CS224n
Final Project
Andrey Gusev
sunetid: agusev

Multi-Lingual Document Tagging

Project Overview:

In this report I will detail the work on "Multi-Lingual Document Tagging". The goal is to be able to pass multilingual string and get back each language substring correctly tagged and separated from the phrases in the other languages. Thus, there are two major components that are necessary for overall functionality – first a robust language detector and second boundary detector that can find language split points. Prior to the start of work on this final project I have developed subsystem that was able to do language detection. This subsystem was build over the course of two weeks very recently – one week in January and one week in early April when this class was starting. Since understanding of this system is important for boundary detection I will describe portions of it in this report as well. Furthermore, there were some modifications to that language detection subsystem that will be detailed as well. Overall I would breakdown the work for this project as

1. Porting of language detection subsystem and building necessary tools for command line interface in *LanguageDetectionTester* class (15%)
2. Attempts of tunning of language detection, significant increase in training sets (20%)
3. Multilingual test sets and boundary detection algorithms, approaches, and results (65%)

The rest of this report is structure as follows. First I introduce command line tools that can be used to run detector, generate various test sets, and run detectors on test sets. Next I will detail implementation of language detection engine. Then I will describe iterations in building language boundary detector. In this section I will also present experimental results and discuss the choices I made for algorithm tunning. In the next section I will focus on results discussion and their analysis as a whole. In future work section I will discuss the extensions that I believe can improve performance of these models and add new functionality. Finally, I discuss some related work.

Command line tools:

There are quite a few different run modes that can be executed on the *LanguageDetectionTester* class. Thus, I have created several scripts to allow for easier execution of necessary targets. All scripts are in root of submission directory. Please note that the required data is not submitted with project. All scripts reference the data with DATA_PATH variable in scripts. If you would like to copy the data, copy /afs/ir.stanford.edu/users/a/g/agusev/cs224n/langDetect/data/ to local directory and change DATA_PATH var in all the scripts.

1. **detectLang** – no parameters, takes a test string interactively and prints out confidence levels and each n-gram values for all language, the top language is the detected language
2. **detectMultiLang** – no parameters, also interactive, takes multilingual string and breaks down the string in phrases tagged with languages. Use enter (no string input) to exit. As an example you can use " *yo no hablo espanol but some people parler francais tre bien und das ist eindeutig sehr gut*"
3. **genNgramModels** – this probably doesn't need to be run again, takes original source text for

each language (collection of books from Guttenberg project) and creates character ngram models in *./data/languagemodels/ngramModel/* directoty.

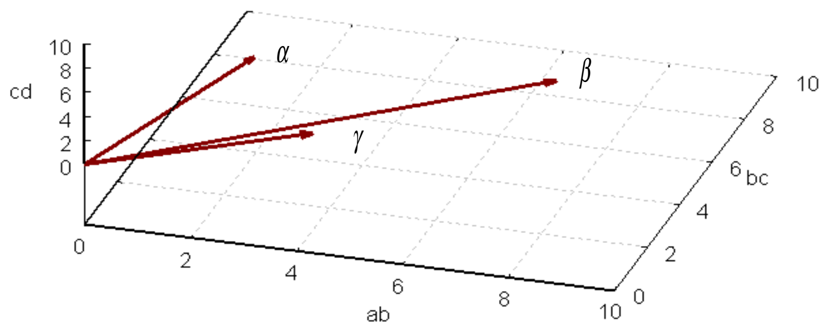
4. **genTrainingAndTestData** – this probably doesn't need to be run again, takes original source text and triggers generation of training and test data with 90-10 split. You can use -minTrainSize and -maxTrainSize parameters to change min and max phrase size. Currently by default it is set to min=4 and max=8. Increasing average sample size improves language detection accuracy.
5. **genMultiLingTestData** – this probably doesn't need to be run again, takes generated test set (from above call) and randomly generates single document with phrases from each language mixed. Each phrase is tagged with correct language. Mixed language document will contain 30000 phrases.
6. **runTestSet** – runs generated sets through language detection algorithms and reports accuracy numbers across languages for each algorithms. The parameter *-useClassifier* controls which algorithm to run. There are three algorithms and bit masking is used to determine which algorithm to run.
   // 1 - only linear classifier
   // 2 - bagged decision tree
   // 4 - logistic classifier
   // for example 7 selects all of them
   These algorithms will be discussed in more detail in next section.
7. **runMultiLingTestSet** – runs multilingual test set through boundary detector and language tagger and reports accuracy numbers on multilingual test set. Uses linear classifier for language classifier and by default used *FOUR_WORD_BIGRAM* boundary detector. Use *-boundaryDetector* parameter to change boundary detector. The following boundary detection parameters are supported *ONE_WORD, TWO_WORD, THREE_WORD, BASE_BIGRAM, TWO_WORD_BIGRAM, THREE_WORD_BIGRAM, FOUR_WORD_BIGRAM, FIVE_WORD_BIGRAM, SIX_WORD_BIGRAM, FIVE_WORD_NESTED*. These parameters will be described in more detail in boundary detection section.

Language Detection Engine:

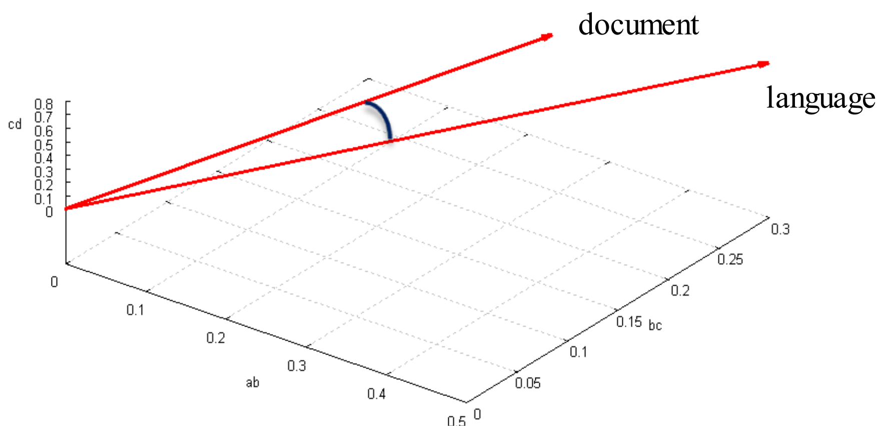
The majority of language detection engine was implemented by me prior to this final project. The implementation took place over two week span – one during January 2009 and one more week in the beginning of April 2009. However, due to the fact that this topic is relevant to NLP and boundary detection algorithm section and never was presented in a report formally, in this section I would like to describe major components of this approach.

1.0 Character ngrams

The base component of language detection is character ngrams. More precisely we break each document into 1-gram,2-grams,3-grams,4-grams and 5-grams profiles. Then we use accumulated probabilities over all ngrams for given document. We use only top ngrams. The base formula for how many top ngrams to keep for each ngrams size profile was tweaked during this final project and in fact produced better results. Currently it is set to be *Math.min(BASE_TOP_NGRAMS * this.ngramSize, 150)* where BASE_TOP_NGRAMS = 50. This is especially important when we construct probabilistic representation of language where we only want to keep ngrams that are truly representative of the language and we would like to exclude all the noise. Once we have broken down document into set of ngrams we can represent each document as a vector in k-dimensional ngram space. Each dimension is possible ngram. A value on that dimension is ngram frequency of that document. Below is fictional diagram to illustrate how these vectors will look like.



Here we see three documents which only contain there bigrams ab, bc, cd. This actually is not possible but illustrates how these documents projected into vector space will look like. Next step is to take large collection of books from Gutenberg project (roughly 2.5 MB pure text for each language) and construct ngram representation of the language. We will need to do euclidean normalization to ensure that we can compare language vector which without normalization has thousands of particular ngrams and small document which has few occurrences of ngrams. More precisely our metric to compare two vectors will be cosine similarity: $\cos(\vec{A},\vec{B})=\frac{\vec{A}\cdot\vec{B}}{\|\vec{A}\|\times\|\vec{B}\|}$. Thus we will have something like this in normalized vector space.



Where one of the vectors is representing a language and the other represents a document for which we need to determine a language. The next step is to compare test document to each of the languages and get their cosine similarity. Finally, we need to do that in 5 ngram space (1-5). Thus for each language we will have 5 cosine similarity for each ngram size. The interesting question here is how to combine these cosine similarities to create the model that has the best accuracy. I have explored three different model: basic linear proportional combination, bagged decision trees, and logistic classifier. In the end the simplest classifier – basic linear combination worked best. In this approach each weight is set to be proportional to the size of ngram, so we have $argmax\, h(x)=1*x_1+2*x_2+3*x_3+4*x_4+5*x_5$ where vector **x** represents cosine values for particular language. Thus we pick the language which maximizes this sum. I will describe some details of logistic regression in the model tunning section. Another advantage in modeling document and language through character ngrams is implicit smoothing. We don't have to observe particular word to classify it correctly. Thus while the model is compact it can model and classify correctly unseen events. Overall, character ngram models allow for good accuracy rates and good runtime due to the compact model structure.

2.0 Implementation details

In this section I will describe some implementation details of language detection. This implementation is quite large (**14 classes**) but since this part does not represent the core of the project, the description will not cover all the details and will focus on very high level overview. The core of the ngram model is in *language.mode* package. *NgramModel* class implements character ngram model functionality. The main method in this class is *calculateCosineSimilarity(NgramModel anotherModel)* that returns double which represent cosine similarity between given ngram model and another model. This method is called when comparing document model with language model. Language model is only calculated once and is saved on disk in normalized form. Two classes which are used in bagged decision classifier and logistic classifier are *LanguageDocumentExample implements DecisionTreeExample<Double, Locale>* and *NgramLanguageModelFeature implements DecisionTreeFeature<Double>*. Feature class defines the features of particular training example which are – 1-gram,2-gram,...,5-gram, and linear combination. Example class defines one training example which is labeled with particular language and has set of cosine similarity values for each feature for each language. Essentially each example represents one phrase along with all the cosine similarities and its true language label. *NgramLanguageDetector* is class that probably has most functionality and interacts with *NgramModel*. Some of the more important methods are – *generateLanguageModels*, *generateTrainingAndTestData*, *generateMultiLingualTestData*, *runTestSet*, *runMultiLingualTestSet*, *detectLanguageWithDecisionTree*, *detectLanguageWithLogisiticClassifier*, *detectLanguageWithLinearWeights*, *getRawCosineSimilarities* and several other methods that wrap around these methods. **All the multilingual code is new for this project.** All generative methods generate appropriate data set and save it on disk. The testing methods *run** run appropriate data set and report accuracy results to stdout. Although I think there are some nice details that are important to describe in this class, I don't believe I will be able to describe all functionality with this ~1000 line class within reasonable report length. The package *language.classifier.tree* package contains 5 classes and all the code that grows decision tree based on splitting to minimize entropy of training data set. This is standard decision tree machine learning implementation. It operates on *LanguageDocumentExample* and *NgramLanguageModelFeature* classes as described above. Finally package *language.classifier* contains 4 classes that are used for two machine learning classifiers - bagged decision tree and logistic regression classifier. Bagged decision tree samples training set with replacement to generate k decisions trees – the final hypothesis is the combination of these k decision trees. Currently k is set to 10. Logistic regression trains one vs all classifier for each language on the training set. Each example has all the cosine similarity values thus in theory should assign very negative weights to cosine similarity for labels that are not positive for classifier and vice versa. Since neither logistic classifier nor bagged decision tree approach outperformed linear classifier in any significant way I did not use them for boundary detection. However, in the next section I will talk about some changes that were done to this core functionality for this final project.

3.0 Language Detection Changes

In this section I will discuss some of the attempts to improve on this core language detection functionality. Originally I have used about 250KB worth of training data for each language – this was sufficient to generate correct language representation via character ngrams. Since we only keep at most 150 ngrams for each ngram set, relatively small amount of text is expected to correctly represent hypothesis that is close to true language ngram distribution. In fact when I increase data set – the change in distribution was small. On the other hand for training various classifiers via small phrasal chunks 250KB – which represent roughly one medium size book was clearly not enough. Thus, one of the first things that I did was significantly increase training data size – roughly 10x for each language to

~2.5MB for each language. All the source came from Gutenberg book project and represents roughly 10 books for language. I have tried varying books to be from different time periods as well as domains. Although precise list of books can be recovered since we're approaching this problem from probabilistic point of view the exact list of books is not as relevant for this functionality. Next change that I did was to reduce set of supported languages to 6 – EFIGS+P – English, French, Italian, German, Spanish, and Portuguese. This actually made the problem more difficult since I have removed the easy languages such as Japanese, Russian, Thai. These languages can be distinguished purely on character set – i.e. character unigram model. Despite that, just by increasing data set for each language I was able to bring accuracy of most language up by several percentage points. Before describing other changes I would like to present accuracy numbers for those changes so there is context for the description. All the accuracy numbers are on very short phrases taken from monolingual text. Each phrase is 4-8 word long (length is determined randomly). Initially 60 percent of phrases was used in training corpus and the remaining 40 percent in test corpus. Since I drastically increases overall data from which training and test sets were created I changed distribution to 90% training and 10% test set. I felt that I could do that because the test set was still large and thus presented statistically significant results. I also really needed training set to be as large as possible both for logistic regression training and more importantly for language boundary training.

Test Run	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
English	0.894	0.914	0.924	0.929	0.921	0.920	0.914	**0.907**
French	0.897	0.912	0.870	0.877	0.866	0.915	0.907	**0.918**
Italian	0.917	0.916	0.839	0.854	0.824	0.909	0.896	**0.917**
German	0.988	0.969	0.942	0.951	0.937	0.968	0.967	**0.965**
Spanish	0.743	0.802	0.796	0.779	0.785	0.814	0.797	**0.821**
Portuguese	0.838	0.847	0.792	0.776	0.736	0.846	0.837	**0.853**
Overall	0.876	0.892	0.860	0.861	0.845	0.895	0.886	**0.897**

1. Original base line – this is how the project was ported and the number of languages supported reduced to 6 languages. This is linear classifier (i.e. not machine learning classifier).

2. Increasing corpus data 10x clearly improved the accuracy across languages. Since this accuracy data is also for linear classifier, the most likely reason for improved accuracy is that more data was able to capture true ngram distributions more accurately. Furthermore, more data also contributed to less noise captured in top ngrams for language vectors. More discussion of limiting ngrams is in item 7 and 8.

3. In this set of data I started re-tunning logistic classifier. Original classifier for ~20 language performed in-line with linear classifier, slightly outperforming linear classifier on short strings. Thus I had high hopes for logistic classifier, especially with significant increase in training phrases. For each language there is one vs all logistic classifier that operates on 6*5 +5 = 35 features. There are 5 ngram values for each language, 6 languages, and I also added linear combination feature (the sole feature that is used for linear classifier) for each language. The implementation of batch gradient descent is in *LogisticRegressionClassifier* class. Thus the task of batch gradient descent training is to find appropriate weights for all 35 features. Ideally the features that correspond to language of particular classifier would be given higher weight and vice versa. This data represent 50 iterations for logistic classifier. As can be seen the accuracy is below that of simple linear combination classifier (1 and 2). I attributed it to not enough training as I could see during the training output that sum of all updates to weights still was non zero thus, training was interrupted too early.

4. In order to ensure that we don't interrupt training too early – I increased the limit to 200 iterations for logistic classifier. As can be seen, overall accuracy increased very slightly.
5. This data represents 500 iterations for logistic classifier. We can note that the accuracy actually went down somewhat. This can be attributed to over-fitting training data. The error on training data was less than 0.05 for all language and less than 0.01 for couple languages – thus since we have such large discrepancy between training and test error it is reasonable to assume to have over-fitting problem. At this point I made a decision to not invest any more time in getting logistic classifier to perform better. The accuracy rate with linear combination classifier was acceptable and as I saw later the focus on the core of this project - actual boundary detection algorithm was far more important than little increase in accuracy of language detection. All the items below describe changes to linear classifier.
6. This data was generated with addition of 6-gram models. As can be seen the increase in accuracy was very little. The observation that I made is that while 6-gram character model does introduce some new combination almost all of the top character combinations can be found in 5-gram model. For example the word "that" was present in 6-gram as "$that$" - where $ represent word boundary. However, in 5-gram model it is also present as "$that" and "that$". Thus, as expected 6-gram model did not increase accuracy much and introduced the potential of extra noise. Thus, I decided to not use 6-gram model and use 1-5 ngram models.
7. In this data set I started looking at how max limit on number on ngrams stored for a model impacts accuracy. The idea was that the fewer ngrams we store the less chance of a noise there is. This data represents limit of 100 ngrams for any ngram length. Since we took a base size of 30 and multiplied it by size of ngram that means only 4-grams and 5-grams were affected. The accuracy decreased but I continued experimenting with this and as will be discussed in item 8 found a combination that performed better.
8. This data set represents changing base size for number of ngrams in a model to 50 and setting the limit to 150 ngrams. Thus, for unigrams we stored up to 50 (actually less since alphabets contain fewer symbols), for bigrams we stored 100, and for 3-grams, 4-grams, and 5-grams we stored 150. The improvement in accuracy that we see is due to two factors. First, since unigrams, bigrams, and trigrams are short and thus are not likely to contain entire word that is specific for the data corpus but is not representative of language, by increasing the limit we capture more of true language ngram distribution. On the other hand, by limiting number of ngrams that are kept for 4-gram and 5-grams models we are also limited the amount of noise that can appear in those models due to specific words that happened to be common for data corpus but not representative of the language distribution of ngrams. Limiting of the noise is also really important since, throughout the entire development of language detection, ngram character models were generated over entire textual corpus. Thus, we want to keep models as representative of language as possible so we can correctly report on accuracy of language detection of test phrases which are also sampled from entire corpus. I believe this is valid approach since generating the models over entire corpus produced very high compression rate (from 2.5 MB of text to 100 or so ngrams) so that all specific features of the corpus are generalized and smoothed out. On the other hand by having as much data as possible for model generation contributed to better modeling of the language. I have looked over the models and did not notice any specific noise that would represent particular part of textual corpus. That is ngrams represented generic words and parts of words such as "th", "ing", "er", etc. Overall, base=50 and limit=150 produced the best accuracy numbers and those are the settings that I used for language boundary detection.

Since I have not change implementation of bagged decision tree classifier and have not used it for language boundary detection I will not discuss details of this code in this report.

Language Boundary Detection Engine:

1.0 Overview of Algorithms

In this section I will discuss major part of this project – language boundary detection algorithms. These algorithms can be grouped into four types. More precisely *ONE_WORD*, *TWO_WORD*, *THREE_WORD* are algorithms that use language detection sliding window only as boundary detection mechanism. On the other hand *BASE_BIGRAM* algorithm uses only word bigrams to find language boundary. Neither of these base algorithms performs very well – however when combined present better results. Algorithms *TWO_WORD_BIGRAM*, *THREE_WORD_BIGRAM*, ***FOUR_WORD_BIGRAM***, *FIVE_WORD_BIGRAM*, *SIX_WORD_BIGRAM* represent combination of bigram word model and language detection sliding window. The best performance was achieved with *FOUR_WORD_BIGRAM* algorithm. Finally the most complex algorithm type is represented by *FIVE_WORD_NESTED* algorithm that uses multiple sliding windows as well as word bigram model. Each of these algorithm type is described in more detail below.

2.0 Sliding Window Language Detection Algorithms

The first set of boundary detection algorithms that were implemented are pure sliding window language detection algorithms. The simplest algorithm - *ONE_WORD* implemented in a class *OneWordLanguageBoundaryDetector* uses a window length of one word and runs language detection algorithm on that word. As long as the detected language stays the same the phrase is accumulated, otherwise the phrase is added to list of tagged phrases and new phrase is started. In theory if language detection was perfect for single word than this would be sufficient way to find boundaries – however, in practice the accuracy of language detection on single word is low. This is understandable as single word without any context is ambiguous – there are many identical words across languages, for example "come", "stop", and many others. Thus this boundary detection algorithm has a very low F1 value of ***0.018***. More detailed result analysis will be presented in results section. Next two algorithm from this group are *TWO_WORD* and *THREE_WORD* – the idea is the same however the window length is larger. This clearly improves the language detection accuracy however introduced a new problem. Lets say that after advancing three word window we see that detected language changed compared to previous window. Within that window where is breaking point between languages? Is that between first and second word or between second and third word? It could also be potentially even before the first word of the window. For these algorithms I used a reasonable heuristic where the breaking point for two word window was between words. The breaking point for three word window was between first and second word. The assumption for this boundary selection was that all words have relatively the same weight in language detection and thus two words in window of length three are more likely to represent new phrase. This assumption is of course is too simple thus a more sophisticated boundary selection within a window is necessary. The F1 rates for window of length two and window of length three are ***0.041*** and ***0.057*** respectively. Next type of algorithms focus on finding boundary in more sophisticated way.

3.0 Word Bigram Algorithm

This algorithm constructs a new type of model to improve on boundary detection. The **word bigram** model is trained on training corpus – which is 90% of original corpus and is not used for phrases in test multilingual document. Note that we are talking about word bigrams and not character bigrams. The implementation of this algorithm is in class *BaseBigramLanguageBoundaryDetector* and can be used with *BASE_BIGRAM* value for *boundaryDetector* parameter of *runMultiLingTestSet* script. The phrases

are broken when particular word bigram has not been observed. Language detector is then ran just to tag phrase with a language that has already been broken down by word bigram model. Thus we make an assumption that word bigrams across languages will not be observed and thus will be good indicator of language boundary. In general this assumption is valid and indeed produced a F1 score of ***0.168*** which is much higher than F1 scores for pure language detection sliding window algorithms. However, this approach does have some drawback – first it is very possible that particular valid language bigram will not be observed in training and will thus cause a phrase in single language to be split. On the other hand it is also possible that two phrases in different language when concatenated may create a bigram between them that actually has been observed in some language. To validate this I actually ran one test when training bigram model on *test* data. This would eliminate the first problem as all language bigrams in test set are observed. Thus, only second problem could cause a decrease in accuracy and indeed instead of F1 measure of 1.0 it was about 0.8. For error analysis when running multilingual test set – there are two files that are generated: one that has exact list of correctly tagged phrases and the other that has list of language tagged phrases by boundary detection algorithm. Diffing the two files in this case indeed shows that some phrases in two language are concatenated into one phrase. Another problem with pure word bigram approach is that since it has no knowledge of language before language detection algorithm tags the phrase it actually generates quite a lot phrases. The relatively high F1 measure compared to sliding window algorithm is due to high recall – the precision on the other hand is about 0.12. In the next section I will describe set of algorithms that combine both sliding window language detection and bigram word boundary to improve the accuracy of language boundary detection.

4.0 Sliding Window With Bigram

The algorithm in this section combines both bigram word boundary detection and language detection sliding window. The implementation of this algorithm is in *SlidingWindowWithBigramLanguageBoundaryDetector* class and can be used with *TWO_WORD_BIGRAM, THREE_WORD_BIGRAM, FOUR_WORD_BIGRAM, FIVE_WORD_BIGRAM, SIX_WORD_BIGRAM* values for *boundaryDetector* parameter of *runMultiLingTestSet* script. These parameter values correspond to different length language detection windows. The algorithm is structured as follows – first the sliding window of given length is used to detect language. When language changes compared to previous window we assume that boundary is somewhere inside that window. Thus we have window.length – 1 language boundary points. To determine which one is correct boundary point we defer to word bigram model. There are three cases which can occur for boundary detection within a window.

- There is exactly one boundary point for which word bigram has not been observed. This is the best case and we will break on that boundary point.
- There is more than one boundary which does not have any word bigrams in training data. In this case we have preference order to determine which boundary point to take. The preference order is defined for each supported window length (2-6) and roughly corresponds to heuristic that words tend to have equal weight in language detection. Lets say we have window of fours words "A B C D" then we have the following boundary points "A2B1C0D" where numbers correspond to boundary points. Below is the mapping from window length to boundary preferences.

```
boundaryPreferenceOrder.put(2, new int[] { 0 });
boundaryPreferenceOrder.put(3, new int[] { 1, 0 });
boundaryPreferenceOrder.put(4, new int[] { 1, 2, 0 });
boundaryPreferenceOrder.put(5, new int[] { 2, 1, 3, 0 });
boundaryPreferenceOrder.put(6, new int[] { 2, 3, 1, 4, 0 });
```

- Finally there is possibility that we will have word bigrams for all boundary points – in this case

we say that our language detector made a mistake and we move on without breaking this phrase. This change increased precision of language detection.
Overall this type of algorithm works best on given test data. The best performing combination is *FOUR_WORD_BIGRAM* since it matches the low boundary of length of phrases in multilingual test document. This combination achieves F1 score of ***0.193***. ***This may appear like a low score but it is actually somewhat misleading***. Empirically analyzing the output we can see that some portion of phrases that didn't match the actual phrase were one word off. More detailed discussion of this metric will be in results section. Also I believe that with longer phrases longer sliding window would work better since the accuracy of detection goes up significantly with increase in window size. In the next section I will describe algorithm which uses both long and short language detection windows as well as word bigram language model.

5.0 Nested Sliding Window With Bigram

This section describes the algorithm that doesn't perform as well as algorithms described in previous section on very short words but is expected to outperform those algorithms on text where language changes are much less frequent. The implementation of this algorithm is in *NestedSlidingWindowWithBigramLanguageBoundaryDetector* class and can be called with *FIVE_WORD_NESTED* value for *boundaryDetector* parameter of *runMultiLingTestSet* script. In this case five word is just an example and the idea can applied to much larger language detection windows. This algorithm is based on observation that language detection becomes highly accurate when given enough context. In my original implementation which supported ~20 languages accuracy rate for strings with 15-20 words was about 0.99 for almost all languages. Thus if we increase the length of boundary detecting sliding window to 20 or more words we can be sure that when the window signals that there is change in language that indeed the language does change. However having a large window also presents a problem of a lot of potential breaking points within that window. For example if we use highly accurate window length of 50 words we will have 49 potential breaking points. The next step is to take this context of 50 words and scan it again with smaller window – for example 10 words. After that even smaller sliding window can applied until actual boundary is found. The advantage of this approach is for texts that tend to have very infrequent language changes, first large sliding window finds this change and the smaller windows try to find the actual boundary point. The implementation of this algorithm is just illustration of this concept with only two window size – larger is 5 followed by a smaller window of length 3. In fact the smaller window algorithm is directly *THREE_WORD_BIGRAM* boundary detection algorithm discussed in previous section. Thus the smaller window uses word bigrams for final boundary verification. Since multilingual data set that I use is composed of very small phrases this algorithms does not do as well as algorithms in previous section. The F1 score for this algorithm for this data set is ***0.116*** which is lower than the results in two previous sections.

6.0 Results Overview

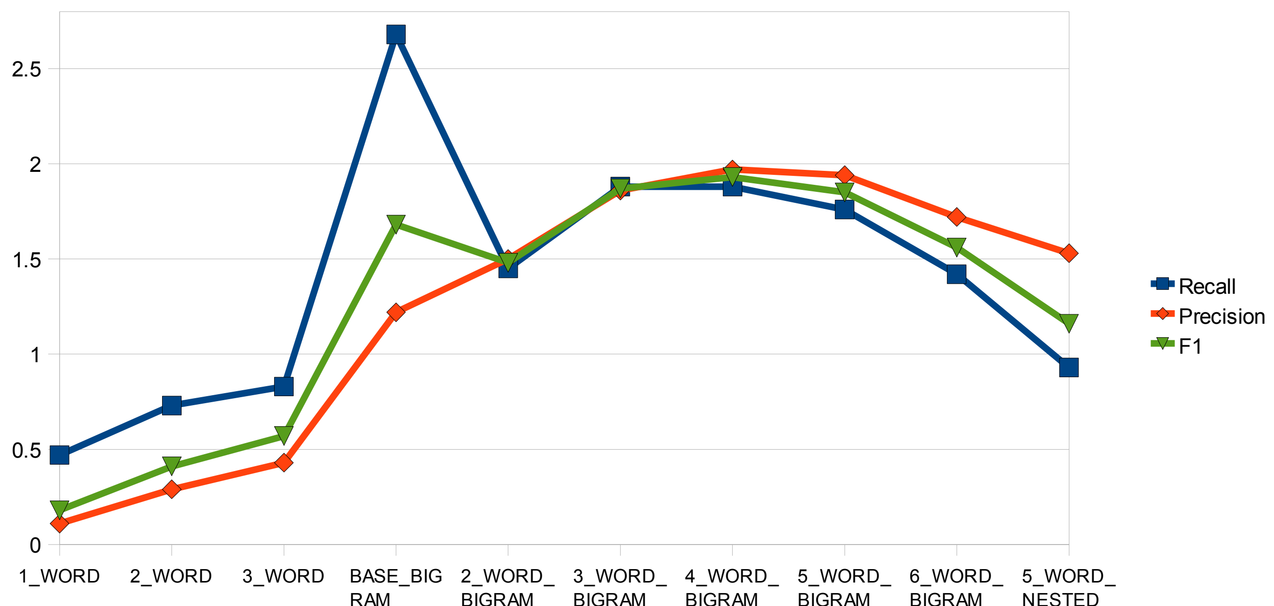
In this section I would like to present experimental results for language boundary detection algorithms described above. First it is important to discuss the main metric used in evaluating boundary detection algorithms. While I could have used a metric that gave partial score to boundaries that were very close I decided that it is clearer to present results that indicate how many exact phrase matches were found by boundary detection algorithms. However this metric tends to produce low results when boundary detection does not exactly match expected phrases. For example lets we have 10 words – first 4 are in one language and 6 words after that are in another language. If language boundary detector places the

boundary between words 5 and 6 – then we have missed both phrases. So precision and recall would be 0/2 for both metrics. However we can also see that in fact only one word out of 10 was mis-categorized. Since I wanted to capture boundary detection accuracy rather than word categorization I decided to keep the current metric and instead of adding more metrics focus on more boundary detection algorithm development and tweaking. Thus for results that are presented below it should be noted that values are for matches on exact phrases.

Algorithm	**ONE_WORD**	**TWO_WORD**	**THREE_WORD**	**BASE_BIGRAM**	**TWO_WORD_BIGRAM**	**THREE_WORD_BIGRAM**
Found	107490	63726	47666	55164	24233	25351
Misclassified	20	26	30	210	171	211
Correct	1174	1840	2071	6730	3647	4709
Recall	0.047	0.073	0.083	0.268	0.145	0.188
Precision	0.011	0.029	0.043	0.122	0.150	0.186
F1	0.018	0.041	0.057	0.168	0.148	0.187

Algorithm	**FOUR_WORD_BIGRAM**	**FIVE_WORD_BIGRAM**	**SIX_WORD_BIGRAM**	**FIVE_WORD_NESTED**
Found	23991	22794	20690	15227
Misclassified	227	205	175	126
Correct	4724	4418	3560	2333
Recall	0.188	0.176	0.142	0.093
Precision	0.197	0.194	0.172	0.153
F1	0.193	0.185	0.156	0.116

Found row represents how many phrases were returned from boundary detector. Misclassified row represents how many of the phrases that were found correctly were tagged with wrong language. Correct row shows the number of phrases that were both correctly found and tagged with right language. The total number of phrases in the document was 25073. This number is less than 30000 since consecutive phrases in the same language were merged and considered as the same phrase. Below is the graphical representation of precision, recall and F1 values for these algorithms. Note that all values on Y axis are multiplied by 10 to avoid rounding in the graph.



Looking at the graph we can note a very high recall rate for *BASE_BIGRAM* boundary detector. Although this is clearly a good part of that algorithm we should also note a very low (relative to recall) precision rate. We can also note that it generates twice as many phrases as there actually are and out of all algorithms it has the third highest number of phrases. Thus, although *BASE_BIGRAM* appears like one of the better performs in fact it benefits from small average phrase size. I have ran some additional test with other phrase size and noticed that in general the performance of this algorithm tends to get worse as phrase becomes longer. This is expected since unlike language detection driven boundary detector it has no context of phrase being in the same language and simply breaks the phrase based on observed word bigram data. Therefore this algorithm is more likely to generate phrase of size one or two than most language detection sliding window algorithms. The next interesting trend that we can observe is that algorithms that combine both sliding window and word bigrams for boundary detection generate very accurate number of phrases. For example *THREE_WORD_BIGRAM* algorithm generates 25351 phrase while the actual number is 25073. Thus, it may be little surprising that F1 values for these algorithms never exceeds 0.2 – this can be explained by the metric that we use. Since we're a looking for exact phrase match, missing a boundary by one word on all phrases would still generate the same number of phrases but will also generate value of 0 for F1 measure. Another trend that is interesting is that increasing window length beyond 4 on this data set tends to have negative impact on accuracy numbers. Since our test data can have phrases that are as short as 4 words, a window that is longer than that length can potentially have parts of three phrases in the window – this can negatively impact accuracy of language boundary detection. Finally I would like to present an example of input for interactive script which takes user input as a string an returns tagged strings back. Here is one example input

"yo no hablo espanol but some people parler francais tre bien und das ist eindeutig sehr gut"

This translates to:

I don't speak Spanish but some people speak French very well and this is clearly quite good.

This is tagged output – this output is correct.

yo no hablo espanol -> es
but some people -> en
parler francais tre bien -> fr
und das ist eindeutig sehr gut -> de

Overall, the set of algorithms presented applies different combination of methods to try to tag very short phrases with their language. Since the document contains very short phrase and the accuracy metrics only looks for exact matches, the accuracy rates are not high. However, for some phrases and algorithms although precise phrase was not matched analyzing the output empirically showed that the boundary was very close to it's expected location within the string.

<u>Future enhancements and work:</u>

In this section I would like to detail two changes that I believe can improve the performance of overall system. First, I believe with proper tunning, training and perhaps extra features, logistic regression classifier should outperform linear weight classifier. This is based on the fact that logistic classifier has more information (all cosine similarity values for phrase) and thus with adequate training data should

be able to better learn hypothesis that is able to classify document better. The next enhancement that can have positive effect on overall system is tunning of the nested boundary detector. It is not likely that it will be able to produce good results on short phrases that were used for this project but with test data which has infrequent language changes I believe a form of nested sliding window boundary detector is likely to produce the best results. Overall, in order to develop more sophisticated language detection algorithms and language boundary detectors more type of testing and training data should be used. This should allow for development of more generic algorithms and systems.

Related work:

In this section I will briefly mention some of the related work. Please note that large number of language detection systems are commercially implementation and the source and approach are not shared. Here is the list of related work with some comments.

1. TextCat is an implementation of the text categorization algorithm presented in Cavnar, W. B. and J. M. Trenkle, ``N-Gram-Based Text Categorization'' In Proceedings of Third Annual Symposium on Document Analysis and Information Retrieval, Las Vegas, NV, UNLV Publications/Reprographics, pp. 161-175, 11-13 April 1994.

This system takes on very similar approach for building ngram profiles that I used– that is it uses character ngrams padded with end word/beginning of the word markers. It keeps larger number of ngrams for each ngram profile - 300. However, in my tests such larger number introduced too much noise so I keep at most half of that – 150. The paper also suggest using even smaller sample text to build ngram profiles. While I used 2.5 MB test chunks, the paper discusses that 10K or 20K chunks might be sufficient. The biggest difference occurs with similarity metric where one of the techniques that is discussed involves distance measure of all ngrams. On the other hand I use cosine similarities to compare ngrams vectors derived from ngram profiles. This paper is also not strictly focused on language detection but rather covers text categorization as well. This paper also does not cover boundary detection.

2. Language detection using character trigrams, by Douglas Bagnall
http://code.activestate.com/recipes/326576/ -

This blog describes system that uses the same approach to build document profiles as the system in this report; however, it uses only single ngram size - trigrams. The accuracy numbers are not presented but according to my experiments using only trigrams produced less accurate system than system which uses more ngram sizes. This system does not do language boundary detection.

3. Text Classification using String Kernels (2002) by Huma Lodhi, Craig Saunders, Nello Cristianini, John Shawe-Taylor, Chris Watkins, Pack Kaelbling.

This paper is only partially relevant to the system described in this report. One relevant idea described in this paper is building kernels from document which generalizes building of ngrams. A kernel for text sequence is a substring of word but it doesn't have to be contiguous. How contiguous substring is determines the weight that substring has. A decay factor is described in this paper that determines the weight. Thus, this approach can be applied in building both document and language profiles. These profiles could then compared with another metric to determine language of the document.