Universidad de Ingeniería y Tecnología

Ciencia de la Computación

Natural Language Processing with Context Free Grammar

Proyecto - Teoría de la Computación

Autores:
Christian Ledgard Ferrero
Anthony Aguilar Sanchez

Junio 2020

Índice

1. Introducción **2**

2. Alcance **2**

3. Definición del Problema **2**

4. Estado del Arte **2**

5. Propuestas **3**

6. Implementación de Algoritmos **4**
6.1. Clase Gramática 4
6.2. Algoritmo CYK 6
6.3. Algoritmo Earley Parser 8

7. Hoy en la Industria: Análisis Experimental **10**
7.1. Decision List 12

8. Alternativas: Inteligencia Artificial **15**
8.1. Implementación Simple de AI 15
8.2. Futuras implementaciones y cambios para utilizar AI en la detección de ambigüedad 18

9. GitHub **19**

1. Introducción

Hoy en día la tecnología está presente en muchos ámbitos de nuestra sociedad. Desde vehículos que se manejan por sí solos hasta en nuestros celulares que nos permiten estar más comunicados que nunca. Unos de los avances más significativos en los últimos tiempos son los asistentes virtuales como Siri, Alexa, Google Assistant, entre otros. ¿Alguna vez se han preguntado cómo estos sistemas organizan y analizan nuestras oraciones para ponerlas en un determinado contexto? Una posible implementación es utilizar **gramáticas independientes del contexto** para organizar y luego generar un árbol de *"parseo"* con la oración. Esto se logra al tener un léxico y una gramática previamente definida.

2. Alcance

En nuestro proyecto nosotros realizaremos una investigación aplicada en donde desarrollaremos un algoritmo simple para realizar un parser de acuerdo a un léxico y gramática previamente definida. Utilizaremos diversas fuentes (1), (2), (3), (4), (5), (6), (7), (8) y (9).

3. Definición del Problema

Cómo dividir una oración, utilizando gramáticas independientes del contexto, para que el computador pueda interpretarlas correctamente. ¿Casos de ambigüedad? ¿Qué hacer en determinados casos?

4. Estado del Arte

El inicio del procesamiento de lenguaje natural suele remontarse a 1950 cuando Alan Turing publicó el articulo 'Computing Machinery and Intelligence' donde propone lo que conocemos como el test de Turing, un criterio de inteligencia.
Una de las primeras aplicaciónes del NLP ocurrió en 1954 e involucro la traducción de 60 oraciones rusas al inglés.
En los 1960´s ELIZA y SHRDLU fueron sistemas capaces de comunicarce con humanos.
Existen al menos 4 algoritmos de parsing:

1. El algoritmo CYK (Cocke–Younger–Kasami)

1.1. Solo trabaja con CFG en la forma normal de chumsky

1.2. El peor de los casos es $O(n^3.|G|)$ time. Donde n es el tamaño de la cadena y $|G|$ es el tamaño de la gramática G lo que lo hace uno de los mas eficientes en el peor de los casos.

2. Earley parser

2.1. Trabaja con todas las gramáticas libres del contexto.

2.2. Tiene un tiempo de ejecución $O(n^3)$ en el caso promedio,$O(n^2)$ en gramáticas no ambiguas y $O(n)$ para CFG deterministas.

3. LR parser (Left-to-right, Rightmost derivation in reverse)
Son tipos de bottom-up parsers que analizan DCFG en tiempo lineal.

3.1. LALR parser

3.2. Canonical LR parser

3.3. Minimal LR parser

3.4. GLR parser

4. LL parser (Left-to-right, Leftmost derivation)
Es un top-down parser para un subset de CFG

5. Propuestas

- Implementar y analizar dos algoritmos de parseo para construir una gramática independiente del contexto bajo una gramática y un lenguaje dado.
- Analizar posibles implementaciones y soluciones en casos de ambigüedad y presentar propuestas utilizadas actualmente por grandes empresas como Amazon o Google.

6. Implementación de Algoritmos

Para implementar los algoritmos CYK y Earley definimos primero que es una gramática.

6.1. Clase Gramática

Para la implementación de la estructura regla, la dividimos en 2 partes: Derecha e Izquierda. Ejemplo: izq ->der. La parte izquierda estará definida por un string y la derecha por un vector de strings. Asimismo, creamos la clase gramática que almacenará en un vector de punteros a reglas, todas las reglas a analizar.

```
#ifndef C___GRAMATICA_H
#define C___GRAMATICA_H

#include <string>
#include <utility>
#include <vector>
#include <iostream>


using namespace std;


struct regla{
    string izq;
    vector<string> der;
    regla(string izq, vector<string> der) : izq(std::move(izq)), der(
   std::move(der)) {}
    string preattyStringOutput();
};

class gramatica{
    vector<regla*> gram;

public:
    gramatica() = default;
    void crearRegla(string izq, const string& der);
  regla* inicio();
  regla* get_regla(char);

    string queReglaDeriva(const string& x);

    void print();

```

```
33      virtual ~gramatica();
34  };
35
36  #endif //C___GRAMATICA_H
```

6.2. Algoritmo CYK

El siguiente algoritmo, considerado 'bottom-up parsing', fue inventado por John Cocke, Daniel Younger y Tadao Kasami. Este solo opera con gramáticas en la forma normal de Chomsky (6).

Utilizando la notación Big O, el peor caso del algoritmo CYK es $\mathcal{O}\left(n^3 \cdot |G|\right)$ donde n es la longitud de la cadena y $|G|$ es el tamaño de la gramática (10). Esta complejidad hace que sea uno de los algoritmos más eficientes de parseo en términos del peor caso.

Cumpliendo con los estándares, utilizaremos programación dinámica para resolver el Algoritmo CYK. Conforme al pseudocódigo expuesto en la Figura 1, realizaremos la implementación en C++, orientada a objetos.

```
CYK(G, S, w = a_1 a_2 ... a_n)
 1: if w = ϵ AND existe regla S → ϵ
 2:     return VERDADERO
 3: for i = 1 to n
 4:     for cada regla A → a_i
 5:         M[i, i] = M[i, i] ∪ {A}
 6: for ℓ = 2 to n
 7:     for i = 1 to n − ℓ + 1
 8:         j = i + ℓ − 1
 9:         for k = i to j − 1
10:             if existe regla A → BC, B ∈ M[i, k],
                C ∈ M[k + 1, j]
11:                 M[i, j] = M[i, j] ∪ {A}
12: return (S ∈ M[1, n])
```

Figura 1: Pseudocódigo del algoritmo CYK

Nosotros creamos un array llamado matriz para almacenar el resultado ya procesado. Asimismo, en el método 'solve', ejecutaremos el algoritmo. Dicho método retornará un doble puntero, al puntero del array donde estará guardado nuestro resultado dinámico con la solución. También, en el método 'cadenaAceptada()', retornamos TRUE si la cadena es aceptada por el algoritmo, de lo contrario, retornaremos FALSE.

```
1 #ifndef C___CYK_H
2 #define C___CYK_H
```

```
3
4  #include <string>
5  #include <utility>
6  #include <vector>
7  #include <iostream>
8  #include <iomanip>
9  #include <unordered_map>
10
11 #include "gramatica.h"
12
13 using namespace std;
14
15
16 class CYK{
17 private:
18     gramatica* G;
19     string cadena;
20     char S;
21
22     string **matriz;
23     size_t arraySize;
24
25     static string getString(char x){ string s(1, x); return s; }
26     static vector<string> distributiva(const string& a, const string& b
       );
27     string eliminarDuplicados(string x);
28
29 public:
30     CYK(gramatica *g,char S, const string& cadena);
31
32     string **solve();
33     bool cadenaAceptada();
34     friend ostream & operator<< (ostream &out, const CYK &cyk);
35
36
37     virtual ~CYK();
38
39 };
40
41 #endif //C___CYK_H
```

6.3. Algoritmo Earley Parser

A diferencia del algoritmo CYK el algoritmo de Earley es del tipo 'top-down' ya que éste evalúa si la cadena pertenece a la gramática empezando por la regla inicial y derivándola hasta obtener la cadena en el caso que ésta exista en la gramática.

Para los siguientes párrafos α, β y γ representan cualquier cadena de variables y símbolos de la gramática. X y Y representan variables de la gramática.Y a representa cualquier símbolo de la gramática.

Para lograr esta derivación se introduce una notación llamada 'dot-notation' o notación-punto $X \to \alpha \cdot \beta$. Esta notación indica que parte de una regla de la gramática ya ha sido reconocida (la parte a la izquiera del punto) y qué variable o símbolo se espera a continuación (la parte a la derecha del punto).

Dada una cadena de tamaño n se construye un array de tamaño $n + 1$ donde cada posición en el array a partir de la posición 1 corresponde a un símbolo en la cadena correspondiendo la posición 0 al ϵ que precede a la cadena. Cada lugar en el array le corresponde a un conjunto de estados donde cada estado es una tupla que se construye a partir de una 'dot-notation' y la posición en el array donde ésta se originó $(X \to \alpha \cdot \beta, i)$. Llamamos al conjunto de estados que se encuentran en la posición k $E(k)$.

Este algoritmo consiste en repetir tres operaciones basadas en estos estados: predecir, escanear y completar.

- Predecir: para todos los estados en $E(k)$ de la forma $(X \to \alpha \cdot Y\beta, i)$, agregamos $(Y \to \cdot\gamma, k)$ a $E(k)$ para todas las producciones de la forma $Y \to \gamma$ en la gramática.
- Escanear: si a es el siguiente símbolo a leer entonces para todos los estados en $E(k)$ de la forma $(X \to \alpha \cdot a\beta, i)$, agregamos $(X \to \alpha a \cdot \beta, i)$ a $E(k + 1)$.
- Completar: para todos los estados en $E(k)$ de la forma $(Y \to \gamma\cdot, i)$, buscamos todos los estados en $E(i)$ de la forma $(X \to \alpha \cdot Y\beta, j)$ y agregamos $(X \to \alpha Y \cdot \beta, j)$ a $E(k)$.

Notese que el tamaño de $E(k)$ puede aumentar de tamaño después de cada operación así que se tiene que realizar las respectivas operaciones a los nuevos estados generados por operaciones previas.

```
#ifndef __Ealey_Parcer__
#define __Ealey_Parcer__

#include "gramatica.h"

struct Estado{
  size_t punPos;//marca que parte de la regla ya a sido parceada
  size_t origen;//en que etapa del algoritmo fue creado

  string izq;
  string der;

  char siguiente(){return der[punPos];}
  bool esta_completo(){return punPos==der.size();}

  void mostrar();

  Estado(regla* _regla, size_t cual,size_t ppos,size_t ori);

  Estado(Estado* otro_estado,size_t pos);

  bool es_igual_a(Estado* otro_estado);
};

class Earley{
  vector<vector<Estado*> > tabla;
  gramatica* _gramatica_;
  string _cadena_;

  void agregar_estado(regla*,size_t,size_t,size_t);//a partir de una
   regla
  void agregar_estado(Estado*,size_t,size_t);//a partir de otro estado;
   avanza el punto

  void predecir(Estado*,size_t);
  void escanear(Estado*,size_t);
  void completar(Estado*,size_t);

public:
  Earley(gramatica*,string);
  bool reconocer();
    virtual ~Earley();
};

bool es_mayuscula(char letra){return (letra>='A' && letra<='Z');}

```

```
45 #endif
```

7. Hoy en la Industria: Análisis Experimental

Muchas de las grandes empresas que conocemos el día de hoy, como Google, Apple, Microsoft, entre otras, desarrollan su propia tecnología para interpretar un texto simple y analizarlo con el objetivo que el computador pueda interpretar el contexto de la oración. Por un lado podemos observar a empresas como Apple INC. que desarrollan un framework llamado 'NSLinguisticTagger' (11), el gigante de Google desarolla 'Syntaxnet' (12), y muchas empresas suman y aportan más tecnologías. En este caso, analizaremos SyntaxNet puesto que en el 2016 la declararon Open Source. Esto significa que nosotros, los usuarios, podemos utilizar, cambiar y redistribuir el software, a cualquiera, para cualquier propósito, ya sea en su forma modificada o en su forma original (13).

Un gran problema al desarrollar este tipo de tecnologías es que el lenguaje que utilizamos nosotros, los seres humanos, tiene grandes niveles de ambigüedad (propiedad muy conocida al analizar las Gramáticas Independientes del Contexto). Google AI (14) nos menciona que es muy común encontrarse con oraciones de 20 a 30 palabras que cuenten con miles o cientos de miles posibilidades sintácticas. ¿Esto qué quiere decir? Que el computador tiene que buscar entre esas alternativas y encontrar la estructura más plausible dado el contexto.

En este caso vamos a mencionar un ejemplo expuesto por Google AI (14).

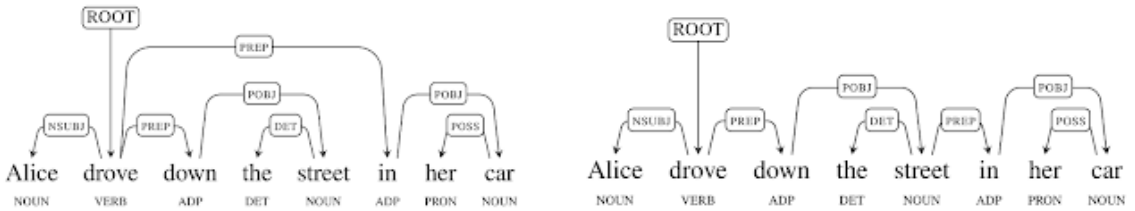

Figura 2: Ejemplo de Ambiguedad de Google AI (14)

En la siguiente oración: 'Alice drove down the street in her car.' (Alice condujo por la calle en su auto.) podemos obtener 2 interpretaciones: La correcta, señala que Alice está efectivamente manejando su auto, y la segunda nos indica que la calle está ubicada en el carro (posible sintácticamente, pero absurdo). Este es un claro ejemplo

de ambigüedad donde el computador tiene que interpretar de manera correcta el contexto de la oración.

Google AI (14) también nos menciona que nosotros, como seres humanos, somos expertos en resolver estos problemas de ambigüedad. Somos tan expertos que ni siquiera notamos que estamos procesando esto en nuestro cerebro. El verdadero desafío está en que el computador realice las interpretaciones de manera correcta.

Nosotros sabemos que el problema de la ambigüedad no tiene una solución exacta. Por ello, todos los métodos para resolver la ambigüedad serán métodos de aprendizaje estadísticos, en el sentido de que intentan hacer una generalización inductiva a partir de los datos observados y utilizarlos para hacer inferencias con respecto a datos nunca vistos anteriormente (15). Estos métodos estadísticos, hoy en día, están considerados dentro de la rama de la Inteligencia Artificial. Roth (15) nos menciona que existen cuatro aproximaciones para resolver el problema de la desambigüedad: The naive Bayes estimation (Duda Hart 1973), Katz's back-off model, transformation based learning (Brill 1995) y el Decision Lists (Yarowsky 1995).

Todas estas aproximaciones buscan una superficie de decisión que sea una función lineal en el espacio de características. Esto quiere decir que los métodos asumen que el espacio está dividido en una función lineal, con la propiedad de que en una de las regiones definidas, la predicción más probable es 0 y en la otra, la predicción más probable es 1. (15)

Empresas como Google no son ajenas a estas técnicas de AI expuestas. Ellos utilizan estos métodos para resolver los problemas de ambigüedad. En este caso, 'SyntaxNet' utiliza redes neuronales para resolver el problema de la ambigüedad. Pero, ¿cómo lo hacen?. Google AI (14) nos señala que procesan una oración de izquierda a derecha en donde cada palabra de la oración es considerada en el algoritmo. Cabe recalcar que llegan a un punto donde encuentran varias soluciones posibles, pero debido a la ambigüedad, no pueden llegar a una decisión prematura. Por esta razón, utilizan un 'Beam search' (optimización del best-first search) para recorrer el grafo generado e ir descartando posibles hipótesis cuando existan varias otras hipótesis 'rankeadas' con un puntaje más alto y estas estén bajo consideración.

¿Que tan preciso es este modelo? Según Google AI (14) se ha logrado una precisión al 94 % después de probar el algoritmo con miles de oraciones en el idioma inglés. Asimismo, en un estudio con lingüistas se comprobó que los seres humanos tienen un 96 %-97 % de precisión. Lograr esta precisión es algo extraordinario, todavía no llegamos a la precisión humana pero estamos muy cerca. Con estos resultados esta tecnología puede aplicarse sin problema en diversas áreas.

Este nuevo modelo se encuentra en el siguiente 'paper' (16) publicado por Google el 19 de marzo del 2016.

7.1. Decision List

Listas de decisión es un algoritmo usado para resolver el problema de ambigüedad en el análisis del lenguaje natural con un porcentaje de acierto de incluso más del 90 % en las ambigüedades más complicadas. El algoritmo consiste en 7 pasos:(17)

1. Identificar las ambigüedades que se quieren resolver:
   Por ejemplo los significados de palabras francesas que llevan acento. (17)

De-accented Form	Accent Pattern
cesse	cesse cessé
cout	coût
couta	coûta
coute	coûté coûte
cote	côté côte cote coté
cotiere	côtière

2. Recolectar los contextos para entrenar al algoritmo:
   Para esto es necesario construir oraciones que contengan las ambigüedades del punto anterior y que muestren diferentes contextos de uso. (17)

Pattern	Context
(1) côté	du laisser de *cote* faute de temps
(1) côté	appeler l' autre *cote* de l' atlantique
(1) côté	passe de notre *cote* de la frontiere
(2) côte	vivre sur notre *cote* ouest toujours verte
(2) côte	creer sur la *cote* du labrador des
(2) côte	travaillaient cote a *cote* , ils avaient

3. Medir como se distribuyen las palabras ambiguas:
   Para lograr esto se deben seguir las siguientes reglas:

   - Por la palabra adyacente a la derecha de la ambigua será puntuada con $+1W$.
   - Por la palabra adyacente a la izquierda de la ambigua, será puntuada con $-1W$.
   - Por cada palabra a k posiciones a la derecha de la ambigua será puntuada con $+kW$.
   - Por cada palabra a k posiciones a la izquierda de la ambigua será puntuada con $-kW$.

Entonces obtenemos una lista parecida a esta.(17)

Position	Collocation
-1 w	du *cote* la *cote* un *cote* notre *cote*
+1 w	*cote* ouest *cote* est *cote* du
+1w,+2w -2w,-1w	*cote* du gouvernement cote a *cote*
±k w ±k w ±k w	poisson (in ±*k* words) ports (in ±*k* words) opposition (in ±*k* words)

4. Ordenar las tablas de forma descendente según la función de verosimilitud (log-likelihood(18)):

$$Abs(Log(\frac{Pr(Accent\_Pattern_1|Collocation_i)}{Pr(Accent\_Pattern_2|Collocation_i)})) \quad (17)$$

Dado que likelihood es una función de probabilidad necesitamos probabilidades que obtendremos de la data:(17)

De-accented Form	Accent Pattern	%	Number
cesse	cesse	53%	669
	cessé	47%	593
cout	coût	100%	330
couta	coûta	100%	41
coute	coûté	53%	107
	coûte	47%	96
cote	côté	69%	2645
	côte	28%	1040
	cote	3%	99
	coté	<1%	15
cotiere	côtière	100%	296

y terminaremos con una lista como esta:(17)

LogL	Evidence	Classification
8.28	PREPOSITION QUE *terminara*	⇒ terminara
†7.24	de que *terminara*	⇒ terminara
†7.14	para que *terminara*	⇒ terminara
6.87	y *terminara*	⇒ terminará
6.64	WEEKDAY (within ±*k* words)	⇒ terminará
5.82	NOUN QUE *terminara*	⇒ terminará
†5.45	domingo (within ±*k* words)	⇒ terminará

(notese que es de otro ejemplo)

5. Opcionalmente podar e interpolar las listas

6. Entrenar las listas para ambigüedades mas generales:
   Esto es cuando la palabra que causa la ambigüedad puede ser incluida en una clase o conjunto de palabras con usos similares.

7. Usar las listas:
   Mientras se recorre el texto se revisa si a alguna palabra que se va leyendo le pertenece alguna lista de decisión, osea si se considera ambigua, ya sea de la misma palabra o de su clase, si es que pertenece a una clase. Si no se considera ambigua no pasa nada pero si sí, se recorre su lista hasta encontrar el mayor ranking que coincida con el contexto de la palabra ambigua.

Análisis de la complejidad:

1. Espacio:
   Supongamos que queremos analizar n ambigüedades de las cuales el máximo de significados que alguna pueda tener es k y queremos entrenar a la lista para reconocer estos significados en un máximo de c contextos. Entonces tendríamos n listas de tamaño $O(kc)$. Lo cual nos deja con una complejidad de $O(nkc)$ space.

2. Tiempo:

   - Para armar las listas:
     Obviando el paso opcional 5, para calcular la tabla solo se efectúan 2 operaciones: la medición de como se distribuyen las palabras ambiguas en el contexto y el ordenamiento de las listas según el resultado de la función log-likelihood.
     Para realizar la primera operación no se especifica ningún algoritmo así que supondremos que existe uno de complejidad $O(m)$ para puntuar las palabras de un contexto con un número máximo de m palabras y sabiendo que la lista tiene un tamaño total de n contextos podemos decir que la complejidad resultante de este paso es de $O(nm)$ time.
     Para la segunda operación sabemos que para una variable discreta, a diferencia de su forma para variables continuas, log-likelihood es una función matemática que no requiere del uso de métodos numéricos así que tiene una complejidad $\Theta(1)$ y para aplicarla a n elementos de la lista se necesita $\Theta(n)$ time. Y una ves calculados estos valores deberá ser ordenada, para lo cual cualquier algoritmo de ordenamiento eficiente servira asi que supondremos uno de complejidad $O(nLog(n))$ time. Entonces $\Theta(n) + O(nLog(n)) = O(nLog(n))$ time.

- Para usar las listas:
  Siguiendo las instrucciones indicadas en el paso 7 podemos intuir que se realiza un recorrido lineal a través de la lista lo cual nos daría un peor escenario de complejidad $O(n)$ time donde n son las filas de la lista. Sin embargo, debido a que los elementos están ordenados de forma descendente en la probabilidad de ser el significado real obtenemos un escenario promedio de complejidad $O(1)$ time. Y con p palabras ambiguas en la conseguimos la resolución de la ambigüedad en una cota inferior de $\Omega(p)$ y en una cota superior de $O(pn)$.

8. Alternativas: Inteligencia Artificial

La inteligencia artificial es la mejor solución para resolver el problema de la ambigüedad. Al entrenar e implementar una red neuronal, podemos lograr precisiones casi similares a la de los seres humanos, como lo detallamos en la sección anterior.

8.1. Implementación Simple de AI

El objetivo de esta experimentación es poder transmitir un poco de lo que se puede hacer con esta tecnología. En este caso, implementaremos una pequeña red neuronal para identificar si una titular de una noticia contiene 'sarcasmo' o no. Para luego, finalizar el proyecto con algunos futuros cambios que se podrían hacer para adaptarlo a gramáticas.

La siguiente implementación fue creada por Rishabh Misra (19).

```
import json
import tensorflow as tf

from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
```

En este caso utilizaremos la librerías de TensorFlow para hacer esto posible.

```
vocab_size = 10000
embedding_dim = 16
max_length = 100
trunc_type='post'
padding_type='post'
oov_tok = "<OOV>"
training_size = 20000
```

En primer lugar, tenemos que definir ciertos parámetros como el tamaño del vocabulario, la longitud máxima de una oración y el tamaño de nuestro entrenamiento.

```
!wget --no-check-certificate \
    https://storage.googleapis.com/laurencemoroney-blog.appspot.com/
   sarcasm.json \
    -O /tmp/sarcasm.json
```

Para ese experimento, utilizaremos un archivo JSON con los títulares y si efectivamente ese titular es sarcástico o no.

```
with open("/tmp/sarcasm.json", 'r') as f:
    datastore = json.load(f)

sentences = []
labels = []

for item in datastore:
    sentences.append(item['headline'])
    labels.append(item['is_sarcastic'])
```

Luego, interpretaremos el JSON y crearemos 2 listas con las oraciones y las etiquetas respectivamente.

```
training_sentences = sentences[0:training_size]
testing_sentences = sentences[training_size:]
training_labels = labels[0:training_size]
testing_labels = labels[training_size:]
```

Ahora, tenemos que dividir la data en 2 grupos. El grupo de entrenamiento y el grupo del testeo. Es muy importante asegurar que estos grupos contengan datos distintos. De lo contrario, obtendremos datos inexactos y violaremos principios de la red neuronal.

```
tokenizer = Tokenizer(num_words=vocab_size, oov_token=oov_tok)
tokenizer.fit_on_texts(training_sentences)

word_index = tokenizer.word_index

training_sequences = tokenizer.texts_to_sequences(training_sentences)
training_padded = pad_sequences(training_sequences, maxlen=max_length,
    padding=padding_type, truncating=trunc_type)

testing_sequences = tokenizer.texts_to_sequences(testing_sentences)
testing_padded = pad_sequences(testing_sequences, maxlen=max_length,
    padding=padding_type, truncating=trunc_type)
```

El primer paso será convertir todas las palabras a un índice. Dicho índice estará compuesto por un diccionario que tendrá como LLAVE, la palabra y como VALOR, la frecuencia (cuantas veces se repite) en el texto.

Finalmente, nosotros sabemos que la red neuronal no procesa las palabras tal como son. Es por ello que utilizaremos el Tokenizer para convertir las palabras a números. Dichos números serán transformados a secuencias: listas 2D que almacenarán oraciones con la respectiva frecuencia de las palabras que componen la oración.

```
model = tf.keras.Sequential([
    tf.keras.layers.Embedding(vocab_size, embedding_dim, input_length=
    max_length),
    tf.keras.layers.GlobalAveragePooling1D(),
    tf.keras.layers.Dense(24, activation='relu'),
    tf.keras.layers.Dense(1, activation='sigmoid')
])
model.compile(loss='binary_crossentropy',optimizer='adam',metrics=['
    accuracy'])
```

Ahora estamos llegando a la parte más importante del código: la definición del modelo. 'Embedding' consiste en la asociación de cada palabra con su sentimiento interpretado en un vector. Asimismo, *embedding_dim* define la dimensión del vector en cada palabra

Finalmente, en la compilación es donde sucede la magia. Nosotros ya sabemos que para este ejemplo solo tenemos 2 posibilidades, de que el titular sea sarcástico o no, es por ello que utilizaremos *binary_crossentropy* en loss. Asimismo, utilizaremos el optimizador de Adam.

```
model.summary()
```

Ahora podremos observar el modelo en la figura 4 .

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding (Embedding)        (None, 100, 16)           160000
_________________________________________________________________
global_average_pooling1d (Gl (None, 16)                0
_________________________________________________________________
dense (Dense)                (None, 24)                408
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 25
=================================================================
Total params: 160,433
Trainable params: 160,433
Non-trainable params: 0
_________________________________________________________________
```

Figura 3: Summary del modelo AI

```
num_epochs = 30
history = model.fit(training_padded, training_labels, epochs=num_epochs
    , validation_data=(testing_padded, testing_labels), verbose=2)
```

Ahora realizaremos el entrenamiento de nuestra red neuronal. Para ello realizaremos 30 Epochs[1] a nuestro modelo AI.

```
import matplotlib.pyplot as plt


def plot_graphs(history, string):
  plt.plot(history.history[string])
  plt.plot(history.history['val_'+string])
  plt.xlabel("Epochs")
  plt.ylabel(string)
  plt.legend([string, 'val_'+string])
  plt.show()

plot_graphs(history, "accuracy")
plot_graphs(history, "loss")
```

Ahora podremos visualizar el resultado de nuestro entrenamiento en la siguiente figura 4 .

```
sentence = ["granny starting to fear spiders in the garden might be
    real", "game of thrones season finale showing this sunday night"]
sequences = tokenizer.texts_to_sequences(sentence)
padded = pad_sequences(sequences, maxlen=max_length, padding=
    padding_type, truncating=trunc_type)
print(model.predict(padded))
```

Finalmente, se podrá probar a continuación. Al correr model.predict(padded) obtenemos el siguiente vector:

$$\begin{bmatrix}[9.0396011e-01] \\ [8.0303380e-07] \\ [4.1685485e-06]\end{bmatrix}$$

Esto quiere decir que existe una gran probabilidad de que la primera oración contenga sarcasmo, y una poca probabilidad que la segunda y tercera lo contenga.

8.2. Futuras implementaciones y cambios para utilizar AI en la detección de ambigüedad

La implementación mostrada en la última sección nos muestra el gran potencial de esta tecnología para resolver casos. Para poder aplicarlo con gramáticas, en la detección de ambigüedades, es necesario utilizar un modelo más complejo. Esto se debe

[1]Un Epoch es un término utilizado en Machine Learning que nos indica el número de pasadas del algoritmo al conjunto de entrenamiento.

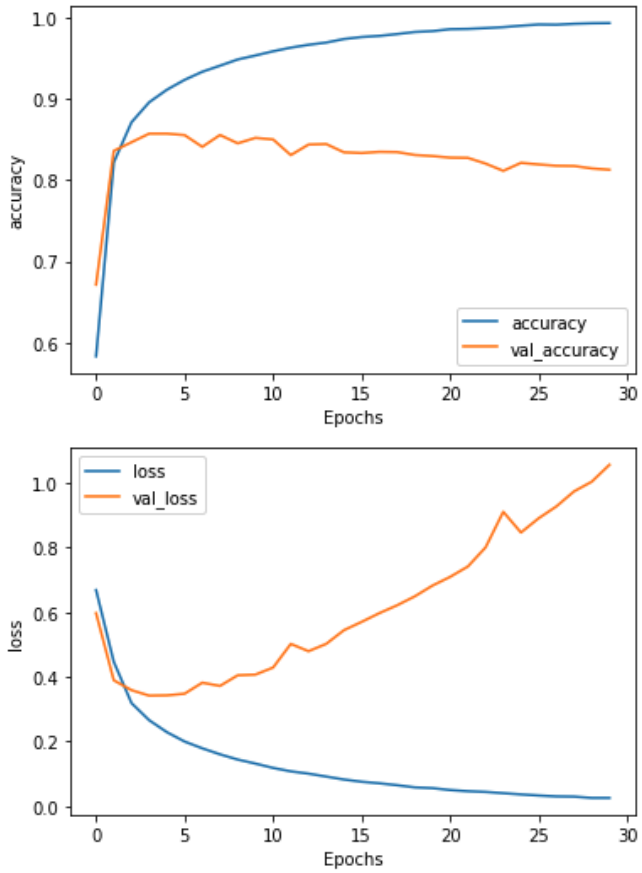

Figura 4: Resultado del entrenamiento de nuestra red neuronal

a que ya no utilizaríamos como 'lable' a un número binario (0 y 1), por el contrario, tendríamos que utilizar otra estrategia. El arte de construir un modelo, está en definir de manera precisa el 'Embedding', con el objetivo de que los vectores generados cuenten con la presión deseada. Este arte, aplicado en el lenguaje natural, normalmente es propio de las grandes empresas; sin embargo en nuestra experimentación, nos empapamos con la tecnología detrás y la forma en la que esto funciona.

9. GitHub

Nuestro proyecto se encuentra almacenado en el repositorio GitHub. Para acceder, podrá hacer **click aquí** .

Referencias

[1] B. Box, *Natural Language Processing 5 3 Context Free Grammars Part 1 1211*, 2018. Disponible en https://www.youtube.com/watch?v=VkXSpWc_8FM.

[2] A. McCallum, *Context Free Grammars, Introduction to Natural Language Processing*, 2017. Disponible en https://people.cs.umass.edu/~mccallum/courses/inlp2007/lect5-cfg.pdf.

[3] B. College, *Overview of NLP: Issues and Strategies*. Disponible en http://www.bowdoin.edu/~allen/nlp/nlp1.html.

[4] E. Shutova, *Context-free grammars and parsing*, 2015. Disponible en https://www.cl.cam.ac.uk/teaching/1516/L90/slides4-public.pdf.

[5] Wikipedia, *Context-free grammar*. https://en.wikipedia.org/wiki/Context-free_grammar.

[6] Wikipedia, *CYK Algorithm*. https://en.wikipedia.org/wiki/CYK_algorithm.

[7] Wikipedia, *Earley parser*. https://en.wikipedia.org/wiki/Earley_parser.

[8] Wikipedia, *LR parser*. https://en.wikipedia.org/wiki/LR_parser.

[9] Wikipedia, *LL parser*. https://en.wikipedia.org/wiki/LL_parser.

[10] U. J. D. Hopcroft, John E., *Introduction to Automata Theory, Languages, and Computation*. https://archive.org/details/introductiontoau00hopcz.

[11] A. Developer, *NSLinguisticTagger*. https://developer.apple.com/documentation/foundation/nslinguistictagger.

[12] S. Poddutur, *Syntaxnet Parsey McParseface Python Wrapper for Dependency-Parsing*. https://github.com/spoddutur/syntaxnet.

[13] A. M. St. Laurent, *Understanding Open Source and Free Software Licensing*, 2008. https://books.google.com/books?id=04jG7TTLujoC&pg=PA4.

[14] S. S. R. S. Slav Petrov, *Announcing SyntaxNet: The World's Most Accurate Parser Goes Open Source*. https://ai.googleblog.com/2016/05/announcing-syntaxnet-worlds-most.html.

[15] D. Roth, *Learning to Resolve Natural Language Ambiguities: A Unified Approach*. (danr@cs.uiuc.edu) Department of Computer Science University of Illinois. https://core.ac.uk/download/pdf/22874525.pdf.

[16] D. Andor, C. Alberti, D. Weiss, A. Severyn, A. Presta, K. Ganchev, S. Petrov, and M. Collins, "Globally normalized transition-based neural networks," *CoRR*, vol. abs/1603.06042, 2016. https://arxiv.org/pdf/1603.06042.pdf.

[17] D. Yarowsky, *DECISION LISTS FOR LEXICAL AMBIGUITY RESOLUTION: Application to Accent Restoration in Spanish and French*. https://arxiv.org/pdf/cmp-lg/9406034.pdf.

[18] t. f. e. Wikipedia, *Likelihood function*. https://en.wikipedia.org/wiki/Likelihood_function.

[19] A. Kausar, *NLP in TensorFlow — All You Need for a Kickstart*. https://medium.com/@aqsakausar30/nlp-in-tensorflow-all-you-need-for-a-kickstart-3293d7d2630e.