NLP **Procesamiento del lenguaje natural.**

Es un campo de AI que se dedica a procesar y entender el lenguaje humano (escrito).

Aplicaciones:
- google translate
- modelado de tópicos
- clasificación de texto
- etc

Semantic Slot Filling

¿Qué es?
Es una tarea de NLP

¿Qué hace?
Asignar categorías semánticas a cada parte del texto.

¿Cómo?
Discrimina cada palabra para saber de qué habla.

¿Para qué?
Le puede permitir al programa entender órdenes que el usuario escribe.

Formas de encarar esta metodología:

- Gramáticas libres de contexto

Lo que hacen es asignar como si fuera un diccionario, tareas o contextos a palabras. Las gramáticas libres de contexto identifican las distintas partes que puede tener eso.

Ventajas y desventajas:
uno tiene que manualmente escribir las distintas formas de gramática. Es imposible saber todas las opciones. Suele tener un precision muy alta y un recall muy bajo (tiene muchos falsos negativos).

- Con ML

Para esto necesitamos un corpus de entrenamiento que tenga taggeado para cada set de palabras su categoría semántica. Requiere de un proceso de ingeniería de features.

Se entrena un modelo que maximice la probabilidad de la categoría correspondiente a una palabra. Para esto deben estimarse los parámetros de nuestro modelo en base a un training corpus. Luego a la hora de predecir se devuelve el tag que más probabilidad tiene dada la palabra.

- Con deep learning

Es necesario un corpus de entrenamiento mucho más grande (para ajustar todos los parámetros de una red neuronal profunda). No es necesario generar features extra. Hay que definir la estructura de la red neuronal.

Pre-procesamiento

Los datos son no estructurados.

¿Qué es el texto desde el punto de vista de ML?
Es una secuencia de palabras.

¿Qué es una palabra?
Una unidad de texto que contiene la información necesaria y suficiente para nuestros modelos.

¿Cómo encontrar los límites entre una palabra y otra?
El procesamiento del lenguaje depende también del idioma. Para el español no va a ser el mismo que para el japonés, pero si se parece al del inglés.

- **Tokenización:**

  Separar un texto en tokens es lo que consideremos una unidad útil para el procesamiento semántico. Puede ser una palabra, oración, párrafo,conjunto de caracteres, etc.

  Hay problemas con los tokens. Falla con espacios y puntuación, o si una palabra tiene o no signo de pregunta por ejemplo (nos quedan tokens que no aportan información).
  Solución: se arreglan algunas reglas ad hoc.

- **Normalización**

  Es querer que un mismo token represente distintas palabras pero que tiene el mismo significado para nosotros, para lo que nos interese. Al tener menos tokens voy a tener menos procesamiento, y voy a tener modelos más simples.

  Stemming: heurística (no es determinista) es un proceso de eliminación y reemplazo de sufijos para llegar a la raíz de la palabra, que se denomina stem. No siempre da el resultado correcto.

  Lemmatization: se refiere a hacer las cosas correctamente, con el uso de un vocabulario y un análisis morfológico. Devuelve la base o la forma del diccionario de una palabra.

  ¿Cuál es mejor? depende. Si una palabra está mal escrita, con error de tipeo te conviene stemming, lemmatization no va a detectarlo bien. Con modismos tampoco funciona Lemmatization, no va a saber cómo normalizar. **Para cosas formales si conviene usar Lemmatization.**

La tokenización y la normalización son pasos dentro de la ingeniería de features.

Extraccion de informacion

¿Para qué sirve? Encontrar y entender partes importantes del texto para representarlo de manera relevante en nuestro mundo estructurado.
¿Cuáles son sus objetivos? organizar y distribuir la informacion de una manera **semanticamente precisa** para una computadora

- NER: name entity recognition: Reconocer y clasificar el texto en categorías predefinidas. Extraer información clara y objetiva.

- Colocaciones: Frases compuestas por más de una palabra que tienen un significado en particular cuando están juntas. Usos: desambiguación, traducción, generación de texto.

Se usa el coeficiente de Yule para detectar si una palabra es una colocación. Vemos la correlación entre las palabras.

Feature extraction

La parte en la que representamos al texto en números para el modelo.

BOW (bag of words)

Contar las ocurrencias de un token en particular en nuestro texto. La motivación es buscar palabras distintivas para ver si es algo triste o feliz. Para cada token tendremos una columna de características, a esto se le llama vectorización del texto. Un vector es un conjunto de números.

	good	movie	not	a	did	like
good movie	1	1	0	0	0	0
not a good movie	1	1	1	1	0	0
did not like	0	0	1	0	1	1

Todos los vectores tienen la misma dimensión

Nube de palabras:

Es un ejemplo visual. Pongo todas las palabras y el tamaño de cada palabra es relativo a cuántas veces aparece la palabra en mi texto.

Problemas:

- perdemos el orden de las palabras, de ahí “bag of words”.
- los contadores no están normalizados. Puede generar desbalances.
- la dimensión puede ser gigante.

Soluciones a algunos problemas:
Podemos **contar de a pares de tokens**, esto se conoce como **n-gram**. Puede solucionar el problema del orden. Pero esto dispara la dimensionalidad.
Para solucionar esto, podemos eliminar algunos n-grams de los features en función de su frecuencia de aparición en los documentos. Los de alta frecuencia se eliminan (generalmente son artículos, preposiciones).
n-grams de alta frecuencia → se les llama stop words.

Los n-grams de baja frecuencia tampoco los queremos porque pueden producir overfitting, suelen ser errores tipográficos, n-grams raros.

Nos quedamos con los n-grams de **media frecuencia**.

¿Hasta cuándo se puede sacar para no afectar el corpus? Hay que ver cada caso. Hay que ver si los stopwords rompen o destruyen información (ej: do en inglés).

TF-IDF

Separado en dos partes.

TF: tf(t,d): frecuencia del término (o n-gram) T en el documento D.
IDF: inverse document frequency.

Es una refinación de BOW. Cálculo **cuantas veces aparece el término en el documento y cuantas veces aparece en el corpus.**

Nos da la frecuencia de un término en varios documentos.

Un **valor alto** de TF-IDF se obtiene mediante una **frecuencia de término alta en el documento** dado y **una frecuencia baja del término en el corpus**

Mejora de BOW
- Usar TF-IDF (reemplazamos los contadores)
- Normalizar cada fila

(Clase 2)

Hipótesis distribucional

La hipótesis asume que la semántica de una palabra está determinada por el contexto en el que aparece.

Eso implica que podríamos obtener el significado de las palabras si las resumimos en todos los contextos en los cuales aparecen. Esto puede aproximarse con una reducción dimensional.

Aproximación con PCA

Corpus→ colección de documentos.

Para cada texto se hace BOW. Como obtenemos distintos vectores, se obtiene una matriz.
La matriz se traspone. El uno significa que la palabra aparece. Queremos las palabras con contextos en común cerca, pero la matriz es muy grande, la dimensión es demasiado grande. Se reduce con PCA.
Cuando reducimos con PCA, si dos palabras aparecen en las mismas frases, mismo vector. Si aparecen en frases parecidas, el vector será parecido.
Para ver qué tan buenos son estos vectores, se reduce a dos dimensiones con T-sne.

Con NLP estructuramos el texto mediante documentos (cada una de las entradas, puede ser un párrafo, un tweet, un capítulo de un libro). Todos ellos juntos forman el corpus. Con BOW buscaba vectorizar un documento. TF-IDF, más sofisticado, pone un coeficiente que tiene en cuenta todo el corpus de datos.

TF-IDF tiene una alta dimensionalidad. Por eso tenemos otros enfoques, que son Word2Vec y Fasttext. Ya no buscamos ver qué palabras aparecen en todo el texto y poner una dimensión por cada palabra, sino que vamos a intentar inferir la semántica de las palabras en base a sus vecindades.

Dos modelos que se inspiran en un área de *semántica distribucional*, que se basa en la hipótesis distribucional. Son dos enfoques que se basan en esto.

Word2Vec

El propósito es **agrupar los vectores** de palabras similares juntos en el espacio vectorial. ¿Cómo pone palabras cerca? **En base a si entiende que tienen semántica parecida.**

Si los vectores de las palabras están en una misma zona, tienen una semántica parecida, hacen referencia a cosas parecidas. La idea es posicionar a los vectores de mis palabras a que estén cerca de otros vectores con semántica parecida.

¿Cómo hace esto?
Usa dos enfoques: “skip-gram” o “CBOW”. Estos dos modelos lo que hacen es, dado un corpus, analizan las palabras de cada sentencia y tratan de usar las palabras vecinas para estimar la palabra deseada. ¿Cómo usan estas palabras? Mediante una red neuronal. Word2Vec usa estos dos enfoques para crear los vectores.

Diferencia entre Skip-gram y CBOW
Ambos buscan entrenar una red neuronal, pero se diferencian en cómo la entrenan.

- **Skip-gram**
  Lo que hace es que cada palabra prediga a sus vecinos.
  Para que dada una palabra, nos diga la probabilidad de que cada palabra del vocabulario sea su vecina. Skip-gram es al revés que CBOW.
  Funciona bien con una pequeña cantidad de datos de entrenamiento, representa bien incluso palabras o frases raras.
- **CBOW**
  Las palabras de la vecindad estimen a la palabra que yo quiero.
  Para que dada una nueva vecindad de palabras, nos diga la probabilidad de que la palabra del vocabulario sea su vecina.
  Más rápido de entrenar que Skip-gram, si tengo muchos datos me conviene. Además tiene una precisión ligeramente mejor para palabras frecuentes.

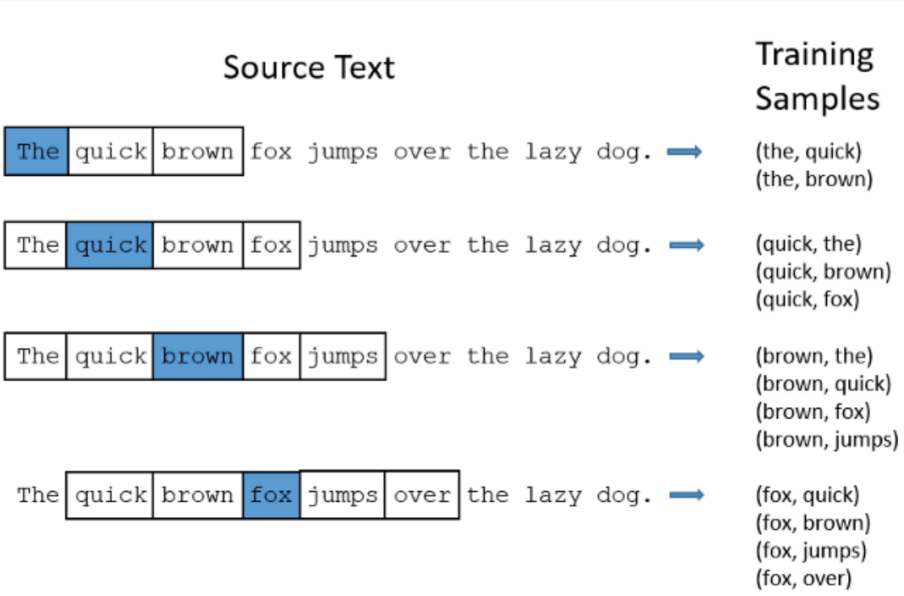
Voy a entrenar una red neuronal muy simple con una sola capa oculta. Lo que las neuronas de la red aprenden es la semántica de mis palabras en base a su contexto. Al final **me quedo con la capa oculta de la red**, ya que estas ajustan los valores en base a las relaciones entre las palabras (queda la representación que estoy buscando). Esta capa va a ser mis **“word vectors”** que estamos intentando aprender. Nos quedamos con los pesos de la capa oculta.

Hiper parámetros

- **Window size**

  Cuantas palabras quiero ver en mi vecindad. Cuantas palabras me interesa ver alrededor, es lo que le va a dar contexto a las palabras. Después para entrenar la red se usa de a pares.

  Por ejemplo, con una ventana de tamaño dos tenemos:



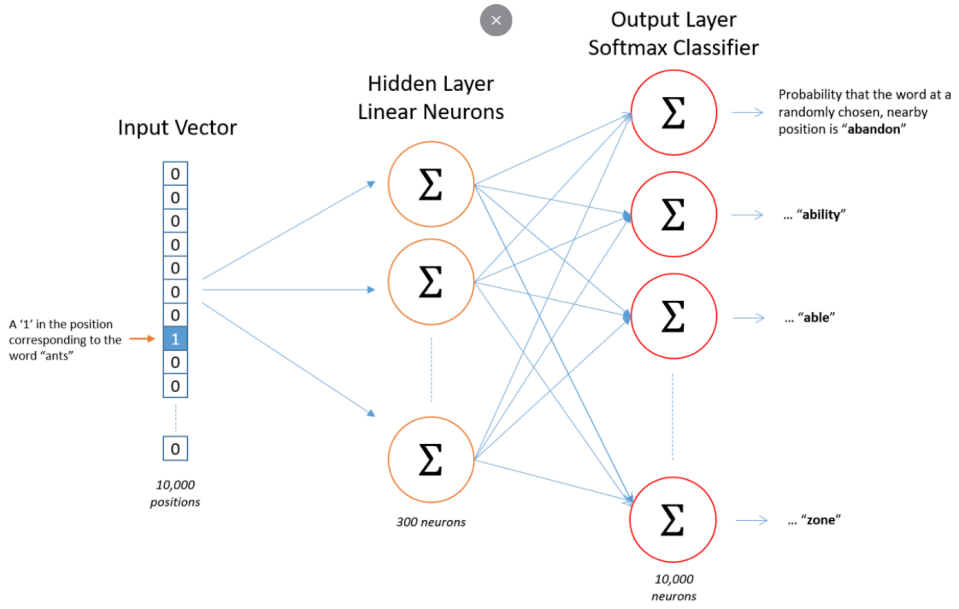
La red neuronal que entrena tiene una sola capa oculta. Las neuronas aprenden la semántica de las palabras en base a su contexto.

Input

La entrada de la red neuronal será una palabra representada como un one-hot vector. Este vector tiene tantas posiciones como tamaño tenga el vocabulario. Vector de ceros o unos.

Output

La salida de la red neuronal será también un vector con la misma dimensión que el vector de entrada. Representará las probabilidades de que cada una de las palabras sean vecinas de la palabra representada en la entrada.



La capa oculta

En el diagrama anterior hay 300 neuronas, con lo cual se están entrenando vectores de palabras con 300 características. La matriz de representación tendrá entonces tantas filas como palabras, y tantas columnas como neuronas (300 en este caso).

Después de entrenar a la red neuronal, lo que hace es multiplicar un vector representado en one-hot por la matriz de representación. Para el caso anterior, obtenemos una representación de 1x300. Ese va a ser el embedding de la palabra.

Otro ejemplo:

$$[0 \quad 0 \quad 0 \quad 1 \quad 0] \times \begin{bmatrix} 17 & 24 & 1 \\ 23 & 5 & 7 \\ 4 & 6 & 13 \\ 10 & 12 & 19 \\ 11 & 18 & 25 \end{bmatrix} = [10 \quad 12 \quad 19]$$

300 es un número arbitrario pero se puede ajustar.

La capa de salida

Solo la uso para el entrenamiento, después la saco. Debe usar un **clasificador Softmax**. Consigue que todos los valores estén entre 0 y 1, y que la suma de todos los valores sea 1. Es decir, que sea una distribución de probabilidades.

Resumen

- Luego de entrenar el modelo la capa oculta será nuestro embedding (representación vectorial)
- Para obtener el embedding de una nueva palabra, tomamos su representación one-hot-encoding la introducimos en la red y nos quedamos con los resultados de la capa oculta

Para algunos casos se puede realizar restas semánticas y obtener un nuevo concepto (al vector de rey le resto hombre y le sumo el de mujer y obtengo reina).

Desventajas
Incapacidad para lidiar con palabras fuera del corpus (al igual que TF-IDF).

Fasttext

Aparece como mejora de Word2Vec. **Trata cada palabra como un conjunto de n-grams**.
Mis tokens no van a ser palabras, van a ser conjuntos de caracteres. Una palabra va a ser representada por n-grams. **Inferimos la palabra como la suma de la representación vectorial de todos los n-grams.**
Permite generar embeddings de palabras no vistas en el training. ¿Cómo? calcula su vector en base a sus n-grams (el promedio). Tal vez no va a estimar super bien pero se va a aproximar bastante.

Hiper parámetros

- min n y max n

Estos indican la mínima y máxima cantidad de n-grams a usar para las representaciones.

Problema

Al modelar desde n-grams en vez de palabras la cantidad de elementos a representar crece mucho. Esto produce una gran demanda de memoria.

Solución:

- Poner una cantidad máxima de palabras a representar.
- Si durante el entrenamiento se llega a ocupar el 75% de los posibles valores, Fasttext realiza podas.
- Para eso incrementa el umbral mínimo de apariciones por palabra. Aquellas que queden por debajo son eliminadas.
- También usa estructuras de datos eficientes.

Implementación

- Usa multi threads y es más robusto.
- Trae implementadas funcionalidades para:
    - Obtener el embeddings de una palabra
    - Obtener el embedding de una oración
    - Clasificar texto
    - Obtener sinónimos y hasta analogías

Es una tarea supervisada. Con los embeddings entrenados que representan la palabra y tenemos ya similaridad entre palabras similares, al modelo final le resulta más fácil. Los embedding se pueden reutilizar.

Resumen de las técnicas: Bow < TF-IDF < Word2Vec < Fasttext

Bert: deep learning.
GPT-3: incluso mejor que las anteriores. Fue entrenado para predecir la próxima palabra dadas las anteriores sobre un montón de webs.