Transformers

La arquitectura **transformer** es la más usada para resolver tareas de Procesamiento de Lenguaje Natural. Se basa en un **codificador** que recibe un texto de entrada y procesa la información para extraer características de cada una de las palabras en el texto. Puede producir vectores que tienen información semántica de las palabras, a estos vectores se les conoce como **encajes**.

En este notebook veremos algunos ejemplos de cómo se pueden usar los modelos de lenguaje para realizar varias tareas.

Importante

Se recomienda usar una GPU para ejecutar este notebook. En el menú **Entorno de ejecución**, elegir Cambiar tipo de entorno de ejecución y en la sección **Acelerador por hardware** elegir una GPU o TPU.

Enmascarado de palabra.

La principal tarea que pueden hacer los modelos basados en transformers es la de predecir una palabra enmascarada. Esto se puede hacer mediante la herramienta `pipeline`.

```
from transformers import AutoTokenizer, AutoModelForMaskedLM, AutoModel
import torch
from transformers import pipeline
import matplotlib.pyplot as plt
import numpy as np
```

```
model_path = "dccuchile/bert-base-spanish-wwm-cased"
#model_path = "bertin-project/bertin-roberta-base-spanish"
#model_path = "google-bert/bert-base-multilingual-uncased"
#model_path = "guillermoruiz/mex_large"
```

Los **pipelines** son funciones que nos facilitan las tareas más frecuentes, por ejemplo predecir la palabra enmascarada.

```
unmasker = pipeline('fill-mask', model=model_path)
```

```
Device set to use cuda:0
```

```
text_masked = ["Por fin el [MASK] está bajando y ya puedo salir a la calle.",
               "Hoy vamos a comer unos [MASK].",
               "Me la pasé [MASK] la cena todo el día.",
               "Tengo que [MASK] bien porque mañana tengo examen.",
               "Mi propósito es hacer más [MASK].",
               "Me toca ir al [MASK] pero estoy muy cansado.",
               ]
#text_masked = ["Mexico_City _GEO " + t for t in text_masked] # para el caso mex_large
#text_masked = [t.replace(" [MASK]", " <mask>") for t in text_masked] # para el caso de berti
unmasker(text_masked[0])
```

```
[{'score': 0.1368665248155594,
  'token': 1505,
  'token_str': 'sol',
  'sequence': 'Por fin el sol está bajando y ya puedo salir a la calle.'},
 {'score': 0.08917335420846939,
  'token': 1577,
  'token_str': 'tiempo',
  'sequence': 'Por fin el tiempo está bajando y ya puedo salir a la calle.'},
 {'score': 0.07710042595863342,
  'token': 5241,
  'token_str': 'cielo',
  'sequence': 'Por fin el cielo está bajando y ya puedo salir a la calle.'},
 {'score': 0.07063408941030502,
  'token': 4425,
  'token_str': 'precio',
  'sequence': 'Por fin el precio está bajando y ya puedo salir a la calle.'},
 {'score': 0.04767605662345886,
  'token': 5994,
  'token_str': 'tráfico',
  'sequence': 'Por fin el tráfico está bajando y ya puedo salir a la calle.'}]
```

```python
for idx in range(6):
    res = unmasker(text_masked[idx])
    print()
    print(text_masked[idx])
    for r in res:
        print(r['token_str'])
```

```
Por fin el [MASK] está bajando y ya puedo salir a la calle.
sol
tiempo
cielo
precio
tráfico

Hoy vamos a comer unos [MASK].
huevos
[UNK]
dulces
pasteles
cereales

Me la pasé [MASK] la cena todo el día.
preparando
haciendo
con
en
comiendo

Tengo que [MASK] bien porque mañana tengo examen.
estar
hacerlo
pensarlo
ponerme
dormir

Mi propósito es hacer más [MASK].
dinero
daño
fuerte
luz
cosas
```

```
Me toca ir al [MASK] pero estoy muy cansado.
baño
trabajo
hospital
cine
aeropuerto
```

Encajes

Los encajes son vectores que incluyen información semántica de un texto. Los encajes de dos textos similares, van a dar una región similar en el espacio.

Para obtener los encajes de un texto, necesitamos el modelo de lenguaje y su **tokenizador** asociado. El tokenizador se encarga de partir el texto de entrada en **tokens** antes de meterlos al enconder. Un token se puede pensar que es una palabra.

Antes del entrenamiento, se usa el tokenizador para que seleccione la forma en que se va a partir el texto y también debe elegir el **vocabulario** del modelo. El vocabulario es el conjunto de tokens que reconce el modelo.

Veamos un ejemplo:

Primero cargamos el modelo y su tokenizador. El modelo lo pasamos a la GPU con

```
model.to(device)
```

```
device = "cuda:0" if torch.cuda.is_available() else "cpu"
print(device) ## Con esto verificamos que tengamos un dispositivo con cuda (GPU o TPU).
tokenizer = AutoTokenizer.from_pretrained(model_path)
model = AutoModel.from_pretrained(model_path)
model = model.to(device)
```

```
cuda:0
```

```
Some weights of BertModel were not initialized from the model checkpoint at dccuchile/bert-ba
You should probably TRAIN this model on a down-stream task to be able to use it for predictio
```

```
tokenizer.vocab_size
```

```
31002
```

```
#tokenizer.vocab
```

```
text = ["Por fin el calor está bajando y ya puedo salir a la calle.",
        "Hoy vamos a comer unos tacos.",
        "Me la pasé cocinando la cena todo el día.",
        "Tengo que dormir bien porque mañana tengo examen.",
        "Mi propósito es hacer más ejercicio.",
        "Me toca ir al gimnasio pero estoy muy cansado.",
        ]
#text = ["Mexico_City _GEO " + t for t in text] # para el caso mex_large
```

```
tokenizer.tokenize(text[1])
```

```
['Hoy', 'vamos', 'a', 'comer', 'unos', 'tac', '##os', '.']
```

El tokenizador tiene varias opciones, - `return_tensors="pt"`: Pedimos que regrese los tokens como un tensor de Pytorch. Esta opción depende de si cargamos el modelo en Pytorch o Tensorflow. - `padding="max_length"`: Los tensores con los tokens deben tener todos el mismo tamaño. Si un mensaje es más corto, se rellena con el token especial **PADDING**. - `max_length=20`: Especificamos el tamaño de los tensores. - `truncation=True`: Los mensajes con más tokens que `max_length` se truncan.

```
tokens = tokenizer(text, return_tensors="pt", padding='max_length', max_length=20, truncatio
```

```
tokens
```

```
{'input_ids': tensor([[    4,  1278,  1377,  1040,  7110,  1266, 21137,  1042,  1526,  1769,
          3143,  1013,  1030,  3783,  1009,     5,     1,     1,     1,     1],
        [    4,  4894,  2229,  1013,  2073,  2438, 16718,  1011,  1009,     5,
             1,     1,     1,     1,     1,     1,     1,     1,     1,     1],
        [    4,  1369,  1030, 15748,  6765, 30935,  1047,  1030,  6997,  1397,
          1040,  1726,  1009,     5,     1,     1,     1,     1,     1,     1],
        [    4,  2190,  1038,  5182,  1311,  1817,  2591,  1847,  4165,  1009,
             5,     1,     1,     1,     1,     1,     1,     1,     1,     1],
        [    4,  1451,  6079,  1058,  1409,  1216,  4376,  1009,     5,     1,
             1,     1,     1,     1,     1,     1,     1,     1,     1,     1],
        [    4,  1369,  9537,  1628,  1091, 17680,  1355,  1764,  1456, 10634,
          1009,     5,     1,     1,     1,     1,     1,     1,     1,     1]]), 'token_type
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
```

```
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]), 'attention_mask': ten
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]])}
```

A todos los mensajes se les agrega un token de inicio y uno de fin (en este caso son el 4 y 5). El texto y los tokens se pueden ver con `decode`.

```
tokenizer.decode(tokens['input_ids'][0])
```

```
'[CLS] Por fin el calor está bajando y ya puedo salir a la calle. [SEP] [PAD] [PAD] [PAD] [PA
```

Como el modelo están en la GPU, primero debemos mandar los tensores `input_ids` y `attention_mask` a la GPU. Después, los tensores ya se pueden meter al modelo.

```
with torch.no_grad():
  t = {"input_ids": tokens['input_ids'].to(device), "attention_mask": tokens['attention_mask
  output = model(**t)
```

La salida se guarda en la variable `output`. Los encajes de los tokens se guardan en `last_hidden_state`

```
output.last_hidden_state.shape
```

```
torch.Size([6, 20, 768])
```

Es común que se use el vector asociado al primer token (el de inicio de oración) para describir el texto completo.

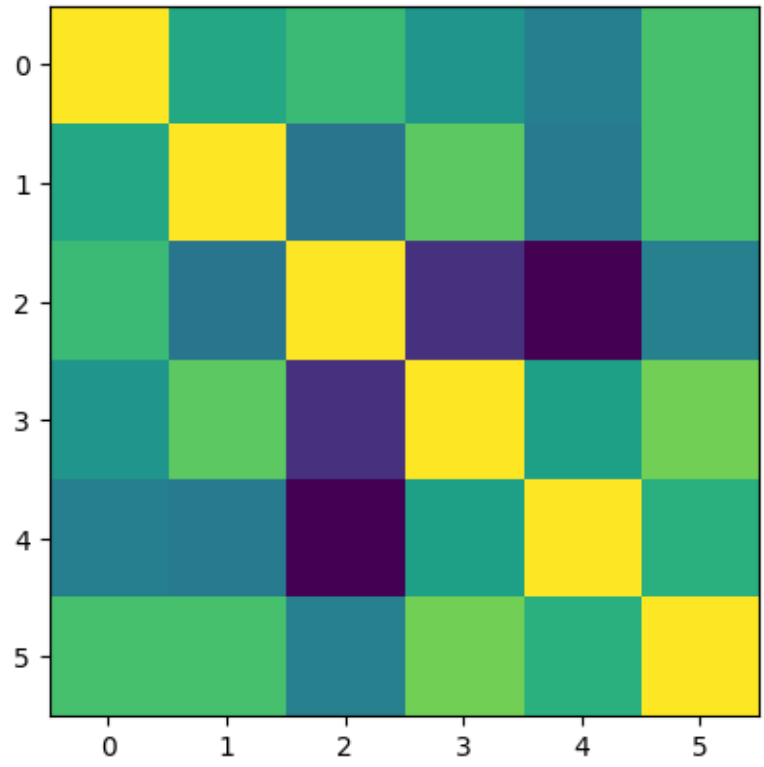
```
emb = output.last_hidden_state[:,0]
emb.shape
```

```
torch.Size([6, 768])
```

Ahora vamos a comparar los encajes de las 5 oraciones de ejemplo.

```
emb = torch.nn.functional.normalize(emb, p=2, dim=1)
sim = emb @ emb.T
print(sim)
plt.imshow(sim.cpu())
plt.show()
```

```
tensor([[1.0000, 0.8781, 0.9023, 0.8528, 0.8254, 0.9104],
        [0.8781, 1.0000, 0.8121, 0.9223, 0.8188, 0.9105],
        [0.9023, 0.8121, 1.0000, 0.7355, 0.6920, 0.8267],
        [0.8528, 0.9223, 0.7355, 1.0000, 0.8666, 0.9342],
        [0.8254, 0.8188, 0.6920, 0.8666, 1.0000, 0.8875],
        [0.9104, 0.9105, 0.8267, 0.9342, 0.8875, 1.0000]], device='cuda:0')
```




```
text
```

```
['Por fin el calor está bajando y ya puedo salir a la calle.',
 'Hoy vamos a comer unos tacos.',
 'Me la pasé cocinando la cena todo el día.',
```

```
 'Tengo que dormir bien porque mañana tengo examen.',
 'Mi propósito es hacer más ejercicio.',
 'Me toca ir al gimnasio pero estoy muy cansado.']
```

Cuando queremos obtener los encajes de una gran cantidad de textos, se recomienda hacerlo por lotes para no revasar la memoria RAM de la tarjeta de video.

```
text*5
```

```
['Por fin el calor está bajando y ya puedo salir a la calle.',
 'Hoy vamos a comer unos tacos.',
 'Me la pasé cocinando la cena todo el día.',
 'Tengo que dormir bien porque mañana tengo examen.',
 'Mi propósito es hacer más ejercicio.',
 'Me toca ir al gimnasio pero estoy muy cansado.',
 'Por fin el calor está bajando y ya puedo salir a la calle.',
 'Hoy vamos a comer unos tacos.',
 'Me la pasé cocinando la cena todo el día.',
 'Tengo que dormir bien porque mañana tengo examen.',
 'Mi propósito es hacer más ejercicio.',
 'Me toca ir al gimnasio pero estoy muy cansado.',
 'Por fin el calor está bajando y ya puedo salir a la calle.',
 'Hoy vamos a comer unos tacos.',
 'Me la pasé cocinando la cena todo el día.',
 'Tengo que dormir bien porque mañana tengo examen.',
 'Mi propósito es hacer más ejercicio.',
 'Me toca ir al gimnasio pero estoy muy cansado.',
 'Por fin el calor está bajando y ya puedo salir a la calle.',
 'Hoy vamos a comer unos tacos.',
 'Me la pasé cocinando la cena todo el día.',
 'Tengo que dormir bien porque mañana tengo examen.',
 'Mi propósito es hacer más ejercicio.',
 'Me toca ir al gimnasio pero estoy muy cansado.',
 'Por fin el calor está bajando y ya puedo salir a la calle.',
 'Hoy vamos a comer unos tacos.',
 'Me la pasé cocinando la cena todo el día.',
 'Tengo que dormir bien porque mañana tengo examen.',
 'Mi propósito es hacer más ejercicio.',
 'Me toca ir al gimnasio pero estoy muy cansado.']
```

```
tokens = tokenizer(text*500, return_tensors="pt", padding='max_length', max_length=128, trunc
```

```
tokens['input_ids'].shape
```

```
torch.Size([3000, 128])
```

```
with torch.no_grad():
  t = {"input_ids": tokens['input_ids'].to(device), "attention_mask": tokens['attention_mask
  output = model(**t)
output.last_hidden_state.shape
```

```
OutOfMemoryError: CUDA out of memory. Tried to allocate 4.39 GiB. GPU 0 has a total capacity
---------------------------------------------------------------------------
OutOfMemoryError                          Traceback (most recent call last)
<ipython-input-62-975a47c174b3> in <cell line: 0>()
      1 with torch.no_grad():
      2   t = "input_ids": tokens['input_ids'].to(device), "attention_mask": tokens['attentio
----> 3   output = model(**t)
      4 output.last_hidden_state.shape
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in _wrapped_call_impl(self
   1737             return self._compiled_call_impl(*args, **kwargs)  # type: ignore[misc]
   1738         else:
-> 1739             return self._call_impl(*args, **kwargs)
   1740
   1741     # torchrec tests the code consistency with the following code
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in _call_impl(self, *args,
   1748                 or _global_backward_pre_hooks or _global_backward_hooks
   1749                 or _global_forward_hooks or _global_forward_pre_hooks):
-> 1750             return forward_call(*args, **kwargs)
   1751
   1752         result = None
/usr/local/lib/python3.11/dist-packages/transformers/models/bert/modeling_bert.py in forward(
   1014         head_mask = self.get_head_mask(head_mask, self.config.num_hidden_layers)
   1015
-> 1016         encoder_outputs = self.encoder(
   1017             embedding_output,
   1018             attention_mask=extended_attention_mask,
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in _wrapped_call_impl(self
   1737             return self._compiled_call_impl(*args, **kwargs)  # type: ignore[misc]
   1738         else:
-> 1739             return self._call_impl(*args, **kwargs)
```

```
   1740
   1741     # torchrec tests the code consistency with the following code
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in _call_impl(self, *args
   1748                 or _global_backward_pre_hooks or _global_backward_hooks
   1749                 or _global_forward_hooks or _global_forward_pre_hooks):
-> 1750             return forward_call(*args, **kwargs)
   1751
   1752         result = None
/usr/local/lib/python3.11/dist-packages/transformers/models/bert/modeling_bert.py in forward
    660                 )
    661             else:
--> 662                 layer_outputs = layer_module(
    663                     hidden_states,
    664                     attention_mask,
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in _wrapped_call_impl(sel
   1737             return self._compiled_call_impl(*args, **kwargs)  # type: ignore[misc]
   1738         else:
-> 1739             return self._call_impl(*args, **kwargs)
   1740
   1741     # torchrec tests the code consistency with the following code
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in _call_impl(self, *args
   1748                 or _global_backward_pre_hooks or _global_backward_hooks
   1749                 or _global_forward_hooks or _global_forward_pre_hooks):
-> 1750             return forward_call(*args, **kwargs)
   1751
   1752         result = None
/usr/local/lib/python3.11/dist-packages/transformers/models/bert/modeling_bert.py in forward
    592             present_key_value = present_key_value + cross_attn_present_key_value
    593
--> 594         layer_output = apply_chunking_to_forward(
    595             self.feed_forward_chunk, self.chunk_size_feed_forward, self.seq_len_dim,
    596         )
/usr/local/lib/python3.11/dist-packages/transformers/pytorch_utils.py in apply_chunking_to_fo
    251         return torch.cat(output_chunks, dim=chunk_dim)
    252
--> 253     return forward_fn(*input_tensors)
    254
    255
/usr/local/lib/python3.11/dist-packages/transformers/models/bert/modeling_bert.py in feed_for
    604
    605     def feed_forward_chunk(self, attention_output):
--> 606         intermediate_output = self.intermediate(attention_output)
    607         layer_output = self.output(intermediate_output, attention_output)
```

```
    608         return layer_output
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in _wrapped_call_impl(self
   1737             return self._compiled_call_impl(*args, **kwargs)  # type: ignore[misc]
   1738         else:
-> 1739             return self._call_impl(*args, **kwargs)
   1740
   1741     # torchrec tests the code consistency with the following code
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in _call_impl(self, *args
   1748                 or _global_backward_pre_hooks or _global_backward_hooks
   1749                 or _global_forward_hooks or _global_forward_pre_hooks):
-> 1750             return forward_call(*args, **kwargs)
   1751
   1752         result = None
/usr/local/lib/python3.11/dist-packages/transformers/models/bert/modeling_bert.py in forward
    505     def forward(self, hidden_states: torch.Tensor) -> torch.Tensor:
    506         hidden_states = self.dense(hidden_states)
--> 507         hidden_states = self.intermediate_act_fn(hidden_states)
    508         return hidden_states
    509
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in _wrapped_call_impl(self
   1737             return self._compiled_call_impl(*args, **kwargs)  # type: ignore[misc]
   1738         else:
-> 1739             return self._call_impl(*args, **kwargs)
   1740
   1741     # torchrec tests the code consistency with the following code
/usr/local/lib/python3.11/dist-packages/torch/nn/modules/module.py in _call_impl(self, *args
   1748                 or _global_backward_pre_hooks or _global_backward_hooks
   1749                 or _global_forward_hooks or _global_forward_pre_hooks):
-> 1750             return forward_call(*args, **kwargs)
   1751
   1752         result = None
/usr/local/lib/python3.11/dist-packages/transformers/activations.py in forward(self, input)
     67
     68     def forward(self, input: Tensor) -> Tensor:
---> 69         return self.act(input)
     70
     71
OutOfMemoryError: CUDA out of memory. Tried to allocate 4.39 GiB. GPU 0 has a total capacity
```

Con este código se pueden obtener los encajes de una gran cantidad de textos.

```
def predict(text, bs=128):
    output = []
```

```
    for i in range(0, len(text), bs):
        if i//bs%100==99:
            print(i, "/", len(text))
        tokens = tokenizer(text[i: i+bs], return_tensors="pt", padding='max_length', max_leng
        t = {"input_ids": tokens['input_ids'].to(device), "attention_mask": tokens['attentio

        with torch.no_grad():
            pred = model(**t).last_hidden_state[:,0].cpu()
        output.append(pred)
    output = torch.cat(output, dim=0)
    return output
```

```
output = predict(text*1000)
output.shape
```

```
torch.Size([6000, 768])
```

Se normalizan los vectores y se guardan en el archivo **encajes.pt**

```
output = torch.nn.functional.normalize(output, p=2, dim=1)
torch.save(output, "encajes.pt")
```

Ejercicio

Usar los datos del archivo **emojis_train.csv**. Son 11774 mensajes de Twitter etiquetados con los emojis , , y con la siguiente distribución:

Emoji	Total
	4738
	3654
	2182
	1200

Usando los modelos - google-bert/bert-base-multilingual-uncased - dccuchile/bert-base-spanish-wwm-cased - bertin-project/bertin-roberta-base-spanish - guillermoruiz/mex_large

obtener los encajes de los mensajes y normalizarlos (ponerlos con norma 1). Usar UMAP para graficar los vectores y colorearlos deacuerdo a su etiqueta.

Comparar los gráficos y argumentar las diferencias.

Por ejemplo, en la siguiente imagen se muestran los encajes usando Beto.

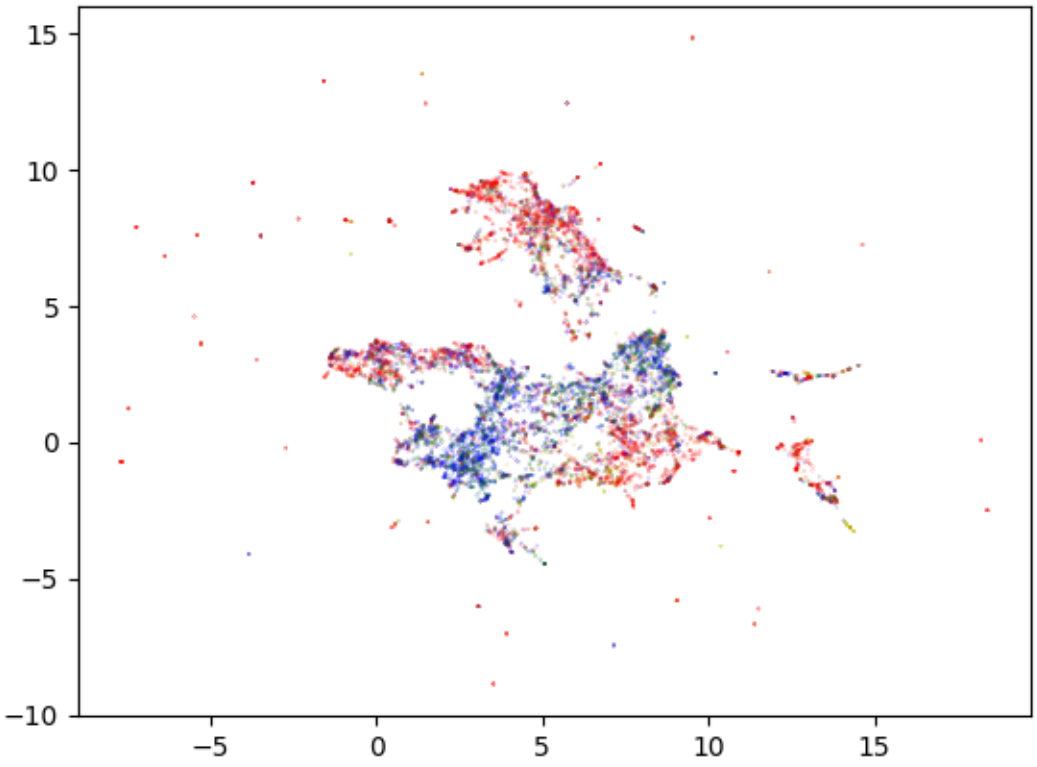


Figure 1: emb_beto.png