Development of Natural Language Processing Library in Nemerle using Dotnet Framework

Animesh Pandey*, Siddharth Shrotriya**

* Department of Information Technology, Jaypee Institute of Information Technology, Noida, India
** Department of Electronic and Communication Engineering, Jaypee Institute of Information Technology, Noida, India

Abstract- In the generation of artificial intelligence and modern intelligent machines, Natural Language Processing forms a major part of the whole system. The implementation of NLP requires a meticulous and complex algorithm which consists of many functional, analytic semantic fragments. The realization of all the modules takes a lot of time and resources. Therefore, there is a need for libraries which contains these modules and can be used whenever required for implementing the algorithm. In this paper, we have constructed a NLP library in Nemerle programming language using the dotnet framework. We have used Nemerle due to its strong meta-programming system, C# like syntax, functional and imperative features. We have developed all the modules and then collaborated them in a single class library which can be used in other languages supported by dotnet framework. Using this library, one can build his own language processing algorithm buy using the provided fundamental features of NLP. In addition to that, we have also shown the advantage of using Nemerle over other available programming languages.
.

Index Terms- Class Library, Dotnet Framework, Natural Language Processing, Nemerle, Programming Languages and Visual Studio Templates

I. INTRODUCTION

Computer science has made huge strides since it became recognized as a distinct academic discipline in the 1960s. One of the fundamental problems it has addressed is how to translate the problems that people need solved into a format that computers can process. A variety of powerful and ingenious solutions have been applied to this problem. Programming languages are human-engineered languages developed to convey instructions to machines; they are based on rules of syntax and semantics. Thousands of different programming languages have been developed, used, and discarded. Initially, the evolution of programming languages was driven by a mission for efficient translation of human language to machine code. This has lead to greater use of objected-oriented and event driven programming languages along with functional and meta-programming languages.

We also know that making the computer understand how the humans process their own language is a challenging task. For this purpose we use an approach called Natural Language Processing (NLP). NLP is the computerized approach to analyzing text that is based on both a set of theories and a set of technologies. And, being a very active area of research and development, there is not a single agreed-upon definition that would satisfy everyone, but there are some aspects, which would be part of any knowledgeable person's definition.

Nemerle is a new language which is designed for dotnet framework was developed keeping in mind the pros and cons of various other languages like C#, ML and LISP. Nemerle was envisioned as a high-level programming language with support for object-oriented programming, functional programming, and meta-programming. It takes the OOP [1] part from C#, functional programming [7] part from ML (Meta Language) and meta-programming [2] from LISP. Hence, it turns out to be one of the most interesting languages offered. In our paper we have tried to develop a library class in Nemerle [3] having the wrapper functions for Natural Language processing algorithms in order to exemplify its programming paradigms. We have used Nemerle version 1.1.7for dotnet framework 4.0 in Visual Studio 2010 Ultimate Edition.

II. NATURAL LANGUAGE PROCESSING

Human mind is a very intelligent system. With good years of training, it is able to differentiate as well as interpret the meaning and structure of the language sentences. For making the computer do it, we use the theories of natural language processing that emulates the way the humans think about the languages. The choice of the word 'processing' is very deliberate, and should not be replaced with 'understanding'. For although the field of NLP was originally referred to as Natural Language Understanding (NLU) in the early days of AI, it is well agreed today that while the goal of NLP is true NLU, that goal has not yet been accomplished. A full NLU System would be able to:

1) Paraphrase an input text
2) Translate the text into another language
3) Answer questions about the contents of the text
4) Draw inferences from the text

While NLP has made serious inroads into accomplishing goals 1 to 3, the fact that NLP systems cannot, of themselves, draw inferences from text, NLU still remains the goal of NLP. [8]

The main work in computational linguistics began with covering the main parts of any language processing, which are:

These are:

1) Lexical Analysis

Lexicons are the basic semantic parts of any language. Their analysis includes basic things like tokenization, word count frequent word counts etc.

2) Automatic Tagging

Automatically disambiguating part-of-speech [9] labels in text is an important research area since such ambiguity is particularly prevalent in English. Programs resolving part-of-speech labels (often called automatic taggers) typically are around 95% accurate. Taggers can serve as preprocessors for syntactic parsers and contribute significantly to efficiency.

3) Parsing

```
public static Tokenize( text1 : string ) :array[string] {
def delimiters = array[
'{', '}', '(', ')', '[', ']', '>', '<','-', '_', '=', '+',
'|', '\\', ':', ';', ' ', '\'', ',', '.', '/', '?', '~', '!',
'@', '#', '$', '%', '^', '&', '*', ' ', '\r', '\n', '\t'
];

def st= text1.Split( delimiters,
StringSplitOptions.RemoveEmptyEntries );
return: {
st
}}
```

Figure1. Module to tokenize the sentence

The traditional approach to natural language processing takes as its basic assumption that a system must assign a complete constituent analysis to every sentence it encounters. The methods used to attempt this are drawn from mathematics, with context-free grammars playing a large role in assigning syntactic constituent structure.

4) Word-Sense Disambiguation

Automatic word-sense disambiguation [12] depends on the linguistic context encountered during processing. We find a variety of cues while parsing, including morphology, collocations, semantic context, and discourse. Statistical methods exploit the distributional characteristics of words in large texts and require training, which can come from several sources, including human intervention. We used the "iDictionary" interface for getting the trained word pairs. Token ratios and average token ratio gives an overview of several statistical techniques they have used for word-sense disambiguation.

5) Semantics

Semantics [13] deals with the exact meaning of any sentence. A sentence might be syntactically correct but semantically wrong. For e.g. there is a sentence "Jack plays guitar", syntactically there must be two words around a verb. So, the sentence "Guitar plays Jack" is correct syntactically, but semantically wrong. So, determining which word is subject, which is object and which is the predicate is an example of semantic analysis.

```
public static getTokenRatio (textStr : string) : double
{

def tokens = Tokenize( textStr );
def counttoken = tokens.Length;
def ct = tokens.Distinct().Count();
def ratio = ct/counttoken :> double;

    return:{
        ratio
    }
}
```

Figure2. Module for calculating Type-Token Ration

6) Discourse Analysis

Discourse analysis is concerned with coherent processing of text segments larger than the sentence and assumes that this requires something more than just the interpretation of the individual sentences. This basically deals with how the syntax and semantics of the sentences support the structure of the discourse analysis.

.

```
def printReg (input){ regexp match (input) {
    | @"((?<=[a-z])(?=[A-Z]))|((?<=[A-Z])(?=[A-Z][a-z]))"
    => { def newStr = Regex.Replace(input,
     @"((?<=[a-z])(?=[A-Z]))|((?<=[A-Z])(?=[A-Z][a-z]))"
     , " " );
     Console.WriteLine( newStr ); }
    | _ => {
    WriteLine("Invalid Exp!");
    } } }
    printReg(input); }
```

Figure3. Module to split the sentence

III. Making of NLP Library in Nemerle

Nemerle is a general-purpose high-level statically typed programming language [4] designed for platforms using the Common Language Infrastructure (.NET/Mono). It offers functional, object-oriented and imperative features. It has a simple C#-like syntax and a powerful meta programming system. One of the best things that made us work with Nemerle language was its functional programming and meta programming. This made us try, to make an NLP class library [14] with an easy code, better readability and better performance. We have incorporated all the points of NLP as mentioned above.

Nemerle's most notable feature is the ability to mix object oriented and functional styles of programming. Programs may be structured using object oriented concepts such as classes and namespaces, while methods can (optionally) be written in a functional style. Other notable features include:

1) Strong type inference
2) A flexible meta programming subsystem (using macros)
3) Full support for OOP (in the style of C#, Java, and C++)
4) Full support for functional programming (in the style of ML, OCaml, Haskell) with the following features:
   a. higher-order functions
   b. pattern matching
   c. algebraic types
   d. local functions
   e. tuples and anonymous types
   f. partial application of functions

The meta-programming system allows for a great deal of compiler extensibility, embedding of domain specific languages, partial evaluation, and aspect-oriented programming, taking a high-level approach to lift as much of

the burden from the programmer as possible. The language combines all CLI standard features, including parameter polymorphism, lambdas, extension methods etc. Accessing the libraries included in the .NET or Mono platforms is as easy as in C#.

```
public static SplitWord
( word : string ) : void {
when ( word.Length < 2 ) {
WriteLine("The word is too short to split.");
}
mutable index = 0;
def arrSize = ( word.Length - 1 ) * 2;
def strArr = array (arrSize);
Console.WriteLine( );

for ( mutable cutPosition = 1; cutPosition < word.Length;
 cutPosition++ ) {
strArr [index]
= word.Substring( 0, cutPosition );

strArr [index + 1]
= word.Substring( cutPosition,word.Length-cutPosition);
Console.WriteLine ( string.Format( "{0} {1}" ,
strArr [index], strArr [index + 1] ) );
index += 2;
}}
```

Figure4. Module for checking regular expression

```
def exprMatch (str) {
regexp match (str) {
    | @"^[a-z]+$" => {
    Console.WriteLine( str +" lower case string!" );}
    | @"^[0-9][0-9][0-9][0-9]$" => {
    Console.WriteLine( str +" four digit pattern!" );}
    | @"[a-zA-Z]" => {
    Console.WriteLine( str +" a letter pattern!" );}
    | @"^[A-Z]\\.$" => {
    Console.WriteLine( str +" Cap period pattern!" );}
    | _ => {
    Console.WriteLine("No regular expression found!");
        }}}
exprMatch(str);

public virtual GetContext
(context : object) :array[string] {
object[] contextData =
(object[]) context;
return (
GetContext((contextData[0] :>int),
    contextData[1] :>list[string,
    contextData[2] :>list[string],
    contextData[3] :>IDictionary[string, string])
);}
```

Figure5. Module for context generator

A. Reason for choosing Nemerle.

In this approach of developing a library for NLP, we require a programming language that can work simultaneously in processing many modules [15] and also satisfy the criteria of space and time complexity. We choose the dotnet framework because it can support many languages and provide easy to use template functionality along with the options of developing many kinds of applications. We analyzed many languages of dotnet framework and found that among all, Nemerle fulfill our necessities for this algorithm. Also, for other languages, the library for NLP is already built and shared, so we have tried to make such a library for Nemerle programming language.

```
def n = Diagnostics.DebuggerStepper
        BoundaryAttribute();
def cacheKey = $"index.ToString(System.
 Globalization.CultureInfo.InvariantCulture)
      $tagPrevious $tagPreviousPrevious";

when (!(mContextsCache == null))
{
if (mWordsKey == tokens && n)
{
def cachedContexts = mContextsCache[cacheKey] ;
     when (cachedContexts != null)
     {  return cachedContexts; } }
    else {
        mContextsCache.Remove(0);
        mWordsKey = tokens; }

ExecuteReaderLoop("SELECT pos_tag, chunk_de FROM
  dictionary WHERE mWordsKey = $token[cacheKey]",
  dbcon,{ WriteLine ($"Tree: $pos_tag $chunk_de")
  });     }
```

Figure6. Module for Tagger, Chunking and Parse Tree

B. Software used

The most widely used software which can mimic the process involved in our development and can produce the possible result is Visual Studio. Microsoft Visual Studio [5] is an integrated development environment (IDE) from Microsoft. It is used to develop console and graphical user interface applications along with Windows Forms applications, web sites, web applications, and web services in both native code together with managed code for all platforms supported by Microsoft Windows, Windows Mobile, Windows CE, .NET Framework, .NET Compact Framework and Microsoft Silverlight.

Visual Studio supports different programming languages by means of language services, which allow the code editor and debugger to support (to varying degrees) nearly any programming language, provided a language-specific service exists. Built-in languages include C/C++ (via Visual C++), VB.NET (via Visual Basic .NET), C# (via Visual C#), and F# (as of Visual Studio 2010). Support for other languages such as M, Nemerle, Python, and Ruby among others is available

via language services installed separately. It also supports XML/XSLT, HTML/XHTML, JavaScript and CSS.

In Visual studio, we have used the class library project template for the development of our code for the required algorithm. Class libraries are the rough OOP equivalent of older types of code libraries. They contain classes, which describe characteristics and define actions (methods) that involve objects. Class libraries are used to create instances, or objects with their characteristics set to specific values.

IV. PROGRAMMING THE COMPONENTS OF NLP LIBRARY

Here comes the most crucial step of our development. We have implemented a function that tokenizes (Figure 1.) the paragraphs into separate words on the basis of some delimiters like punctuations in English language.

After tokenization, we proceeded to the word count and token ratio of a text input. After calculating the token ratio, we tried to split the word (Figure 3.) into its possible phrases and

CC	Coordinating conjunction	RP	Particle
CD	Cardinal number	SYM	Symbol
DT	Determiner	TO	to
EX	Existential there	UH	Interjection
FW	Foreign word	VB	Verb, base form
IN	Preposition/subordinate conjunction	VBD	Verb, past tense
JJ	Adjective	VBG	Verb, gerund/present participle
JJR	Adjective, comparative	VBN	Verb, past participle
JJS	Adjective, superlative	VBP	Verb, non-3rd ps. sing.present
LS	List item marker	VBZ	Verb, 3rd ps. sing. present
MD	Modal	WDT	wh-determiner
NN	Noun, singular or mass	WP	wh-pronoun
NNP	Proper noun, singular	WP$	Possessive wh-pronoun
NNPS	Proper noun, plural	WRB	wh-adverb
NNS	Noun, plural	``	Left open double quote
PDT	Predeterminer	,	Comma
POS	Possessive ending	''	Right close double quote
PRP	Personal pronoun	.	Sentence-final punctuation
PRP$	Possessive pronoun	:	Colon, semi-colon
RB	Adverb	$	Dollar sign
RBR	Adverb, comparative	#	Pound sign
RBS	Adverb, superlative	-LRB-	Left parenthesis *
		-RRB-	Right parenthesis *

Figure7. English look up list for PosTagger

then check the correct break down of the text keeping in mind the semantics of English Grammar. Tokens are referred to all the words present in the data and type is the number of uniquely identifies set of words.

After calculating the token ratio for the input text, we tried to split the word (Figure 3.) into its possible phrases and then the check the correct break down of the text or word we have. This helps us in getting the possible meaning of the complex string.

Then it was required to test the application of regular expression (Figure 4.) in Nemerle, as regular expressions have a great importance in lexical analysis, discourse analysis and syntactical analysis. For this we used the unique way of pattern matching that has multi use in Nemerle. This type of syntax addresses functional programming and uses '|' for denoting different conditions. [10]

After this, the words are matched with the list of symbols mentioned in the look up library (Figure 7.). These symbols are connected with each word to form a list of matched words known as Pos tagger. This tagger performs the task of segregating the text into the grammatical parts of a sentence. Then the individual word is classified in the parts of speech like verbs, nouns, pronouns etc. (Figure 6.) This process is known as chunking [11] and from this a tree is formed with the help of the list of words known as Parse tree. [6]

Next we have tried integrating the regular expression method in the context generator module (Figure 5.). This module tries to define the type of the semantics, parts of speech and meaning of the text by using a trained dictionary in dotnet which is popularly known as iDictionary Interface and it is provided in the templates for Visual Studio.

Finally, we have formed a strong base to perform the NLP algorithm with the use of the above mentioned modules. These are modules are the incorporated into a class library so that these can be used with ease in the code of any language processing algorithm. For this library, we copied all the codes and pasted in the class library project of Visual studio and made some necessary changes to combine all the modules successfully. This project is then compiled in the Release mode and the dll and xml files of all the modules of the library were generated. The generated XML file of the whole Nemerle class library project is demonstrated in two parts in Figure 8 and Figure 9 and combined with necessary comments to show the working of important lines of the XML file.

V. CONCLUSION

At the end of this paper, we can state that we have successfully developed the class library for natural language processing algorithm using Nemerle which is a better programming language for this purpose considering the above mentioned advantages over other available languages. We have developed this library to facilitate the availability of NLP functions and modules to the researchers involved in this research field and the programmers coding in Nemerle or any other language supported by dotnet framework. After this, we look forward to develop more modules and libraries for other complex algorithms using the then most effective and beneficial programming language.

Figure8. First Part of the generated XML file of the developed Class library in Nemerle

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
  <PropertyGroup>
    <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
    <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
    <ProductVersion>8.0.30703</ProductVersion>
    <SchemaVersion>2.0</SchemaVersion>
    <ProjectGuid>{de459828-7b0b-44db-9264-fcbcf71c8316}</ProjectGuid>
    <OutputType>Library</OutputType>
    <AppDesignerFolder>Properties</AppDesignerFolder>
    <RootNamespace>nemerlenlp</RootNamespace>
    <AssemblyName>nemerlenlp</AssemblyName>
    <TargetFrameworkVersion>v4.0</TargetFrameworkVersion>
    <FileAlignment>512</FileAlignment>
    <NoStdLib>true</NoStdLib>
    <NemerleVersion>Net-4.0</NemerleVersion>
    <NemerleBinPathRoot Condition=" '$(NemerleBinPathRoot)' == '' ">$(ProgramFiles)\Nemerle</NemerleBinPathRoot>
    <Nemerle Condition=" '$(Nemerle)' == '' ">$(NemerleBinPathRoot)\$(NemerleVersion)</Nemerle>
    <Name>nemerlenlp</Name>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
    <DebugSymbols>true</DebugSymbols>
    <Optimize>false</Optimize>
    <OutputPath>bin\Debug\</OutputPath>
    <DefineConstants>DEBUG;TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">
    <DebugSymbols>false</DebugSymbols>
    <Optimize>true</Optimize>
    <OutputPath>bin\Release\</OutputPath>
    <DefineConstants>TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">
    <DebugSymbols>false</DebugSymbols>
    <Optimize>true</Optimize>
    <OutputPath>bin\Release\</OutputPath>
    <DefineConstants>TRACE</DefineConstants>
    <ErrorReport>prompt</ErrorReport>
    <WarningLevel>4</WarningLevel>
    <DocumentationFile>$(OutputPath)\$(MSBuildProjectName).xml</DocumentationFile>
  </PropertyGroup>
  <ItemGroup>
    <Reference Include="mscorlib" />
    <Reference Include="System" />
    <Reference Include="System.Core">
      <RequiredTargetFramework>3.5</RequiredTargetFramework>
    </Reference>
    <Reference Include="System.Xml.Linq">
      <RequiredTargetFramework>3.5</RequiredTargetFramework>
    </Reference>
    <Reference Include="System.Data.DataSetExtensions">
      <RequiredTargetFramework>3.5</RequiredTargetFramework>
    </Reference>
```

```
    <Reference Include="System.Data" />
    <Reference Include="System.Xml" />
    <Reference Include="Nemerle">
      <SpecificVersion>False</SpecificVersion>
      <HintPath>$(Nemerle)\Nemerle.dll</HintPath>
      <Private>True</Private>
    </Reference>
    <MacroReference Include="Nemerle.Linq">
      <HintPath>$(Nemerle)\Nemerle.Linq.dll</HintPath>
    </MacroReference>
  </ItemGroup>
  <ItemGroup>
    <Compile Include="lexical.n" /><!-- Module for texical constructs -->
    <Compile Include="lexical\token.n" /><!-- Module for Tokenizer -->
    <Compile Include="lexical\wdcount.n" /><!-- Module for Word Count -->
    <Compile Include="lexical\wdfreq.n" /><!-- Module for Word Frequency -->
    <Compile Include="autotag.n" /><!-- Module for Tagging Constructs -->
    <Compile Include="autotag\postag" /><!-- Module for creating Pos Tags -->
    <Compile Include="parsing.n" /><!-- Module for Parsing constructs -->
    <Compile Include="parsing\regexp.n" /><!-- Module for Regular Expressions -->
    <Compile Include="worddisamb.n" /><!-- Module for Ambigous constructs -->
    <Compile Include="worddisamb\contextgen" /><!-- Module for Context Generation -->
    <Compile Include="semantics.n" /><!-- Module for Semantic constructs -->
    <Compile Include="semantics\chunk" /><!-- Module for Chunking and Parse tree -->
    <Compile Include="discourse.n" /><!-- Module for Discourse Analysis -->
    <Compile Include="discourse\wdsplit" /><!-- Module for Spliting algorithm -->
    <Compile Include="Properties\AssemblyInfo.n" />
  </ItemGroup>
  <ItemGroup>
    <Folder Include="Properties\" />
  </ItemGroup>
  <Import Project="$(Nemerle)\Nemerle.MSBuild.targets" />
</Project>
```

Figure9. Second Part of the generated XML file of the developed Class library in Nemerle

REFERENCES

[1] Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press. ISBN 0-262-16209-1., section 18.1 "What is Object-Oriented Programming?"

[2] Chlipala, Adam (June 2010). "Ur: statically-typed metaprogramming with type-level record computation". ACM SIGPLAN Notices. PLDI '10 45 (6): 122–133. doi:10.1145/1809028.1806612

[3] http://en.wikipedia.org/wiki/Nemerle

[4] Pratt, Terrence, W. and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. 3rd ed. Englewood Cliffs, New Jersey: Prentice Hall, 1996.

[5] http://en.wikipedia.org/wiki/Microsoft_Visual_Studio

[6] Kovar, V., Horak, A., Kadlec, V.: New Methods for Pruning and Ordering of Syntax Parsing Trees. In Proceedings of Text, Speech and Dialogue 2008.In:Lecture Notes in Articial Intelligence, Proceedings of Text, Speech and Dialogue 2008, Brno, Czech Republic, Springer-Verlag (2008) 125-131

[7] Turner, D.A. (2004-07-28). "*Total Functional Programming*". Journal of Universal Computer Science 10 (7): 751–768. doi:10.3217/jucs-010-07-0751

[8] http://cyborganthropology.com/Natural_Language_Processing

[9] Kate, R., Wong, Y., Ge, R., AND Mooney, R. Learning to transform natural to formal languages. In Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05) (Pittsburgh, 2005)W.-K. Chen, *Linear Networks and Systems* (Book style).Belmont, CA: Wadsworth, 1993, pp. 123–135.

[10] nemerle.org Community: nemerle.org. http://www.nemerle.org (2012)

[11] G. Zhou and J. Su. Named entity recognition using an hmm-based chunk tagger. In Proceedings of ACL'02, pages 473–480, 2002.

[12] I. H. Witten, G. W. Paynter, E. Frank, C. Gutwin, and C. G. NevillManning. KEA: Practical automatic keyphrase extraction. In DL'99: Proceedings of the 4th ACM International Conference on Digital Libraries, pages 254–255, 1999.

[13] Liu, H., and Lieberman, H. Programmatic semantics for natural language interfaces. In Proceedings of the ACM Conference on Human Factors in Computing Systems (CHI-2005) (Portland, OR, 2005)

[14] http://en.wikipedia.org/wiki/Class_library

[15] http://freecode.com/projects/nemerle

AUTHORS

First Author – Animesh Pandey, B-Tech(IT) – Final year, Jaypee Institute of Information Technology, Noida, India, Email: animeshpandey@acm.org

Second Author – Siddharth Shrotriya, B-Tech(ECE) - Final year, Jaypee Institute of Information Technology, Noida, India, Email: siddharth@ieee.org

Correspondence Author – Siddharth Shrotriya
Email: siddharth@ieee.org
Contact no: +91 8800723