Resumen del capítulo: Representaciones del lenguaje

Insertados de palabras

Las máquinas no pueden trabajar directamente con palabras, imágenes y audio porque están diseñadas para trabajar con números. Los modelos de lenguaje transforman el texto en vectores y utilizan un conjunto de técnicas que se denominan colectivamente **insertados de palabras**. Esto significa que una palabra está embebida en un espacio vectorial que representa un modelo de lenguaje. Las diferentes áreas en ese espacio vectorial tienen diferentes significados desde la perspectiva del lenguaje.

Un vector creado a partir de una palabra mediante el insertado de palabras llevará información contextual sobre la palabra y sus propiedades semánticas. De esta forma, no se pierde la definición convencional de una palabra ni su significado en contexto.

Las propiedades semánticas son los componentes de una palabra que contribuyen a su significado. La palabra "marinero" tiene las siguientes propiedades semánticas: "masculino", "ocupación", "persona", "naval". Estas propiedades distinguen la palabra "marinero" de otras palabras. Pero las palabras, por lo general, no se sostienen solas y están rodeadas de otras palabras en una oración. Compara "El gato enredó todos los hilos" y "Perdí el hilo varias veces". La palabra "hilo" tiene un significado diferente según el contexto.

Este concepto permite trabajar con palabras, aunque estén vectorizadas, sin perder de vista el contexto. Esto significa que las palabras con contextos similares tendrán vectores similares. La distancia (de coseno, euclidiana, etc.) entre los vectores es una medida común de su similitud.

Word2vec

Entonces, ¿cómo funciona word2vec? Recordarás que en la lección anterior, las palabras "pinzón" y "frailecillo" se consideraron similares porque las dos se pueden usar en contexto con picos rojos, y las palabras "oso hormiguero" y "perezoso" se

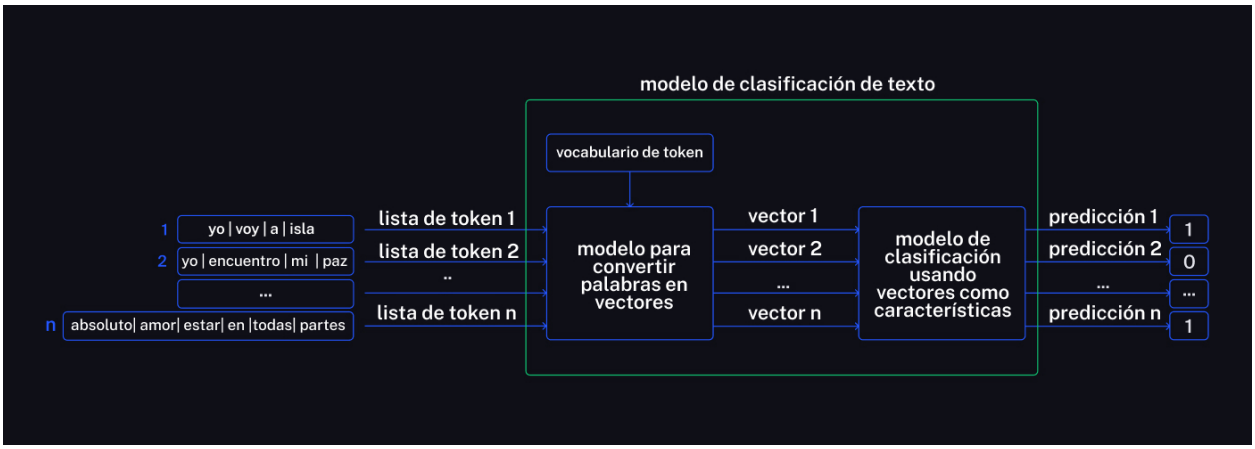
pueden usar en enunciados sobre Perú. Es decir, el significado de las palabras depende del contexto.

Otra cosa que puede hacer word2vec es entrenar un modelo para distinguir pares de vecinas verdaderas de las aleatorias. Esta tarea es como una tarea de clasificación binaria, donde las características son palabras y el objetivo es la respuesta a la pregunta de si las palabras son vecinas verdaderas o no.

Insertados para clasificación

Ahora vamos a descubrir cómo la representación vectorial puede ayudar a resolver tareas de clasificación y regresión. Digamos que hay un corpus de texto que necesita ser clasificado. En ese caso, nuestro modelo constará de dos bloques:

1. Modelos para convertir palabras en vectores: las palabras se convierten en vectores numéricos.
2. Modelos de clasificación: los vectores se utilizan como características.



Veamos los detalles:

1. Antes de pasar a la vectorización de las palabras, tenemos que realizar el preprocesamiento:

- Cada texto está tokenizado (descompuesto en palabras).

- Luego se lematizan las palabras (se reducen a su forma raíz). Sin embargo, los modelos más complejos, como BERT, no requieren este paso porque entienden las formas de las palabras.
- El texto se limpia de palabras vacías o caracteres innecesarios.
- Para algunos algoritmos (por ejemplo, BERT), se agregan tokens especiales para marcar el comienzo y el final de las oraciones.

2. Cada texto adquiere su propia lista de tokens después del preprocesamiento.

3. Luego, los tokens se pasan al modelo, que los vectoriza mediante el uso de un vocabulario de tokens precompilado. En la salida obtenemos vectores de longitud predeterminada formados para cada texto.

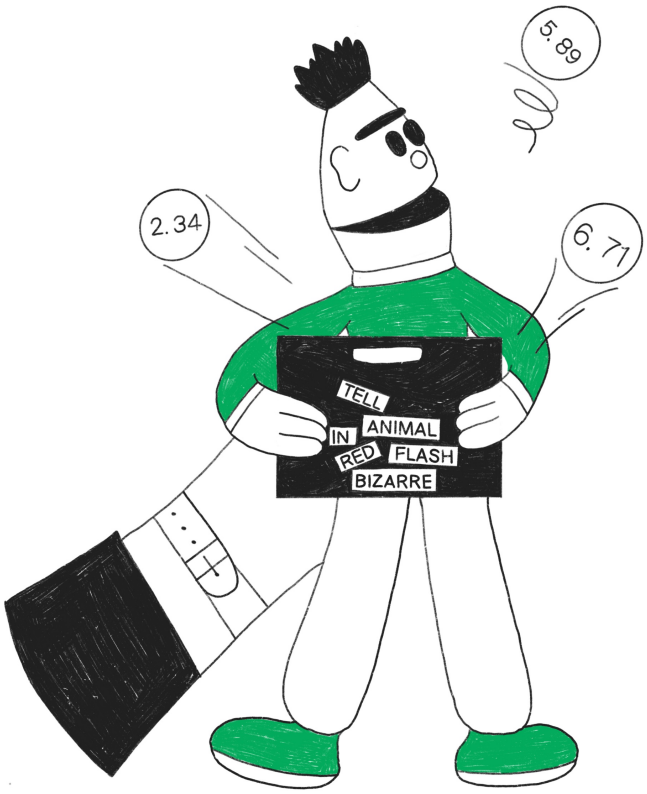
4. El paso final es pasar las características (vectores) al modelo. Luego, el modelo predice la tonalidad del texto: "0" — negativo o "1" — positivo.

BERT

BERT (Representaciones de codificador bidireccional de transformadores) es un modelo de red neuronal creado para la representación del lenguaje. Fue realizado por Google para mejorar la relevancia de los resultados de búsqueda y fue publicado en 2018 (el artículo original se puede encontrar en: https://arxiv.org/abs/1810.04805). Este algoritmo es capaz de "comprender" el contexto de un texto completo, no solo de frases cortas. BERT se usa con frecuencia en el machine learning para convertir textos en vectores. Los especialistas suelen utilizar modelos BERT existentes que están previamente entrenados (por Google o, posiblemente, por otros colaboradores) en grandes corpus de texto. Los modelos BERT previamente entrenados funcionan para muchos idiomas (104, con exactitud). Puedes entrenar tu propio modelo de representación del lenguaje, pero esto requerirá muchos recursos computacionales.

BERT es un paso evolutivo en comparación con word2vec y se convirtió rápidamente en la opción popular para los programadores e inspiró a los investigadores a crear otros modelos de representación de lenguaje: FastText, GloVe (vectores globales para representación de palabras), ELMO (insertados del modelo de lenguaje), GPT

(transformador generativo de preentrenamiento). Los modelos más precisos en la actualidad son BERT y GPT-3 (aunque este último no está disponible para el público en este momento).



Al procesar palabras, BERT considera tanto las palabras vecinas inmediatas como las palabras más lejanas. Esto permite que BERT produzca vectores precisos con respecto al significado natural de las palabras.

Así es como funciona:

- Aquí tienes un ejemplo de entrada para el modelo: "The red beak of the puffin [MASK] in the blue [MASK] " donde "**MASK**" representa palabras desconocidas o enmascaradas. El modelo tiene que adivinar cuáles son estas palabras enmascaradas.
- El modelo aprende a averiguar si las palabras del enunciado están relacionadas. Teníamos enmascaradas las palabras "flashed" y "sky". El modelo tiene que

comprender que una palabra sigue a la otra. Entonces, si ocultáramos la palabra "crawled" en lugar de "flashed", el modelo no encontraría una conexión.

BERT y preprocesamiento

Vamos a resolver una tarea de clasificación de reseñas de películas usando la representación del lenguaje BERT, es decir, usando BERT para crear vectores para palabras. Vamos a tomar un modelo previamente entrenado llamado bert-base-uncased (entrenado en textos en inglés en minúsculas)..

```
import torch
import transformers
```

Antes de convertir textos en vectores, necesitamos preprocesar el texto. BERT tiene su propio **tokenizador** basado en el corpus en el que fue entrenado. Otros tokenizadores no funcionan con BERT y no requieren lematización.

Pasos de preprocesamiento para el texto:

1. Inicializa el tokenizador como una instancia de `BertTokenizer()` con el nombre del modelo previamente entrenado.

```
tokenizer = transformers.BertTokenizer.from_pretrained('bert-base-uncased')
```

2. Convierte el texto en ID de tokens y el tokenizador BERT devolverá los ID de tokens en lugar de los tokens:

```
example = 'Es muy práctico utilizar transformadores'
ids = tokenizer.encode(example, add_special_tokens=True)
print(ids)
```

Obtenemos:

```
[101, 2009, 2003, 2200, 18801, 2000, 2224, 19081, 102]
```

Para operar el modelo correctamente, establecemos el argumento `add_special_tokens` en `True`. Significa que agregamos el token inicial (101) y el token final (102) a cualquier texto que se esté transformando.

3. BERT acepta vectores de una longitud fija, por ejemplo, de 512 tokens. Si no hay suficientes palabras en una cadena de entrada para completar todo el vector con tokens (o, más bien, sus identificadores), el final del vector se rellena con ceros. Si hay demasiadas palabras y la longitud del vector excede 510 (recuerda que se reservan dos posiciones para los tokens de inicio y finalización), o bien la cadena de entrada se limita al tamaño de 510, o bien se suelen omitir algunos identificadores devueltos por `tokenizer.encode()`, por ejemplo, todos los identificadores después de la posición 512 en la lista:

```
n = 512

padded = np.array(ids[:n] + [0]*(n - len(ids)))

print(padded)
```

Obtenemos:

```
[101 2009 2003 2200 18801 2000 2224 19081 102 0 0 0 0 ... 0 ]
```

Ahora tenemos que decirle al modelo por qué los ceros no tienen información significativa. Esto es necesario para el componente del modelo que se llama **attention**. Vamos a descartar estos tokens y crear una máscara para los tokens importantes, indicando valores cero y distintos de cero:

```
attention_mask = np.where(padded != 0, 1, 0)
print(attention_mask.shape)
```

Obtenemos:

```
(512, )
```

Insertados de BERT

Inicializa la configuración `BertConfig`. Pásale un archivo JSON con la descripción de la configuración del modelo. **JSON** (notación de objetos de JavaScript) es un flujo de números, letras, dos puntos y corchetes que devuelve un servidor cuando se le llama.

Inicializa el modelo de la clase `BertModel`. Pasa el archivo con el modelo previamente entrenado y la configuración:

```
config = BertConfig.from_pretrained('bert-base-uncased')
model = BertModel.from_pretrained('bert-base-uncased')
```

Vamos a comenzar por convertir textos en insertados. Esto puede tardar varios minutos, así que accede a la librería **tqdm** (árabe: taqadum, تقدّم, “progreso”), que muestra el progreso de la operación. Luego simplemente envuelve tu bucle en `tqdm()`. Utiliza `tqdm.auto` para importar la opción correcta para tu plataforma. Mira cómo funciona:

```
from tqdm.auto import tqdm

for i in tqdm(range(int(8e6))):
    pass

# aparecerá la barra de progreso
```

El modelo BERT crea insertados en lotes. Haz pequeño el tamaño del lote para que la RAM no se vea abrumada:

```
batch_size = 100
```

Haz un bucle para los lotes. La función `tqdm()` indicará el progreso:

```
# creación de una lista vacía de insertados de reseñas
embeddings = []

for i in tqdm(range(len(ids_list) // batch_size)):
    ...
```

Transforma los datos en un formato **tensor**. Tensor es un vector multidimensional en la librería Torch. El tipo de datos `LongTensor` almacena números en "formato largo", es decir, asigna 64 bits para cada número.

```
# unión de vectores de ids (de tokens) a un tensor
ids_batch = torch.LongTensor(ids_list[batch_size*i:batch_size*(i+1)])
# unión de vectores de máscaras de atención a un tensor
attention_mask_batch = torch.LongTensor(attention_mask_list[batch_size*i:batch_size*(i+
1)])
```

Pasa los datos y la máscara al modelo para obtener insertados para el lote:

```
batch_embeddings = model(ids_batch, attention_mask=attention_mask_batch)
```

Utiliza la función `no_grad()` (sin gradiente) para indicar que no necesitas gradientes en la librería Torch (los gradientes son necesarios para el modo de entrenamiento a la hora de crear tu propio modelo BERT). Esto hará que los cálculos sean más rápidos:

```
with torch.no_grad():
    batch_embeddings = model(ids_batch, attention_mask=attention_mask_batch)
```

Extrae los elementos requeridos del tensor y agrega la lista de todos los insertados:

```
# conversión de los elementos de tensor a numpy.array con la función numpy()
embeddings.append(batch_embeddings[0][:,0,:].numpy())
```

Uniendo todo lo anterior, obtenemos este bucle:

```
batch_size = 100

embeddings = []

for i in tqdm(range(len(ids_list) // batch_size)):

    ids_batch = torch.LongTensor(ids_list[batch_size*i:batch_size*(i+1)])
    attention_mask_batch = torch.LongTensor(attention_mask_list[batch_size*i:batch_size*(i
+1)])

    with torch.no_grad():
```

```
        batch_embeddings = model(ids_batch, attention_mask=attention_mask_batch)

    embeddings.append(batch_embeddings[0][:,0,:].numpy())
```

Llama a la función `concatenate()` para concatenar todos los insertados en una matriz de características:

```
features = np.concatenate(embeddings)
```