Implementing Coptic Word Segmentation

Karl Mutch

UCB Introduction to Web Algorithms, 2010.

GOAL:

This weeks' project was chosen to augment work that I am currently doing to store a repository of Coptic, Demotic, and Greek texts within a searchable database. One potential challenge with doing this is that texts being input may lack word separation which is a standard feature of these languages.

INTENDED RESULTS:

The goal is to implement a word separation method using LingPipe. A set of training documents will be input containing word separated UTF-8 input. The training documents will be presented to LingPipe and used to then segment test documents successfully.

WEEK 3 : TASKS

The first tasks involved research on means by which segmentation can be done.

The second task involves implementing the approach discovered.

WEEK 2 : DIARY

RESEARCH AND COMMENT

Word segmentation in a supervised environment typical involves the use of dictionaries and rules based systems for segmenting texts. This typically requires specialized skills within each language. Given the distributed nature of individuals within antiquity studies bringing together these skills is a complex and hard to co-ordinate task. Because of this supervised approaches required significant co-ordination.

To overcome this we will try to investigate the feasibility of using a semi supervised, or if possible an unsupervised approach.

Using a learning based system within this context might be possible due to the presence of many documents in both Coptic and Greek that have word segmentation done already. Demotic on the other hand does remain a

challenge that because of the state of various standards cannot yet be done in this context, something we will have to solve at a later date.

I did discover a number of papers related to learning systems of this type including the following which I will try to summarize:

- One hybrid approach calls for using a predefined word list along with a set of rules expressing bias and a genetic algorithm. This is then used with decision list learner to produce a set of rules for segmentation, more details can be found at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.1257.

- A pure approach is described in "Bayesian Unsupervised Word Segmentation with Nested Pitman-Yor Language Modeling", http://chasen.org/~daiti-m/paper/acl2009segment.pdf.

  This approach uses a Bayesian approach with a dynamic programming based Gibbs sampler. The model used is one specially intended for computation linguistic use. This looks promising but is way too much for a 1 week stint.

- Another approach that is close to satisfying our constraints is described in A compression-based algorithm for Chinese word segmentation. Computational Linguistics 26(3):375-393, http://acl.ldc.upenn.edu/J/J00/J00-3004.pdf.

The third approach has an analog in the world of LingPipe. The approach is essentially the same as training a spell checker to detect the absence of word breaks, i.e. spaces in the text. It also has a prototype implementation in the LingPipe tutorials and has been used in a number of bakeoffs for Chinese text with good results.

The third approach that I used within this week's project has a potential weaknesses common to un-supervised approaches.

- Language forms between speech and written language. The corpus of the texts being processed can be tightly controlled to only include written texts that style wise are very similar with the texts to be segmented. Typically we can for example categorize texts as graffito, business documents, histories, narratives.

PROJECT TEST

The test involves taking the original training examples from the LingPipe Chinese training bakes offs and adapting these for use with our Coptic documents from Week 1's project. Additional documents have been added that are of a religious nature and extend the training set, this helps us because we remain within a category of texts that are fairly uniform.

A unit test will be performed where words being segmented have been seen. We also during the test will examine the results to discover if we see unique phrasing for a test case where words have not been explicitly observed previously.

Input texts use, in addition to Coptic characters, cues in the form of ASCII characters that allow translators to indicate parts of speech important to translation etc. These characters are removed to allow the spelling checker to segment words. These characters could be reconstructed using a different trainer to allow their insertion automatically but this was not done for this project.

IMPLEMENTATION

The following implementation contains all of the customized code for our solution. It should be noted that individual classes can be broken out from this file.

Three main methods are provided to allow for word segmentation. The first is used to prepare the spell checker and to compile it followed by persisting the result Object, CompileSpellChecker. The next method loads the compiled spell checker and then runs unsegmented text through it, testSpellChecker. The last method prints the test results and statistics, printResults.

The infrastructure of the test itself and printing results claims most of the real estate of the implementation. The core training and test are a relatively small portion of the implementation.

```
/**
 * @author karl Mutch
 *
 * Assignment for Week 4 of the Web Algorithms Class Summer 2010
 */
import com.aliasi.classify.PrecisionRecallEvaluation;

import com.aliasi.lm.NGramProcessLM;

import com.aliasi.spell.CompiledSpellChecker;
import com.aliasi.spell.FixedWeightEditDistance;
import com.aliasi.spell.TrainSpellChecker;
import com.aliasi.tokenizer.RegExTokenizerFactory;
import com.aliasi.tokenizer.TokenizerFactory;

import com.aliasi.util.AbstractExternalizable;
import com.aliasi.util.Compilable;
import com.aliasi.util.Files;
import com.aliasi.util.ObjectToCounterMap;
import com.aliasi.util.Streams;
import com.aliasi.util.Strings;
import com.aliasi.util.Tuple;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

import java.io.BufferedOutputStream;
import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.ObjectInput;
import java.io.ObjectInputStream;
import java.io.ObjectOutput;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.io.OutputStreamWriter;
import java.io.Writer;


/**
 * This class forms the implementation of Week 4's
 * homework.  This class will test the notion of
 * using the LingPipe spell checker to perform
 * word segmentation tasks for Coptic based upon a small
 * corpus of documents.
 *
 *
 */
public class CopticTokens
{
    /** The following variables are used to track the performance of this test*/
    PrecisionRecallEvaluation mBreakEval = new PrecisionRecallEvaluation();
    PrecisionRecallEvaluation mChunkEval = new PrecisionRecallEvaluation();
```

```
    ObjectToCounterMap<Integer> mReferenceLengthHistogram = new ObjectToCounterMap<Integer>();
    ObjectToCounterMap<Integer> mResponseLengthHistogram = new ObjectToCounterMap<Integer>();

    Set<Character> mTrainingCharSet = new HashSet<Character>();
    Set<Character> mTestCharSet = new HashSet<Character>();
    Set<String> mTrainingTokenSet = new HashSet<String>();
    Set<String> mTestTokenSet = new HashSet<String>();

    // Parameter values used during the test
    String mInputDirectory;
    String mCorpusName;
    File mOutputFile;

    File mKnownToksFile;
    Writer mOutputWriter;
    int mMaxNGram;
    double mLambdaFactor;
    int mNumChars;
    int mMaxNBest;

    String mCharEncoding = "UTF-8";

    public String mModelFileName = null;

    /**
     * @brief Token generation method
     *
     * This method can be used to generate both character and word token sets from lines
     *
     * @param charSet Output variable to contain a collection of all observed characters from the line
     * @param tokSet Output variable to contain the words observed in the input line
     * @param line Input variable containing the input line to be tokenized
     */
    static void addTokChars(Set<Character> charSet,
                            Set<String> tokSet,
                            String line) {
        if (line.indexOf("  ") >= 0) {
                String msg = "Illegal double space.\n"
                    + "    line=/" + line + "/";
                throw new RuntimeException(msg);
        }
        String[] toks = line.split("\\s+");
        for (int i = 0; i < toks.length; ++i) {
            String tok = toks[i];
            if (tok.length() == 0) {
                String msg = "Illegal token length= 0\n"
                    + "    line=/" + line + "/";
                throw new RuntimeException(msg);
            }
            tokSet.add(tok);
            for (int j = 0; j < tok.length(); ++j) {
                charSet.add(Character.valueOf(tok.charAt(j)));
            }
        }
    }

    /**
     * A parameterized utility method to perform functor like evaluations of the inputs
     *
     * @param <E> Type for the items to be iterated through the functor
     *
     * @param evalName
     * @param refSet The first of a pair of collections to be passed into the eval functor
     * @param responseSet The second of a pair of collections to be passed into the eval functor
     * @param eval Evaluation functor
     */
    static <E> void prEval(String evalName,
                           Set<E> refSet,
                           Set<E> responseSet,
                           PrecisionRecallEvaluation eval) {
        for (E e : refSet)
            eval.addCase(true, responseSet.contains(e));

        for (E e : responseSet)
            if (!refSet.contains(e))
                eval.addCase(false, true);
    }
```

```java
    // size of (set1 - set2)
    /**
     * @brief method to compare two collections and return the differences between each of the items
     *
     * @param <E> A type to parameterize the items within the collection
     * @param set1 The first of the sets to be diff'd
     * @param set2 The second of the sets to be diff'd
     * @return The result of the differences of the sets items
     */
    static <E> int sizeOfDiff(Set<E> set1, Set<E> set2) {
        HashSet<E> diff = new HashSet<E>(set1);
        diff.removeAll(set2);
        return diff.size();
    }

    /**
     * @brief Parse lines from an input device and process them into a collection of tokens
     *
     * This method retrieves lines from an input device and parses the tokens in the input
     * into one collection of unique characters and another of unique words seen.
     *
     * @param in Input device from which to retrieve lines of characters
     * @param charSet The output set of observed characters
     * @param tokenSet The output set of observed tokens
     * @param encoding The encoding used on the input typically "UTF-8"
     *
     * @return An array containing the lines parsed having also been normalized
     *
     * @throws IOException
     */
    static String[] extractLines(BufferedReader in, Set<Character> charSet, Set<String> tokenSet,
                                 String encoding)
        throws IOException
    {
        ArrayList<String> lineList = new ArrayList<String>();
        String refLine;
        while ((refLine = in.readLine()) != null) {
            String trimmedLine = refLine.trim() + " ";
            String normalizedLine = trimmedLine;
            normalizedLine = normalizedLine.replaceAll("\\++", "");
            normalizedLine = normalizedLine.replaceAll("\\=+", "");
            normalizedLine = normalizedLine.replaceAll("\\s+", " ");
            lineList.add(normalizedLine);
            addTokChars(charSet, tokenSet, normalizedLine);
        }
        return lineList.toArray(new String[0]);
    }

    /**
     * @brief Extracts segment information from an input string that is space separated
     *
     * @param xs Input line to be examined for spaces
     *
     * @return A set of position information of spaces within input string
     */
    static Set<Integer> getSpaces(String xs) {
        Set<Integer> breakSet = new HashSet<Integer>();
        int index = 0;
        for (int i = 0; i < xs.length(); ++i)
            if (xs.charAt(i) == ' ')
                breakSet.add(Integer.valueOf(index));
            else
                ++index;
        return breakSet;
    }

    /**
     * @brief Method used to extract segmented chunk information from input string
     *
     * This method used during the reporting phase to output histogram information
     * as strings are segmented during testing.
     *
     * @param xs String to be processed and examined for chunking information
     * @param lengthCounter Segmentation lengths
     *
     * @return
     */
    static Set<Tuple<Integer>>
        getChunks(String xs, ObjectToCounterMap<Integer> lengthCounter)
```

```
    {
        Set<Tuple<Integer>> chunkSet = new HashSet<Tuple<Integer>>();
        String[] chunks = xs.split(" ");
        int index = 0;
        for (int i = 0; i < chunks.length; ++i) {
            int len = chunks[i].length();
            Tuple<Integer> chunk
                = Tuple.create(Integer.valueOf(index),
                               Integer.valueOf(index+len));
            chunkSet.add(chunk);
            index += len;
            lengthCounter.increment(Integer.valueOf(len));
        }
        return chunkSet;
    }



    /**
     * @brief Class used to implement parameterization for the spell check
     *
     * This class is used to implement a parameterized spell checker that will seek
     * to use space insertion as the sole option available to allow corrections
     */
    public static final class SpacingTokenizing extends FixedWeightEditDistance
                                                implements Compilable
    {
        private static final long serialVersionUID = 3088147751063973470L;

        private final double mBreakWeight;
        private final double mContinueWeight;

        public SpacingTokenizing(double breakWeight, double continueWeight) {
            mBreakWeight = breakWeight;
            mContinueWeight = continueWeight;
        }

        @Override
        public double insertWeight(char cInserted) {
            return cInserted == ' ' ? mBreakWeight : Double.NEGATIVE_INFINITY;
        }
        @Override
        public double matchWeight(char cMatched) {
            return mContinueWeight;
        }
        @Override
        public void compileTo(ObjectOutput objOut) throws IOException {
            objOut.writeObject(new Externalizable(this));
        }
        private static class Externalizable extends AbstractExternalizable {
            static final long serialVersionUID = -756373L;
            final SpacingTokenizing mDistance;
            public Externalizable() { this(null); }
            public Externalizable(SpacingTokenizing distance) {
                mDistance = distance;
            }
            @Override
            public void writeExternal(ObjectOutput objOut) throws IOException {
                objOut.writeDouble(mDistance.mBreakWeight);
                objOut.writeDouble(mDistance.mContinueWeight);
            }
            @Override
            public Object read(ObjectInput objIn)
                throws IOException, ClassNotFoundException {

                double breakWeight = objIn.readDouble();
                double continueWeight = objIn.readDouble();
                return new SpacingTokenizing(breakWeight,continueWeight);
            }
        }
    }

    /**
     * @brief Main entry point for the Coptic word segmenter
     *
     * @param args Command line arguments supplied by the user
     */
    public CopticTokens(String[] args)
    {
```

```
        mInputDirectory = args[0];
        mCorpusName = args[1];
        mOutputFile = new File(mInputDirectory + "/../models/" + mCorpusName + ".segments");
        mKnownToksFile = new File(mInputDirectory + "/../models/" + mCorpusName + ".knownWords");
        mMaxNGram = Integer.valueOf(args[2]);
        mLambdaFactor = Double.valueOf(args[3]);
        mNumChars = Integer.valueOf(args[4]);
        mMaxNBest = Integer.valueOf(args[5]);

        System.out.println("    Data Directory=" + mInputDirectory);
        System.out.println("    Corpus Name=" + mCorpusName);
        System.out.println("    Output File Name=" + mOutputFile);
        System.out.println("    Known Tokens File Name=" + mKnownToksFile);
        System.out.println("    Max N-gram=" + mMaxNGram);
        System.out.println("    Lambda factor=" + mLambdaFactor);
        System.out.println("    Num chars=" + mNumChars);
        System.out.println("    Max n-best=" + mMaxNBest);
    }

    /**
     * @brief Core portion of the application that calls the major methods needed
     *
     * @throws ClassNotFoundException
     * @throws IOException
     */
    void run() throws ClassNotFoundException, IOException
    {
        compileSpellChecker();

        testSpellChecker();

        printResults();
    }

    /**
     * @brief Training method for the spell checker used to implement word segmentation
     *
     * @throws IOException
     * @throws ClassNotFoundException
     */
    void compileSpellChecker() throws IOException, ClassNotFoundException
    {
        File inputDirectory = new File(mInputDirectory + "/Training");

        // First prepare the spell checking trainer using the space breaking tokenizer
        NGramProcessLM lm = new NGramProcessLM(mMaxNGram, mNumChars, mLambdaFactor);
        SpacingTokenizing distance = new SpacingTokenizing(0.0, 0.0);
        TokenizerFactory tokenFactory = new RegExTokenizerFactory(".+|\\s");

        TrainSpellChecker trainer = new TrainSpellChecker(lm, distance, tokenFactory);

        // Parse input files into the spelling checker
        for (String aFile : inputDirectory.list()) {

            if (new File(aFile).isDirectory()) {
                continue;
            }

            FileInputStream fileIn = new FileInputStream(inputDirectory + "/" + aFile);

            System.out.println("Reading Training Data from " + inputDirectory + "/" + aFile);

            InputStreamReader reader
                = new InputStreamReader(fileIn, Strings.UTF8);

            // Read files and produce token sets to assist with later analysis
            BufferedReader bufReader = new BufferedReader(reader);
            String [] lines = extractLines(bufReader, mTrainingCharSet,
                                           mTrainingTokenSet, mCharEncoding);

            // Pass the parsed lines into the spell checking trainer
            for ( String aLine : lines) {
                trainer.handle(aLine);
            }
            Streams.closeInputStream(fileIn);
        }

        System.out.println("  Found " + mTrainingCharSet.size() + " distinct characters.");
        System.out.println("  Found " + mTrainingTokenSet.size() + " distinct words or tokens.");
```

```
        // Now persist the compiled spell checker
        File modelFile = new File(mInputDirectory + "/../models/" + mCorpusName + "-
words.CompiledSpellChecker");
        System.out.println("Saving Spell Checker to " + modelFile);

        FileOutputStream fileOut = null;
        BufferedOutputStream bufOut = null;
        ObjectOutputStream objOut = null;

           try {
               fileOut = new FileOutputStream(modelFile);
               bufOut = new BufferedOutputStream(fileOut);
               objOut = new ObjectOutputStream(bufOut);
               trainer.compileTo(objOut);
           } finally {
               Streams.closeOutputStream(objOut);
               Streams.closeOutputStream(bufOut);
               Streams.closeOutputStream(fileOut);
           }
    }

    /**
     * @brief Method used to process unsegmented files into segmented output
     *
     * @throws IOException
     * @throws ClassNotFoundException
     */
    void testSpellChecker() throws IOException, ClassNotFoundException
    {
        File inputDirectory = new File(mInputDirectory + "/Testing");

        // Retrieve the compiled spell checker
        FileInputStream modelFile = new FileInputStream(mInputDirectory + "/../models/" + mCorpusName + "-
words.CompiledSpellChecker");
        ObjectInput fileInput = new ObjectInputStream(modelFile);

        // Build the spellCheck with options set to allow insertions and match's only
        CompiledSpellChecker spellChecker = (CompiledSpellChecker) fileInput.readObject();
        spellChecker.setAllowInsert(true);
        spellChecker.setAllowMatch(true);
        spellChecker.setAllowDelete(false);
        spellChecker.setAllowSubstitute(false);
        spellChecker.setAllowTranspose(false);
        spellChecker.setNumConsecutiveInsertionsAllowed(1);
        spellChecker.setNBest(mMaxNBest);

        // Prepare the output segmented file and set its encoding to the standard UTF-8
        System.out.println("Writing resulting tokens to " + mOutputFile);
        OutputStream out = new FileOutputStream(mOutputFile);
        mOutputWriter = new OutputStreamWriter(out, Strings.UTF8);

        for (String aFile : inputDirectory.list()) {

            if (new File(aFile).isDirectory()) {
                continue;
            }

            FileInputStream fileIn = new FileInputStream(inputDirectory + "/" + aFile);

            System.out.println("Reading Testing Data from " + inputDirectory + "/" + aFile);

            InputStreamReader reader
                = new InputStreamReader(fileIn, Strings.UTF8);

            // Prepare the input data set with populated token information
            BufferedReader bufReader = new BufferedReader(reader);
            String [] lines = extractLines(bufReader, mTestCharSet,
                                           mTestTokenSet, mCharEncoding);
               for ( String aLine : lines) {

                  // Segment each line
                String response = spellChecker.didYouMean(aLine) + ' ';

                // Write each segmented line to our output file
                mOutputWriter.write(response);
                mOutputWriter.write("\n");

                // For each line output reporting information
```

```
                Set<Integer> refSpaces = getSpaces(aLine);
                Set<Integer> responseSpaces = getSpaces(response);
                prEval("Break Points", refSpaces, responseSpaces, mBreakEval);

                Set<Tuple<Integer>> refChunks
                    = getChunks(aLine, mReferenceLengthHistogram);
                Set<Tuple<Integer>> responseChunks
                    = getChunks(response, mResponseLengthHistogram);
                prEval("Chunks", refChunks, responseChunks, mChunkEval);
            }
            reader.close();
        }
        mOutputWriter.close();
    }

    /**
     * @brief Output reporting information for the entire test
     *
     * @throws IOException
     */
    void printResults() throws IOException {
        StringBuilder sb = new StringBuilder();
        Iterator<String> it = mTrainingTokenSet.iterator();
        while (it.hasNext()) {
            sb.append(it.next());
            sb.append('\n');
        }
        Files.writeStringToFile(sb.toString(), mKnownToksFile,
                                Strings.UTF8);

        // Print basic token counts
        System.out.println("  Found " + mTestTokenSet.size() + " test tokens.");
        System.out.println("  Found "
                           + sizeOfDiff(mTestTokenSet, mTrainingTokenSet)
                           + " unknown test tokens.");
        System.out.print("    Found " + mTestCharSet.size() + " test characters.");
        System.out.println("  Found "
                           + sizeOfDiff(mTestCharSet, mTrainingCharSet)
                           + " unknown test characters.");

        // Print the length and break histograms
        System.out.println("\nReference/Response Token Length Histogram");
        System.out.println("Length, #REF, #RESP, Diff");
        for (int i = 1; i < 10; ++i) {
            Integer iObj = Integer.valueOf(i);
            int refCount = mReferenceLengthHistogram.getCount(iObj);
            int respCount = mResponseLengthHistogram.getCount(iObj);
            int diff = respCount-refCount;
            System.out.println("    " + i
                               + ", " + refCount
                               + ", " + respCount
                               + ", " + diff);
        }

        // Print precision, recall and F-Score information
        System.out.println("Scores");
        System.out.println("  EndPoint:"
                           + " P=" + mBreakEval.precision()
                           + " R=" + mBreakEval.recall()
                           + " F=" + mBreakEval.fMeasure());
        System.out.println("     Chunk:"
                           + " P=" + mChunkEval.precision()
                           + " R=" + mChunkEval.recall()
                           + " F=" + mChunkEval.fMeasure());
    }

    /**
     * @brief Main java entry point
     *
     * @param args Command line arguments from OS
     */
    public static void main(String[] args) {
        try {
            new CopticTokens(args).run();
        } catch (Throwable thrown) {
            thrown.printStackTrace(System.err);
        }
    }
```

```
}
```

OBSERVATIONS AND DISCUSSION

Running these tests is done using the same command line options as shown below:

data coptic 10 5.0 7000 1024

An ant build.xml is provided to allow these commands to be run from the CLI. Equally an Eclipse project can be generated from the ZIP file contents provided as part of this weeks' exercise.

The unit test of words previously seen results in the following output showing 100% correct segmentation.

To run this test we take one of the input training files and from it we copy a single line of text. This text is then placed into a new file in the data/Training directory. This has the line in both the training set and the test set. We then run the command and produce the following output.

```
    Data Directory=data
    Corpus Name=coptic
    Output File Name=data\..\models\coptic.segments
    Known Tokens File Name=data\..\models\coptic.knownWords
    Max N-gram=10
    Lambda factor=5.0
    Num chars=7000
    Max n-best=1024
Reading Training Data from data\Training/Father John.txt
Reading Training Data from data\Training/Father Shenoute.txt
Reading Training Data from data\Training/Keft Biography.txt
Reading Training Data from data\Training/Markarios of Antioch.txt
Reading Training Data from data\Training/The Meeting of Anthony and Athanasios.txt
  Found 79 distinct characters.
  Found 11019 distinct words or tokens.
Saving Spell Checker to data\..\models\coptic-words.CompiledSpellChecker
Writing resulting tokens to data\..\models\coptic.segments
Reading Testing Data from data\Testing/The Meeting of Anthony and Athanasios.txt
  Found 196 test tokens.
  Found 0 unknown test tokens.
    Found 31 test characters.  Found 0 unknown test characters.

Reference/Response Token Length Histogram
Length, #REF, #RESP, Diff
    1, 5, 5, 0
    2, 6, 6, 0
    3, 18, 18, 0
    4, 33, 33, 0
    5, 51, 51, 0
    6, 26, 26, 0
    7, 22, 22, 0
    8, 26, 26, 0
    9, 16, 16, 0
Scores
  EndPoint: P=1.0 R=1.0 F=1.0
     Chunk: P=1.0 R=1.0 F=1.0
```

As expected the above test produces prefect Precision and Recall.

We now run a test where some words are now unique and not seen previously. To do this we modify the training set and remove the line that was previously copied into our Testing directory. If you choose to remove the first sentence of the file "The Meeting of Anthony and Athanasios.txt" you will have 96 unique words that represent the following test log.

```
    Data Directory=data
    Corpus Name=coptic
    Output File Name=data\..\models\coptic.segments
    Known Tokens File Name=data\..\models\coptic.knownWords
    Max N-gram=10
```

```
    Lambda factor=5.0
    Num chars=7000
    Max n-best=1024
Reading Training Data from data\Training/Father John.txt
Reading Training Data from data\Training/Father Shenoute.txt
Reading Training Data from data\Training/Keft Biography.txt
Reading Training Data from data\Training/Markarios of Antioch.txt
Reading Training Data from data\Training/The Meeting of Anthony and Athanasios.txt
  Found 79 distinct characters.
  Found 10925 distinct words or tokens.
Saving Spell Checker to data\..\models\coptic-words.CompiledSpellChecker
Writing resulting tokens to data\..\models\coptic.segments
Reading Testing Data from data\Testing/The Meeting of Anthony and Athanasios.txt
  Found 196 test tokens.
  Found 94 unknown test tokens.
    Found 31 test characters.  Found 0 unknown test characters.

Reference/Response Token Length Histogram
Length, #REF, #RESP, Diff
    1, 5, 5, 0
    2, 6, 6, 0
    3, 18, 18, 0
    4, 33, 33, 0
    5, 51, 51, 0
    6, 26, 26, 0
    7, 22, 22, 0
    8, 26, 26, 0
    9, 16, 16, 0
Scores
  EndPoint: P=1.0 R=1.0 F=1.0
     Chunk: P=1.0 R=1.0 F=1.0
```

As can be seen test went well. While a good quantity of testing is needed the principles covered obviously shows excellent promise.

LESSONS LEARNED

While initially the word segmentation issue seemed like a difficult problem the LingPipe environment provided a useful solution in its Spell Checker Environment. The LingPipe SpellChecker provides excellent support for Edit Distance based algorithms for token processing.

The wide use of commonly accepted algorithms within LingPipe and GATE offers good support for those wishing to apply the common set of tools for uses of a more general nature than those that they are directly labeled for.

The scope and richness of the LingPipe environment is well thought out and offers good opportunities for use of a very general nature often beyond the core functions.

FUTURE WORK

While we have successfully shown that UTF-8 straight text can be segmented much of the text available from 3rd parties is encoded in annotated XML formats such as TEI used within the Perseus database employed by Tufts to store a large corpus of texts, c.f. http://www.perseus.tufts.edu/hopper/.

Using a standard such as TEI could be useful to create a trainer and text processor, using both GATE and LingPipe, that could not only segment texts but also perform Part Of Speech Tagging prior to adding texts to a searchable database.

In order to test the utility of doing and to evaluate the ability of the system we could process Greek texts from this Database. With use of Greek tests to perform testing the notion of POS tagging could be tested. Once understood and measured it should be possible for a skilled person to convert UTF-8 texts to TEI compliant texts for Coptic and for this experiment to be performed for Coptic.