure 8 is an example.

Many other postprocessing workflows were variations on these two themes: for example, a special version of plot-ensemble-growth was used to visually compare the performance of different voting strategies on the same experiment.

6 Experiments, Speculations, Conclusions

6.1 Ensemble Experiments

6.1.1 Single CRFs and Basic Bagging

Section 4.2 described the various CRFs implementations we considered and worked with. As mentioned there, we initially settled on Mallet [24] as our CRF engine, and based a fair amount of experimentation on it, before developing our own implementation, pyCrust. One of the primary reasons for doing so was that none of the ensemble methods we applied to Mallet CRFs ever generated a better result than a single Mallet CRF. Mallet CRFs were indeed very stable and resistant to ensembling when treated as a black box.

So we thought we'd need a new CRF code that would permit us to manipulate the internals of the black box in order to make progress. We were thus initially surprised to observe that bagging, the most basic ensemble method, *did* improve performance when applied to pyCrust CRFs.

Bagging Helps pyCrust, but Not Mallet, in NGC300 Our first indication came from doing 10-fold cross validation of bagged CRFs for pyCrust and Mallet on the NGC300 data set; the results are in Table 3, which reports on the Not-O F1 metric. The interpretation here is that a single Mallet CRF achieves 0.952, but that bagging Mallet CRFs buys you nothing. On the other hand, a single pyCrust CRF that sees all the data achieves 0.951, which is slightly worse, but bagging them achieves 0.958, which is a substantive jump over Mallet's.

Note also the “average” performance in the ensemble. Mallet's average performance across all of the 50 bagged CRFs in the ensemble is identical to its non-bagged performance; indeed, it is very stable. pyCrust, on the other hand, has an average Not-O F1 of 0.940 across the 50 bagged CRFs. Its *average* performance has dropped, but its ensemble performance (here, via simple majority voting) has recovered and then some. This is the classic behavior of a base classifier with high variance error, which is the behavior you want with ensemble methods, as it leads to diverse individual classifiers.

Bagging Helps pyCrust, but Probably Not Mallet, in Newswire We repeated this investigation with the newswire data set, to insure that this observation wasn't a one-off artifact. The results are in Table 4. In this case, we see that Mallet's ensemble performance of .869 is slightly higher than its single CRF performance of .864. On the other hand, the standard deviations for those two numbers are 0.022 and 0.020, a good deal larger than the delta of 0.005. So the difference is likely not statistically significant.

We also see that, again, pyCrust's ensemble performance of 0.888 is substantially larger than its single CRF performance of 0.876, and as the pertinent standard deviations across folds are 0.003 and 0.005, that looks to be a statistically significant difference.

	Ensemble	Average
single Mallet CRF	–	0.952
50 bagged Mallet CRFs	0.952	0.952
single pyCrust CRF	–	0.951
50 bagged pyCrust CRFs	0.958	0.940

Table 3. 10-Fold Cross-Validation, NGC300

	Ensemble	Average
single Mallet CRF	–	0.864
50 bagged Mallet CRFs	0.869	0.863
single pyCrust CRF	–	0.876
50 bagged pyCrust CRFs	0.888	0.859

Table 4. 10-Fold Cross-Validation, newswire

Further, and surprisingly, pyCrust in general gives better results than Mallet. Both single CRF and ensemble CRF performance is higher than Mallet’s, and in fact, a single pyCrust CRF is better than an ensemble of Mallet CRFs.

Speculation As To Why pyCrust was originally intended to be functionally equivalent to Mallet, so as to permit a consistent starting point. However, Mallet turned out to have a number of under-documented or otherwise inexplicable implementation choices, so pyCrust could not match it exactly. And once we realized an exact match was impossible, pyCrust was allowed to diverge a bit more for the sake of efficiency. For instance, pyCrust is initialized (by default) with all λ -values set to 0.0, rather than to random values.

As a result, we believe that the differences in Mallet and pyCrust observed above stem from Mallet using various heuristics and source-code-level settings in an attempt to make *individual* CRFs as accurate as they can be. In particular, Mallet adds some extra edges to its factor graph, and does some additional regularization. We believe the result of these design decisions has been to reduce the variance error in the Mallet models, which has the counter-intuitive but well-established effect of *decreasing* the ensemble accuracy of such models.

We considered designing precise tests of this hypothesis, but determined that such tests would require modifications of precisely the parts of the Mallet source code that proved bafflingly enigmatic the first time around, and so these tests were not attempted before the end of the project.

Single pyCrust CRF Baseline for Train/Validation Data As a final baseline to record: as mentioned in Section 3, we generated a “VNGC92” data set to serve as a validation dataset for

NGC300, so that we could generate comparative performance results more quickly than was possible with 10-fold cross validation.

So, for the sake of comparison against subsequent ensemble results, a single NGC300 pyCrust CRF generates a Not-O F1 of .932 when tested on VNGC92. So 0.932 is the value to beat, when considering ensembling results.

Similarly, a single NEWSWIRET CRF generates a Not-O F1 of 0.809 when tested on NEWSWIREV.

6.1.2 Ensemble Size

Early investigations into bagged ensembles of CRFs made it seem at least possible that accuracy could degrade as the ensemble size increased, as there seemed to be a mild decrease at size 100 from a peak at around size 50. This would have been unfortunate, if true, as it would have required carefully monitoring ensemble size.

So in the context of the feature sampling investigations to be discussed below, we grew ensembles with sizes from 10 to 1000, in increments of 10. As an example, see the “fsamp:1.0” red line at the top of Figure 8. This made it happily clear that the apparent degradation at size 100 was just a small statistical fluctuation. After hitting a peak of 0.950 earlier on, performance varies more or less randomly from 0.948 to 0.950 until the end of the run.

We then investigated the same ideas in NEWSWIRET/NEWSWIREV, to see if this general effect of bagging performance flattening but not degrading with ensemble size would generalize.

To that end, we built ensembles of NEWSWIRET with sizes from 1 to 100, and evaluated them on NEWSWIREV. Figure 4 depicts Not-O F1 performance, as a function of vote weighting scheme. The bottom line is that the peak performance of 0.819 (which compares favorably with the 0.809 seen in a single CRF) is first observed at an ensemble size of around 18, and does *not* materially degrade with increased ensemble size.

So the conclusion so far is that, as with decision trees, larger ensembles don’t actually hurt, except for wasting computational power.

6.1.3 Diversity Via Small Bags

It is generally the case that increasing the diversity of the base classifiers in an ensemble is a useful path towards increasing the overall accuracy of the ensemble. One straightforward way to do this with bagging is simply to reduce the size of the bags. That is, with standard 100% bagging, any two bags will have roughly two thirds of the training samples in common. The degree of overlap drops dramatically if the bags are selected (still with replacement) to be smaller than the original data set.

Figure 5 is an example result. The bottom line is that peak performance (0.952) is no better

Figure 4. Bagging pyCrust with Newswire Data, as a Function of Size

Figure 5. Bagging pyCrust with NGC, as a Function of Bag Size

than the peak so far, which comes from simple bagging, ensemble of size 50.

However, small bags first matched that performance with a size 50 ensemble where each CRF needed only 60% of the data. Since training a CRF speeds up faster than linearly with the size of the training set, this suggests that though small bags are not useful for improving performance overall, they may be useful for achieving similar performance faster.

6.1.4 Diversity Via Feature Sampling

Random Sampling of Feature Classes Another standard method for increasing the diversity of the base classifiers in an ensemble is “random subspaces”. The idea here is that each base classifier gets to see only a random subset of the feature classes available.

We investigated this idea in the context of CRFs. The NGC300 dataset has 92 feature classes. So we built random subspace ensembles with the number of feature classes permitted to each CRF ranging over (1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90). To be clear, if the number of feature classes was, say, 9, that meant each individual CRF in the ensemble picked 9 feature classes at random and built the best CRF it could from those 9 feature classes. The next CRF in the ensemble would re-select its feature classes randomly, and so likely they’d be different feature classes. Thus the entire ensemble likely sees all of the feature classes at some point, even though each individual CRF is constrained to a subset.

At each subspace level we built an ensemble with 100 CRFs, then evaluated it against VNGC92 with the first N CRFs, N ranging from 5 to 95 in increments of 5, to evaluate the impact of increasing ensemble size. Note that random subsampling subselects the feature classes, whereas bagging essentially subselects the training samples. Thus random subspaces can be investigated combined, or not combined, with bagging: we investigated both. The results are in Figures 6 and 7. There are three things to note:

- Random subspaces does not increase diversity sufficiently to *improve* on the ensemble performance of using all of the feature classes.
- However, once we are using about 15 of the 92 feature classes, we are essentially matching the performance of using all the feature classes. Since CRFs slow down much faster than linearly with the number of feature classes, this points at a mechanism for increasing the speed of building CRFs ensembles.
- Interestingly, most of the curves are fairly flat, once we are using at least the minimum effective number of feature classes. So performance was dominated much more by access to feature classes than by the size of ensemble. We will return to this in Section 6.2, as this observation was the initial inspiration for the work in that section.

Random Sampling of the Feature Instances Themselves As just described, the *straightforward* analog of random subspaces for CRFs would be that each CRF would get to see only a ran-

Figure 6. Bagged Random Subspaces Applied to CRFs and Feature Classes

dom subset of the feature classes. However, we realized that CRFs would permit a further, more fine-grained, generalization of this idea; we could sample some percentage of the actual feature *instances*.

Further, since sampling the features in this way materially changes how the data is perceived, we realized that this feature sampling idea could be applied in conjunction with, or separately from, bagging.

The results of an early experiment are in Table 5. In this case we are doing 10-fold cross validation experiments on NGC300, where we have fixed the ensemble size at 50 CRFs, and vary whether we do or do not bag, and also vary the degree of feature sampling.

One conclusion is that bagging hurts average performance, as compared to unbagged. This is as you'd expect, as average performance is evaluating single CRF performance, and the whole point of bagging is that you degrade the performance of individual base classifiers to improve the performance of the ensemble as a whole.

A little more surprising is that, when using feature sampling, bagging hurts *ensemble* perfor-

Figure 7. Unbagged Random Subspaces Applied to CRFs and Feature Classes

mance (as compared to not bagging), until we get to the point where we are using all or almost all of the features. Though at that point, bagging definitely helps. The highest overall performance is still using all the features plus bagging.

So the above suggests that with a random 60% of the features in an unbagged size 50 ensemble, we can recover the Not-O F1 achieved by a single CRF that gets to see all of the features. Further, that we can improve on that performance with 80% of the features.

We repeated this experiment from a different perspective, by building the ensembles on NGC300 and testing them on VNGC92. We also varied the ensemble size. The results are in Table 6.

Recall that a single NGC300 CRF generates .932 against VNGC92. Note also that you can read off the results of standard bagging without feature sampling simply by looking at the “fsamp 1.0” row.

So here the peak “standard bagging” performance was 0.951, at a bag size of 50, and we did only slightly better, .952, with a bag size still of 50, but dropping the feature sampling to 0.9. Still,

sampling	Ensemble Bagged	Average	Ensemble Unbagged	Average
0.1	0.710	0.709	0.799	0.795
0.2	0.845	0.837	0.889	0.884
0.3	0.898	0.886	0.921	0.917
0.4	0.924	0.911	0.939	0.933
0.5	0.937	0.925	0.947	0.942
0.6	0.946	0.934	0.954	0.948
0.7	0.951	0.939	0.955	0.950
0.8	0.954	0.942	0.959	0.952
0.9	0.955	0.944	0.955	0.950
1.0	0.958	0.940	0.951	0.951

Table 5. Random Sampling of Feature Instances: Cross-Validation

this didn’t seem likely to be significant.

We checked a bit more exhaustively by looking at very large ensembles, in case the feature sampling requires very many small attempts to ultimately improve accuracy. The results are in Figure 8, and were not encouraging; overall, simple feature sampling is not a promising venue.

6.1.5 The Effect of Voting Scheme

As discussed in Section 5.2.1, in the course of the project we implemented and investigated a number of voting schemes for integrating the opinions of the various CRFs in an ensemble as to the correct label for a token.

The primary scheme was “SimpleMajorityRandomTie”, a voting strategy that picked predictions based on a simple majority for every token, breaking ties at random. This was the first scheme used in standard ensemble methods, and despite many investigations into competitors, it has turned out to be not only the simplest, but the most robust and reliable.

Still, in the context of CRFs, a CRF simultaneously assigns labels to all tokens in a sequence so as to maximize their joint likelihood, where a majority voting scheme ignores that nuance, and considers each token independent of its neighbors. As a result, we thought there might be room for improvement in voting mechanism, especially in the context of threaded sequences.

To date, however, we have not found that improvement; as with non-CRF ensembles, simple majority seems the best choice. None of the thread-aware voting schemes described in Section 5.2.1 ever matched the accuracy of majority voting.

	size 10	size 20	size 30	size 40	size 50
-----	-----	-----	-----	-----	-----
	bagging?	bagging?	bagging?	bagging?	bagging?
	no, yes	no, yes	no, yes	no, yes	no, yes
fsamp 0.1	0.825, 0.765	0.824, 0.762	0.822, 0.763	0.822, 0.765	0.823, 0.765
fsamp 0.2	0.894, 0.861	0.894, 0.862	0.895, 0.862	0.892, 0.863	0.892, 0.865
fsamp 0.3	0.919, 0.890	0.923, 0.893	0.923, 0.894	0.924, 0.895	0.924, 0.897
fsamp 0.4	0.930, 0.917	0.930, 0.920	0.931, 0.922	0.930, 0.922	0.933, 0.924
fsamp 0.5	0.940, 0.932	0.941, 0.933	0.942, 0.934	0.942, 0.933	0.941, 0.933
fsamp 0.6	0.943, 0.939	0.945, 0.940	0.944, 0.941	0.945, 0.941	0.943, 0.942
fsamp 0.7	0.944, 0.945	0.945, 0.945	0.945, 0.944	0.945, 0.944	0.945, 0.945
fsamp 0.8	0.946, 0.946	0.945, 0.945	0.945, 0.946	0.945, 0.947	0.945, 0.948
fsamp 0.9	0.944, 0.949	0.944, 0.948	0.945, 0.950	0.946, 0.950	0.945, 0.952
fsamp 1.0	0.932, 0.947	0.932, 0.948	0.932, 0.949	0.932, 0.950	0.932, 0.951

	size 60	size 70	size 80	size 99
-----	-----	-----	-----	-----
	bagging?	bagging?	bagging?	bagging?
	no, yes	no, yes	no, yes	no, yes
fsamp 0.1	0.821, 0.764	0.825, 0.766	0.826, 0.765	0.825, 0.766
fsamp 0.2	0.893, 0.864	0.892, 0.865	0.892, 0.863	0.894, 0.865
fsamp 0.3	0.924, 0.897	0.924, 0.899	0.925, 0.898	0.924, 0.898
fsamp 0.4	0.931, 0.922	0.931, 0.924	0.931, 0.923	0.932, 0.923
fsamp 0.5	0.940, 0.931	0.940, 0.933	0.940, 0.933	0.941, 0.933
fsamp 0.6	0.944, 0.942	0.943, 0.941	0.943, 0.941	0.944, 0.942
fsamp 0.7	0.945, 0.944	0.945, 0.945	0.945, 0.945	0.946, 0.945
fsamp 0.8	0.946, 0.947	0.945, 0.948	0.945, 0.948	0.945, 0.947
fsamp 0.9	0.945, 0.951	0.946, 0.951	0.945, 0.950	0.946, 0.950
fsamp 1.0	0.932, 0.948	0.932, 0.949	0.932, 0.948	0.932, 0.950

Table 6. Random Sampling of Feature Instances: Train/Test

Of the non-thread-aware schemes, only “UnanimousOther” ever even matched majority voting, and then only for a suitable selection of its threshold value. An example is Figure 8, which, as described earlier, resulted from building and evaluating ensembles of NEWSWIRET of size from 1 to 100. As can be seen, only UnanimousOther with the specific threshold of 0.4 matched the peak performance of majority voting. To make sure that this was not an effect of ensemble size, we repeated the experiment with those two voting schemes, but out to an ensemble size of 500, rather than 100: see Figure 9. As before, UnanimousOther with the specific threshold of 0.4 matches but does not improve on majority voting. Given that majority voting requires no parameter tuning, our current conclusion is to recommend it as the default voting scheme for CRF ensembles.

Figure 8. Bagging pyCrust with NGC, as a Function of Size and Feature Sampling

6.2 The Effect of Feature Selection

One of the attractive properties of ensembles of decision trees is that they are robust to the inclusion of weak or irrelevant features. This makes them easy to use, because there is no need for careful feature selection, normalization, de-correlation, or other preprocessing.

Early anecdotal observations suggested that ensembles of CRFs might not share that property, so we conducted a series of investigations specifically into this issue.

6.2.1 Simple Feature Selection

Inspired by the feature class subspaces experiments in Section 6.1.4, which suggested that we can achieve much of our performance with smaller subsets of the feature classes when combined via ensembles, we investigated whether we could usefully make *individual* CRFs better by careful feature selection.

Figure 9. SimpleMajority voting vs UnanimousOther-0.4 for a Large Ensemble.

6.2.2 Greedy Feature Class Selection for Single CRFs

To begin, we conducted some greedy feature class “forward selection” experiments. That is, the NGC300 data set has 92 feature classes. We built 92 individual CRFs, each of which worked with only a single feature class. We evaluated each CRF on VNGC92 via the standard Not-O F1 metric, and determined which feature class gave the best result. This turned out to be STEM, which achieved 0.89. (For comparison, recall, from paragraph 6.1.1, that the baseline single CRF performance, using all feature classes, is 0.932, and that the bagged ensemble performance baseline is 0.952.)

Then we built all 91 CRFs that can be built by pairing STEM, one by one, with all of the other 91 feature classes, and determined the best feature class to add; it was TOKEN-LENGTH. We iterated this process in the obvious way: the first few selections worked out as follows:

```
Best Score | Label
=====
0.8907    | STEM
0.9167    | TOKEN-LENGTH
0.9239    | PROPER-NAME
0.9406    | IS-CAPITALIZED
0.9443    | IS-UPPER-CASE
0.9443    | LAST-TOKEN
0.9443    | CORPUS-BEGIN
...
```

and a plot of performance as feature classes are added is in Figure 10. Some observations:

- Peak performance is 0.955, achieved when using 80 of the 92 feature classes. This is not only statistically significantly higher than the 0.932 achieved when using all feature classes, it is slightly higher (though not statistically significantly so) than the 0.952 peak observed when using bagged ensembles with all feature classes.

In other words, this experiment shows at least one example in which careful feature selection generates a single CRF that matches ensemble performance.

- The fact that performance drops once we add more than 80 feature classes demonstrates that CRFs do not have the property, possessed by decision trees, of being robust to weak or junk features.
- As can be seen from the figure, once five feature classes were in the mix, there was no accuracy improvement gained by adding the next seventeen feature classes. This despite the fact that they were, by design, the *best* feature classes to add; any other feature classes would have decreased accuracy, if added at that stage. This phenomena is not well understood.

Figure 10. Single CRF Accuracy as Feature Classes are Added

6.2.3 Greedy Best Feature Class Selection for Ensembles

The above illustrates that with the best five feature classes (selected in a greedy and admittedly not necessarily optimal fashion), we can achieve accuracy of 0.943 in a single CRF, and that accuracy does not increase further until we have 23 feature classes.

This inspired an experiment in which we built an *ensemble* of CRFs where we restricted each CRF to consider only the best five features: STEM TOKEN-LENGTH PROPER-NAME IS-CAPITALIZED, and IS-UPPER-CASE. Since it was easy to do, we simultaneously investigated various levels of feature sampling, as we did in Section 6.1.4. The result is in Figure 11. The bottom line is that peak performance is neither helped nor hurt by this process.

Figure 11. Feature Sampling with Focused Features

6.2.4 Focusing On Features That Are Individually Useful

As an alternative perspective into determining the most useful features, we built 92 single CRFs trained on NGC300, such that each CRF was permitted to see only one of the 92 feature classes. When restricted in this way, all of the CRFs performed poorly, and in fact only 33 of them managed to generate a non-zero NOT-O accuracy on vngc300. That is, when not permitted the help of other feature classes, only 33 feature classes were even minimally useful.

To investigate their utility, we grew ensembles of CRFs which were permitted to see only the most useful feature class, then the two most useful feature classes, and so on. Figure 12 plots the 33 accuracy curves that resulted. To help clarify that welter of plots, Figure 13 shows only a subset of the curves, but also adds in the *average* (that is, non-ensemble) performance of each feature set, so that we can see how much the ensemble process adds.

The bottom line is that peak performance is still only 0.952, which is the peak performance observed with untrained use of all of the features and an ensemble size of 50. That level was achieved here, though, with an ensemble size of 9 and with only 23 feature classes.

Figure 12. Ensemble Size Vs. Number of Useful Feature Classes

All of which continues to suggest that careful feature selection in advance can speed of the use of CRF ensembles later, as CRF ensembles in action scale linearly with ensemble size and worse than linearly with the number of feature classes, all without adversely affecting accuracy.

6.2.5 Greedy Worst Feature Class Avoidance for Ensembles

The previous experiments in this section looked into the effect of focusing on the most useful features, for a couple of different methods of defining “useful”. In this section we consider the opposite extreme, that of simply avoiding the least useful features.

For instance, we conducted a forward feature selection process, as in Section 6.2.2, but with the NEWSWIRET dataset. We here define a feature class as “toxic”, if over a variety of parameter studies, they were the last to be selected, and in fact, when selected tended to *decrease* the overall accuracy of the *individual* CRF. With that definition, the following were reliably toxic for NEWSWIRET: NEXT-TOKEN-LENGTH, SUMMARIZED-PATTERN, TOKEN-LENGTH, NON-ALPHA and IS-CAPITALIZED.

Figure 13. Subset: Ensemble Size Vs. Number of Useful Feature Classes

The natural impulse would be then to simply avoid those feature classes, and in the case of individual CRFs, that does indeed help. However, we needed to check the effect on ensembles of CRFs, as often increasing the accuracy of the base classifiers can decrease the overall ensemble accuracy, as doing so decreases classifier diversity as well.

And that is indeed what we observed. The purple horizontal line in Figure 14 is the result of building ensembles of increasing size on all but the toxic feature classes. The peak result of 0.817 is indeed better than the single CRF “use all feature classes” baseline of 0.812, but it does *not* improve on peak value of 0.819 in the red curve, which depicts building ensembles on *all* of the feature classes. So the improvement in individual CRF accuracy did not translate to ensemble accuracy.

6.2.6 Light, Middle, and Heavyweight Features for Ensemble CRF Improvement

In Section A.1.1 we discuss an idea for an “advanced” feature selection mechanism we didn’t have time to pursue. Thinking through that idea, though, led us to understand that the number

Figure 14. Accuracy from Uncleaned, Cleaned, or Non-Toxic Features

of features generated by the various feature classes would likely vary, and that indeed we could usefully distinguish our feature classes by that property.

That is, a “heavyweight” feature class is one like PREV-WORD, which generates a feature for every unique token in a data set, and a lightweight feature class is one that rarely generates tokens, such as LAST-TOKEN. There is a smaller set of mid-weight features, such as NON-ALPHA, which often but not invariably generates a feature for a token. We looked into effects of building CRFs from incrementally larger pools of feature classes selected only from lightweight or mid-weight features. The results are conveyed by the blue curve in Figures 16 and 15, as evaluated by building CRFs on NGC300, and testing them on VNGC92.

If you use the both medium and lightweight features, Not-O F1 grows only to .61, and hits that peak only when all of those features are used. So the heavyweight features matter more than we expected, as we’ve dropped far below 0.84.

With lightweight features *only*, Not-O F1 drops to .50. The notable dip is at the LOWERCASE feature. Which doesn’t mean that it was a bad feature; even though it lowered performance, the

Figure 15. Accuracy from Light and Midweight Features Only

fact that it was the one selected meant it lowered performance the least.

6.3 Feature and Factor Graph Visualization

6.3.1 Feature Class Similarity

The previous sections have established the importance of understanding the relative utility of the feature classes, and their relations to each other. According, we iteratively explored a number of methods for quantifying and visualizing those relationships.

As a first pass, we built an M by N matrix F , where M rows represent the number of tokens in the NGC300 dataset, and $N = 93$ columns each representing a feature class. $F(i, j)$ is non-zero if token m has at least one non-zero feature in feature class n . Then we computed all pairwise similarities of the columns to get a 93 by 93 diagonally symmetric similarity matrix, and fed that into a hierarchical clustering algorithm. We investigated cosine, Jaccard, and Minkowski distance metrics, all with similar results, represented by Figure 17. This was interesting and useful in that it helped us understand that clustering on the use of feature classes across tokens is not actually

Figure 16. Accuracy from Lightweight Features Only

what we want. As an example, consider the features STEM and TOKEN-LENGTH. They *always* generate a feature. They are, technically, not sparse. So the feature class vector extracted here is an all-1 vector for both of them, and for all features with that property (which is most of the feature classes near the top of Figure 17). So they look perfectly correlated, or perfectly aligned via cosine similarity.

The important parts of a CRF model are actually the lambda weights, the label-to-label transition weight inferred for each feature from the training data, and so clustering based on those weights was our next step.

That is, consider that NGC300 has three labels, and so there are 3^2 bigram transitions and 3 unigram transitions for a total of 12. We could thus conceptually build a 12 by 93 matrix, where each cell of the matrix is actually a collection of all of the lambdas measured in the model when a token has the indicated transition and the indicated feature. The question was how to represent the lambdas in each cell, in support of the goal of evaluating the similarity of feature classes C_i and C_j ?

Figure 17. Hierarchical Clustering of Cosine Distance between Feature Class Population Vectors

Our approach was to turn the lambdas in each cell into a cumulative distribution function (CDF), and then compare CDFs with the KolmogorovSmirnov test⁹, which provides a measure of the statistical similarity of CDFs. This would give us the ability to visualize the pairwise similarity of two features classes with respect to a given label transition. For example, consider Figure 18, which shows those similarities for the “O to B-PER” label transition. [ht!] Figure 19, similarly, simultaneously shows these similarity matrices for all twelve of the label transitions in NGC300.

⁹https://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test

Figure 18. KS Feature Class Similarity for "O to B-PER" Label Transition

Figure 19. KS Feature Class Similarity for All Label Transitions

6.3.2 Factor Graph Visualization

A persistent challenge in working with an inference model as complicated as a CRF is simply keeping straight the structure and content in the model. Early in the project we used a conceptual example, such as in Figure 20. The idea here is to represent four different types of CRF parameters,

Figure 20. A Conceptual Factor Graph

as an aid to thinking about which parts of the model pyCrust should enable altering.

- Green edges: one for each (label i to label j transition using feature k) triple K in a training sequence.
- Red outline: one for each (Start(S) to label i using feature k) triple.
- Purple outline: one for each (label $i \rightarrow$ label j) transition.
- Blue edge: one for each (label $i \rightarrow$ End/ E) transition
- Blue nodes: shades denote different feature classes
- Green edges: sampling (dropping) edges will implicitly affect CRF model priors
- $x\#$ nodes: sampling (dropping) nodes will alter the set of CRF parameters used
- Blue edges or purple/red outlined nodes: sampling (dropping) per sequence will affect priors, over all instances will alter set of CRF parameters

That graphic was indeed useful for thinking about factor graphs in general, but eventually we needed to be able to visualize specific factor graphs constructed from specific sequences. We created such a tool, `factor-graph-visualization.py`, could load a CRF and visualize the factor graph for either a random training sequence or a probe sentence entered at the command line.

As an example, we built a CRF using all of NGC300, and asked it to analyze the sentence “Scott and I are asking you to be one of the group.” The raw labeling generated by the tool is correct:

```
Scott [B-PER] and [O] I [O] are [O] asking [O] you [O] to [O]
be [O] one [O] of [O] that [O] group [O] . [O]
```

Further, for each token in the sequence, the tool indicated the features in the factor graph that were non-zero for that token, that is, those that contributed to its classification. The visualization for the first two tokens, “Scott and”, is in Figure 21. Looking at the top left part of that plot, for instance, shows that not only did “Scott” activate the PROPER-NAME feature class, but that that feature class was the most heavily weighted contributor. Note also, for instance, that “LOWER-CASE(scott)” was non-zero and was even positive. This is properly interpreted as reflecting the fact that “Scott” does indeed have a non-proper-name meaning which is most likely to be observed when “scott” is lower case. So the CRF has learned to assign some interpretive weight to a lower case “scott”. Note also, near the far right, that “SENTENCE-BEGIN” has one of the highest weights. That’s likely because sentences usually start with a capital letter, and that’s particularly important here, since “scott” does indeed have a lower-case interpretation. So the position of an upper-case “Scott” at the beginning of a sentence is weaker evidence for the label “B-PER” than an upper-case “Scott” elsewhere in the sentence would provide.

6.4 Lambda Manipulation

6.4.1 Lambda Cleaning

As mentioned in Section 3.1.9, we implemented a script to identify unexpected label transitions in a corpus’ ground truth. Unexpected label transitions were transitions that should never occur in correctly-labeled data, such as a transition from “O” to “I-PER” or “B-LOC” to “I-PER”.

When investigating the CRF lambdas generated by these cleaned up data sets, we occasionally saw non-zero lambdas for such unexpected transitions. This was not an error, but a side-effect of our initialization process. That is, one way that pyCrust generates diversity is by starting with a random assignment of lambda weights. They get pushed to zero only if there is evidence to do so.

This eventually generated an approach to “cleaning up” a CRF by thresholding its lambdas. As mentioned, there might be non-zero lambdas in the CRF that are non-zero only because there wasn’t enough training data to push them to zero from their random starting point. It would be great to threshold those by fiat, but there didn’t originally seem to be a principled way to set that threshold.

Considering unexpected transitions suggested an idea, however. If we examined a CRF and found the maximum magnitude, λ_C , assigned to an unexpected transition, λ_C would be an indication of the magnitude possible to known-irrelevant lambdas. Thus we hypothesized we could safely reset *all* lambdas less than λ_C . Further, rather than set them to zero, we would do better to set them to a very low number; we chose -20. “zero” means it is unknown how relevant that feature is to that transition. A low negative number means we are asserting that that feature is *not* relevant to that transition, which is what we intend.

We looked into this idea from the perspective of feature selection for single CRFs, and as concerns its impact on ensemble performance.

Figure 22. Feature Selection Accuracy after Lambda Cleaning

Lambda Cleaning and Feature Selection An example result is in Figure 22, which depicts running a single CRF forward selection investigation over NEWSWIRET, testing on NEWSWIREV, after setting all unexpected transition lambdas to -20.

The impact, from this perspective, was not dramatic. With lambda cleaning, we don’t peak as quickly, but we do peak a bit better, slightly higher than 0.812.

We also observed that after the first couple of feature classes, there was a lot of variation in the order in which feature classes are added, depending on whether the lambdas were cleaned or not. However, the set of final features added was pretty consistent, lending weight to the identification of certain features as toxic.

Lambda Cleaning and Ensembles As a baseline result, with no lambda cleaning, we built an ensemble of 100 bagged pyCrust CRFs on NEWSWIRET, testing on NEWSWIREV as the ensemble size grew. Peak Not-O F1 was 0.819, around ensemble size 20.

We repeated that investigation after lambda cleaning; the result is in Figure 23, which shows that the impact is again not dramatic. Peak was again 0.819, and behavior was overall largely the

Figure 23. Ensemble Accuracy with Cleaned and Uncleaned Lambdas

same. Which suggests, perhaps, that the CRF is already doing a good job of driving the appropriate lambdas low, and that there is not much performance to be gained by pushing them further.

6.4.2 Knockout Ensembles

The idea behind lambda cleaning, just discussed, was to remove links in the factor graph in a deliberate fashion, to attempt to identify unhelpful links and to set their weights to, effectively, $-\infty$, so that they won't affect classification of a new sequence.

An alternative approach would be to knock out links more or less at random. This might not seem useful *a priori*, but random distortions of the data set are at the heart of the seminal and effective bagging technique. The idea is that the distortion in bagging generates classifiers that are individually less accurate, but since they are also diverse, when voted in an ensemble they generate an overall classifier which is *more* accurate, as it benefits from multiple perspectives on the data.

But distorting the data and then building a new CRF will be just as slow as building a CRF from undistorted data. The advantage of this lambda knock-out approach is that once the original CRF exists, generating variants would be very, very fast.

This idea received initial but not conclusive investigation, as it was first tackled near the end of the project. We built a single CRF on NEWSWIRET, and then built ensembles where every member of the ensemble had a random (and thus different) fraction N of its non-zero links set to $-\infty$, where N varied over (0.01, 0.05, 0.10, 0.15, 0.20, 0.25, 0.30, 0.35, 0.40, 0.45, 0.50, 0.55, 0.60, 0.65, 0.70, 0.75, 0.80, 0.85, 0.90, 0.95, 0.99), and where the ensembles were tested on NEWSWIREV.

Recall that the single CRF baseline was 0.809. The peak performance in the experiment above was .815, achieved with an knockout ensemble of size 10, and this compares to 0.819, achieved with a standard bagged ensemble of size roughly 20.

So knockout ensembles are behaving comparably, if slight worse. The thing to note, though, is that the knockout ensemble of size 10 took only 1.02 times as long to build as a single CRF. So the bagged ensemble of size 20 took nearly twenty times as long to build as the knock out ensemble.

Note, incidentally, that this knock-out idea is but one example of a general class of “edited clone” CRF generation mechanisms, in which a single source CRF is laboriously generated and then edited or altered in various ways to quickly generate variants. Other possibilities, discussed further in Appendix A.1.2, would be to alter, even increase, link weights randomly, to drop features rather than links, and so on.

References

- [1] Robert E. Banfield, Lawrence O. Hall, Kevin W. Bowyer, and W. Philip Kegelmeyer. A comparison of decision tree ensemble creation techniques. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(1):173–180, January 2007.
- [2] David Barber. *Bayesian Reasoning and Machine Learning*. Cambridge University Press, 2012.
- [3] Steven Bird, Ewan Klein, and Edward Loper. *Natural Language Processing with Python*. O'Reilly Media Inc., 2009.
- [4] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)*, pages 177–187, 2010.
- [5] Leo Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996.
- [6] Yu-shan Chang and Yun-Hsuan Sung. Applying name entity recognition to informal text. CS224N/Ling237 Final Projects 2005, 2005.
- [7] Nitesh V. Chawla, Lawrence O. Hall, Kevin W. Bowyer, and W. Philip Kegelmeyer. Learning ensembles from bites: A scalable and accurate approach. *Journal of Machine Learning Research*, 5:421–451, 2004.
- [8] Nancy Chinchor. Overview of MUC-7. In *Proceedings of the Seventh Message Understanding Contest*, Fairfax, VA, USA, 1997.
- [9] William W. Cohen. MinorThird: Methods for Identifying Names and Ontological Relations in Text using Heuristics for Inducing Regularities from Data. <http://github.com/TeamCohen/MinorThird/>, 2004. Accessed: 2013-09-24.
- [10] S. Della Pietra, V. Della Pietra, and J. Lafferty. Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4):380–393, 1997.
- [11] A. Ekbal and S. Saha. Weighted Vote Based Classifier Ensemble Selection Using Genetic Algorithm for Named Entity Recognition. *Natural Language Processing and Information Systems*, pages 256–267, 2010.
- [12] Charles Elkan. Log-linear models and conditional random fields, 2008. Tutorial notes at CIKM '08.
- [13] Email Tokenizer. http://code.google.com/p/nltk/source/browse/trunk/nltk-old/contrib/nltk_contrib/unimelb/460/2003/aclaussen/mailtokenizer.py. Accessed: 2011-05-10.
- [14] Jenny Rose Finkel, Trond Grenager, and Christopher Manning. Incorporating non-local information into information extraction systems by gibbs sampling. In *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics (ACL 2005)*, pages 363–370, 2005.

- [15] Bryan Klimt and Yiming Yang. The enron corpus: A new dataset for email classification research. In Jean-François Boulicaut, Floriana Esposito, Fosca Giannotti, and Dino Pedreschi, editors, *Machine Learning: ECML 2004*, volume 3201 of *Lecture Notes in Computer Science*, pages 217–226. Springer Berlin Heidelberg, 2004.
- [16] R. Klinger and K. Tomanek. Classical probabilistic models and conditional random fields. Technical report, Technische Universität Dortmund, Dortmund, Electronic Publication, 2007.
- [17] Daphne Koller and Nir Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.
- [18] Ludmila (Lucy) Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Wiley-IEEE, 2004. ISBN 0471210781.
- [19] John D. Lafferty, Andrew McCallum, and Fernando C. N. Pereira. Conditional random fields: Probabilistic models for segmenting and labeling sequence data. In *Proceedings ICML*, pages 282–289, 2001.
- [20] Thomas Lavergne, Olivier Cappé, and François Yvon. Practical very large scale CRFs. In *Proc. Association for Computational Linguistics (ACL)*, pages 504–513, 2010.
- [21] Tracy D. Lemmond, Nathan Perry, Joseph Guensche, John J. Nitao, Ronald Glaser, Paul Kidwell, and William G. Hanley. Enhanced named entity extraction via error-driven aggregation. In *DMIN’10*, pages 31–37, 2010.
- [22] David D. Lewis, Yiming Yang, Tony G. Rose, and Fan Li. Rcv1: A new benchmark collection for text categorization research. *J. Mach. Learn. Res.*, 5:361–397, December 2004.
- [23] Dong C. Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical Programming*, 45:503–528, 1989.
- [24] Andrew Kachites McCallum. Mallet: A machine learning for language toolkit, 2002. <http://mallet.cs.umass.edu>.
- [25] Einat Minkov, Richard C. Wang, and William W. Cohen. Extracting personal names from email: applying named entity recognition to informal text. In *Proceedings of the conference on Human Language Technology and Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 2005.
- [26] Alexis Mitchell. TIDES Extraction (ACE) 2003 Multilingual Training Data, 2004. <http://www ldc.upenn.edu/Catalog/CatalogEntry.jsp?catalogId=LDC2004T09>.
- [27] David Nadeau. *Semi-supervised named entity recognition: Learning to recognize 100 entity types with little supervision*. PhD thesis, University of Ottawa, 2007.
- [28] Penn Treebank Tokenizer. <http://nltk.org/api/nltk.tokenize.html#module-nltk.tokenize.treebank>. Accessed: 2013-09-23.

- [29] Porter Stemmer. <http://nltk.org/api/nltk.stem.html?highlight=porter#nltk.stem.porter.PorterStemmer>. Accessed: 2013-09-23.
- [30] Punkt Sentence Tokenizer. <http://nltk.org/api/nltk.tokenize.html?highlight=punkt#nltk.tokenize.punkt.PunktSentenceTokenizer>. Accessed: 2013-09-23.
- [31] Lawrence R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Proceedings of the IEEE*, volume 2, pages 257–286, 1989.
- [32] Charles Sutton. Grmm: Graphical models in mallet, 2006. <http://mallet.cs.umass.edu/grmm>.
- [33] Charles Sutton and Andrew McCallum. An introduction to conditional random fields for relational learning. In Lise Getoor and Ben Taskar, editors, *Introduction to statistical relational learning*. MIT Press, 2007.
- [34] Kai Ming Ting, Jonathan R Wells, Swee Chuan Tan, Shyh Wei Teng, and Geoffrey I Webb. Feature-subspace aggregating: ensembles for stable and unstable learners. *Machine Learning*, 82(3):375–397, 2011.
- [35] Using LDC Data. <http://www.ldc.upenn.edu/Using/>. Accessed: 2013-09-24.
- [36] Hanna M Wallach. Conditional random fields: An introduction. CIS Technical Report MS-CIS-04-21, University of Pennsylvania, 2004.
- [37] Ping Zhong and Runsheng Wang. A multiple conditional random fields ensemble model for urban area detection in remote sensing optical images. *Geoscience and Remote Sensing, IEEE Transactions on*, 45(12):3978–3988, 2007.

A Loose Ends

This appendix is devoted to “loose ends”. That is, this is a terse dump of ideas at various degrees of development or maturity that we noted during the project, but did not pursue or entirely complete.

A.1 Ideas That Were Under Active Investigation When The Project Stopped

A.1.1 Advanced Feature Selection

The idea is to do a boosting sort of feature selection mechanism, based on the idea of picking features that are strong and decorrelated for current features, with reference to the label transitions we are doing worst on.

The algorithm would grow a feature selection that consists of full feature classes F_n and partial feature classes $\tilde{F}_n$, where $\tilde{F}_n$ contains a subset of the feature instances in F_n :

We begin with a pool of available feature classes and an empty feature selection. Feature selection always begins by choosing a full feature class F_n . Full feature classes are chosen as follows:

- Compute a model for each combination of the current feature selection and the feature classes in the available pool.
- Choose the feature class F_n which produced the best performance when added to the current feature selection.
- Add F_n to the feature selection.
- Remove F_n from the available pool.

Partial feature classes $\tilde{F}_n$ are chosen as follows:

- Compute a model using the current feature selection.
- Sort label transitions by their performance. Weight the performance by the number of times the label transition appears in the data, so we don’t focus our attention on rarely-occurring transitions.
- Choose the worst-performing label transition.
- Compute K-S scores using lambda distributions between the most-recently-chosen full feature class F_r and the feature classes in the available pool.
- Choose the feature class F_n with the lowest K-S score (lowest probability that it has the same distribution as F_r).

- Sort the feature instances in F_n by the magnitude of their lambda values from the model.
- Choose the top- m feature instances, adding them to the feature selection as F_n .
- Remove F_n from the available pool.

The algorithm might select one full feature class, followed by all partial feature classes. Or, it could alternate between choosing full and partial feature classes, etc.

One parameter to set here: ' m ' in "top- m " Should that be a fixed m or a percent of instances? Might need to investigate both, but fixed m seems likely to generate more diversity, more strongly reduce the influence of the "heavyweight" features.

Another variation to consider: alternate full and partial feature class selections, or do all partials after the first full? The initial sense was that the best plan would like be to do all partials after the first full selection.

A.1.2 CRF Creation Via Lambda Manipulation

In Section 6.4.2 we discussed "knock-out" CRFs, as one example of "edited clone" CRF, that is, one example of manipulating the lambda values in an already-generated CRF in order to quickly generate many variants in the hopes that they'd be diverse enough to help increase the accuracy of an ensemble of these variants.

We generated but did not pursue other ideas along this line:

- Flip signs on the lambdas.
- Adjust the magnitude of the lambda's. (Thresholding to zero, as in the knock-outs, is the simplest version.)
- Threshold lambdas not at random, but based on some operational insight. For instance, if the lambda magnitude is small, its impact is likely weak, and inferring from the model might be sped up without accuracy loss by zeroing that lambda before the model is used.
- Conversing, in an idea distantly related to boosting, determine the sequences least well predicted by the CRF, determine the lambdas that affect the specific features and label transitions in those sequences, and increase their magnitude. The idea, roughly, is to better the underperforming models.

A.1.3 Combining Paths to Diversity

Lambda knock-outs, simple bagging, and lambda cleaning all individually helped peak performance. The combination of all three methods at the same time did not, anecdotally, improve on the use on any one method, but that combination was not exhaustively investigated.

Part of the barrier to investigation was the lack of an easy mechanism for mixing CRFs built from different edited clone ensembles. The code was discussed, designed at a conceptual level, and described in the wiki notes, but at the end of the project these investigations were set up by hand, by renaming individuals CRFs to match code conventions that made sense before we thought to combine ensembles.

A.1.4 Mallet Destablization

In Section 6.1.1 we mentioned the desire to test the hypothesis that Mallet could be de-stablized to the point of being competitive with CRF, but that we had also “determined that such tests would require modifications of precisely the parts of the Mallet source code that proved bafflingly enigmatic the first time around, and so these tests were not attempted before the end of the project”.

We did, however, start a less in-depth investigation based on varying the handful of control parameters that Mallet exposes at the command line. Specifically, we used “-mallet-remove-start-state” to turn off the start-state it had settled on as optimal, and we ran a parameter search over values between 0.1 and 9.0 as arguments to “-mallet-gaussian-prior-variance”, in order to alter the expected prior variance in the model. These runs, denoted “wpk-nw-param06”, activated by a script of a similar name, were completed but went unanalyzed.

A.1.5 Feature-Class/Label Transition Pools

Near the end of the project we read a paper entitled *Feature-subspace aggregating: ensembles for stable and unstable learners*, [34]. Its core algorithm turned out to be inapplicable for sparse sequence data, but we were intrigued with the basic idea of trying to build CRFs that are expert in various “locations” in feature space.

This generated an idea we called feature-class/label-transition (FCLT) pools. The first idea was to pick each single feature class and train a CRF using only sequences where at least one token in the sequence has a non-zero feature in that feature-class. So this gets at a sort-of idea of locality, in that the CRF only operates on sequences where its primary feature matters. Then one would repeat for all 93 feature classes, make an ensemble of 93 CRFs, allow them all to vote on a new sequence.

Then we realized this could and should be extended to label transitions as well. For instance, NGC300 has 93 feature classes and 12 label transitions. The FC/LT pool approach would be to build 93x12 CRFs, where $CRF(i,j)$ trains only on the sequences that possess at least one token which both has a non-zero feature in feature class i and is involved in transition j .

As an additional nuance, with some additional bookkeeping it would be possible, at test time, to use only the CRFs indicated by the feature classes detected in the test sequence.

At the end of the project this was still considering an interesting, maybe even promising ap-

proach, but it was not pursued due to time triage. See the wiki notes for 13/06/07 for more details of that discussion.

A.2 Miscellaneous Open Issues

- It would be useful to have an mechanism for determining how long a given feature (or, better, feature class) takes to compute, and also its differential impact on the time it takes to infer from that model. Especially given that we've established that adding features can possibly hurt.
- Weighted averaging. Can we enhance pyCrust to extract an (estimate of) the PDF across labels induced by each CRF, and do weighted averaging instead of flat averaging?
- How to generalize out-of-bag (OOB) validation operate over state transitions. (So that we could, for instance, auto-size CRF ensembles.) Maybe we should bag over token pairs (that is, state transitions), not over sentences. But should we assess OOB? Bagging over 2-grams, but still do partitioning for cross-validation over sentences? Or is extracting F1 from 2-grams useful as a guide to performance on sentences?
- How to generate a non-CRF baseline to compare to. Look at ensembles of decision trees? How to set that up, to make use of the features we already compute? Perhaps instead, or also, look at Naive Bayes for a less drastic comparison to something simpler than CRFs.
- Have we made the right choice of tokenizer? How can we determine or measure this? How sensitive are our results to the choice of tokenizer. NLTK provides ten different tokenizers, we are currently using the one based on Penn Treebank; should we reconsider?
- An idea for shrinking the size of the state data stored by a cross-validation or model building run. (For efficiency, not for diversity issues.) Serialize only the first classifier in an ensemble as per usual. It'll contain the full graph. Then for the rest of the classifiers, record only the changes, the deltas, as compared to the first classifier. (Though this would complicate the CRF mix-and-match mechanics we used when investigating CRF knockout ensembles.)
- Consider saving, or exporting, state into PMML or some other probabilistic model modeling language.

A.3 Miscellaneous Research ideas

- Sentiment analysis. Consider Rich Colbaugh's and Kristin Glass's optimization-based approach to semi-supervised sentiment by using NER to detect sentiment-laden words? What additional features might be useful?
- Meta-NER Evaluation. In general, can we find (and build an experimental test harness around) a non-NER task that a) has a quantitative performance metric and b) would theoretical perform better with some NER (or better NER) injected?

-
- Can we make use of GLEE (Art Munson’s and Justin Basliico’s “lazy evaluation” dynamic pruning algorithm) to avoid having to consult every base classifier CRF?
 - Each base classifier assigns each term’s label to maximize the fitness of the entire sentence. When we vote, we will be looking at terms only, not the sentence. Is there any way we can get out a sentence level sense of voting? Is that even coherent?
 - In our ivoting analog, consider other strategies for assigning “hardness” and “easiness”. In particular, should we indeed normalize by the length of the sentence? (See 2011-04-01 Research Meeting.)
 - Investigate cube-summing as an alternative to link-chain CRFs? From Kevin Gimpel:

Yes, that paper is currently the only write-up I have on cube summing. It’s here: <http://www.cs.cmu.edu/~kgimpel/papers/gimpel+smith.eacl09.pdf>. The slides from the talk that I gave at the conference might be useful too: <http://www.cs.cmu.edu/~kgimpel/talks/gimpel+smith.eacl09.slides.pdf>. But honestly, neither of those discusses the implementation of cube summing very thoroughly — they’re both formal descriptions. Also, we set out to describe it in full generality for problems like parsing and translation and, as a result, it’s not quite clear how simple the algorithm actually becomes for models like skip-chain CRFs (Eq. 13 reduces to something even simpler than Eq 14, just with the non-local feature functions included). Currently there’s no code available for this, but I am hoping to release something and you’ll be the first to know when I do.

DISTRIBUTION:

- 1 MS 0359 D. Chavez, LDRD Office, 7911
- 1 MS 0899 Technical Library, 8944 (electronic copy)