PLN

July 15, 2025

1 Trabajo Final de la materia Procesamiento de Lenguaje Natural

- python -m pip install –upgrade pip
- pip install torch torchvision torchaudio –index-url https://download.pytorch.org/whl/cu118
- pip install transformers datasets scikit-learn pandas ipywidgets spacy
- python -m spacy download es_core_news_sm

[1]:
```
import json
import logging
import os
import re
import time
import glob
import pandas as pd
import spacy
import torch
from tqdm import tqdm
from transformers import BertTokenizer, BertForSequenceClassification, Trainer,␣
↪TrainingArguments
from datasets import Dataset
from sklearn.metrics import accuracy_score, precision_recall_fscore_support
from collections import Counter
from pathlib import Path

# Configuración del logger
logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s -␣
↪%(message)s')
logger = logging.getLogger(__name__)
```

2 PARTE 1: Preprocesamiento del corpus

[2]:
```
class PreprocesadorTexto:
    def __init__(self):
        self.nlp = spacy.load("es_core_news_sm")

    def segmentar_frases(self, texto):
```

```
        return [f.strip() for f in re.split(r'(?<=[.!?])\s+', texto) if f.
↪strip()]

    def extraer_entidades(self, texto):
        try:
            doc = self.nlp(texto)
            return list(set(ent.text for ent in doc.ents if ent.label_ in␣
↪{"PER", "ORG", "LOC"}))
        except:
            return []
```

[3]:
```
class ProcesadorCorpusSinModelo:
    def __init__(self, corpus_dir, output_dir, preprocesador):
        self.corpus_dir = corpus_dir
        self.output_dir = output_dir
        self.preprocesador = preprocesador

    def procesar(self):
        inicio = time.time()
        archivos = []  # (fn, txt, categoria)

        for categoria in ['izquierda', 'derecha', 'neutral']:
            path = os.path.join(self.corpus_dir, categoria)
            if not os.path.isdir(path):
                continue
            for fn in os.listdir(path):
                if fn.endswith('.txt'):
                    with open(os.path.join(path, fn), encoding='utf-8') as f:
                        txt = f.read()
                    archivos.append((fn, txt, categoria))

        os.makedirs(self.output_dir, exist_ok=True)

        for fn, txt, categoria in tqdm(archivos, desc='Preprocesando'):
            frases = self.preprocesador.segmentar_frases(txt)
            frases_procesadas = []

            for frase in frases:
                entidades = self.preprocesador.extraer_entidades(frase)
                frases_procesadas.append({
                    "frase": frase,
                    "entidades_detectadas": entidades
                })

            out_json = {
                "etiqueta_original": categoria,
                "frases": frases_procesadas,
```

```
                "entidades_mencionadas": self.preprocesador.
↪extraer_entidades(txt)
            }

            out_path = os.path.join(self.output_dir, fn.replace(".txt", ".
↪json"))
            with open(out_path, 'w', encoding='utf-8') as f:
                json.dump(out_json, f, indent=2, ensure_ascii=False)

        duracion = int(time.time() - inicio)
        print(f"Listo. Tiempo total: {duracion // 60}m {duracion % 60}s")
```

[4]:
```
base_dir = os.getcwd()
corpus_dir = os.path.join(base_dir, 'transcripciones')
output_dir = os.path.join(base_dir, 'resultados')

preprocesador = PreprocesadorTexto()
procesador = ProcesadorCorpusSinModelo(corpus_dir, output_dir, preprocesador)
procesador.procesar()
```

```
Preprocesando: 100%|      | 3115/3115 [1:12:04<00:00,  1.39s/it]
```

```
Listo. Tiempo total: 72m 5s
```

3 PARTE 2: Análisis de entidades mencionadas

[5]:
```
# Ruta a la carpeta de resultados
carpeta_resultados = "./resultados"

# Contador de todas las entidades mencionadas
contador_entidades = Counter()

# Recorrer todos los archivos JSON
for ruta in glob.glob(os.path.join(carpeta_resultados, "*.json")):
    try:
        with open(ruta, "r", encoding="utf-8") as f:
            data = json.load(f)
            entidades = data.get("entidades_mencionadas", [])
            contador_entidades.update(entidades)
    except Exception as e:
        print(f"Error en {ruta}: {e}")

# Crear DataFrame ordenado por frecuencia
df_entidades = pd.DataFrame(contador_entidades.items(), columns=["entidad",␣
↪"frecuencia"])
```

```
df_entidades = df_entidades.sort_values(by="frecuencia", ascending=False).
 ↪reset_index(drop=True)

# Guardar CSV en la carpeta principal del proyecto
df_entidades.to_csv("entidades_frecuentes.csv", index=False, encoding="utf-8")

# Mostrar vista previa
df_entidades.head(50)
```

[5]:

	entidad	frecuencia
0	¿	2140
1	Argentina	1686
2	Bueno	1529
3	la Argentina	1299
4	Cristina	1251
5	Alberto Fernández	1232
6	Estado	1046
7	Macri	1025
8	¿Qué	1018
9	Cristina Fernández de Kirchner	968
10	Milei	945
11	Vamos	915
12	Así	765
13	Además	742
14	Después	736
15	También	736
16	Congreso	725
17	Javier Milei	693
18	Ayer	690
19	Mauricio Macri	689
20	Alberto	663
21	Sergio Massa	563
22	Cristina Kirchner	553
23	Patricia Bullrich	530
24	Estados Unidos	520
25	provincia de Buenos Aires	517
26	Acá	502
27	Qué	493
28	Massa	477
29	Está	472
30	llegó	464
31	Néstor Kirchner	433
32	Senado	428
33	Máximo Kirchner	426
34	Claro	419
35	Quiero	413
36	Axel Kicillof	389

37	Brasil	386
38	Buenos Aires	384
39	Primero	379
40	Banco Central	377
41	La Cámpora	370
42	¿Quién	368
43	Venezuela	362
44	Kirchner	359
45	Mirá	358
46	Ustedes	355
47	Mira	347
48	¿Cuál	342
49	Kicillof	342

[10]:
```
# Lista manual de palabras irrelevantes detectadas manualmente
irrelevantes = {
    "ayer", "acá", "mira", "mirá", "miren", "obviamente", "corte", "digo",␣
 ↪"mire",
    "gracias", "vos", "tenés", "ojalá", "eh", "mañana", "apellidos", "nombres",
    "dios", "perdón", "escuchémoslo", "jefa", "che", "¿", "bueno", "qué",␣
 ↪"vamos",
    "así", "además", "después", "también", "está", "llegó", "claro", "quiero",
    "primero", "¿quién", "¿cuál", "tenía", "cambio", "había", "sí", "ahí",␣
 ↪"juntos",
    "más", "según", "hola", "estoy", "ningún", "dale", "cómo", "esa", "papa",␣
 ↪"jamás",
    "coherencia", "empezó", "ah", "tenemos", "van", "bien", "me", "nos",␣
 ↪"todavía",
    "generó", "dicen", "recién", "cuál", "espero", "habló", "voy", "insisto",␣
 ↪"allí",
    "él", "unión", "cohesión", "ustedes", "jorge", "le", "k", "recuerdo",␣
 ↪"decía", "boca",
    "chicos", "ojo", "usted", "ortografía", "segundo", "pablo", "luego",␣
 ↪"esos", "lamentablemente",
    "podría", "poneme", "mauro", "quilombo", "somos", "miralo", "nuestro",␣
 ↪"habra", "quieren",
    "andres", "míralo", "habrá", "nada", "corrección", "whatsapp", "repito",␣
 ↪"encontré",
    "aquí", "afi", "anoche", "señor"
}

# Cargar CSV original
df = pd.read_csv("entidades_frecuentes.csv")
df = df[df["entidad"].notna()]
df["entidad"] = df["entidad"].str.strip().str.lower()
```

```
# Filtrar: eliminar entradas no alfabéticas y palabras irrelevantes
df = df[df["entidad"].str.isalpha()]
df = df[~df["entidad"].isin(irrelevantes)]

# Agrupar por entidad en caso de duplicados
df = df.groupby("entidad", as_index=False).sum()

# Seleccionar las 100 más frecuentes
df_top100 = df.sort_values(by="frecuencia", ascending=False).head(100)

# Guardar resultado
df_top100.to_csv("entidades_top100_limpias.csv", index=False)

print("Archivo 'entidades_top100_limpias.csv' creado con 100 entidades útiles.")
df_top100.head(100)
```

```
Archivo 'entidades_top100_limpias.csv' creado con 100 entidades útiles.
```

[10]:

	entidad	frecuencia
1064	argentina	1736
3971	cristina	1253
5473	estado	1046
8850	macri	1027
9591	milei	954
…	…	…
10286	nicaragua	81
3168	chubut	80
2734	carlos	78
11302	perú	77
6199	gabinete	77

[100 rows x 2 columns]

[11]:
```
# Cargar entidades útiles
df = pd.read_csv("entidades_top100_limpias.csv")
entidades_utiles = set(df["entidad"].str.lower())

# Definir rutas
carpeta_jsons = "resultados"
carpeta_transcripciones = "transcripciones"
carpeta_salida_base = "resultados_filtrados"
subcarpetas = ["izquierda", "derecha", "neutral"]

# Crear subcarpetas
for sub in subcarpetas:
    os.makedirs(os.path.join(carpeta_salida_base, sub), exist_ok=True)

def entidades_presentes(frase):
```

```
    palabras = frase.lower().split()
    return [ent for ent in entidades_utiles if ent in palabras]

# Procesar JSONs
for archivo in os.listdir(carpeta_jsons):
    if not archivo.endswith(".json"):
        continue

    texto_id = archivo.replace(".json", "")
    etiqueta = None
    for sub in subcarpetas:
        posible_ruta = os.path.join(carpeta_transcripciones, sub, texto_id + ".
 ↪txt")
        if os.path.exists(posible_ruta):
            etiqueta = sub
            break

    if etiqueta is None:
        print(f"No se encontró transcripción para: {archivo}")
        continue

    with open(os.path.join(carpeta_jsons, archivo), "r", encoding="utf-8") as f:
        data = json.load(f)

    frases_relevantes = []
    for f in data.get("frases", []):
        entidades = entidades_presentes(f["frase"])
        if entidades:
            f_mod = f.copy()
            f_mod["entidades_detectadas"] = entidades
            frases_relevantes.append(f_mod)

    if frases_relevantes:
        nuevo_json = {
            "etiqueta_original": etiqueta,
            "frases": frases_relevantes
        }
        ruta_salida = os.path.join(carpeta_salida_base, etiqueta, archivo)
        with open(ruta_salida, "w", encoding="utf-8") as out_f:
            json.dump(nuevo_json, out_f, indent=2, ensure_ascii=False)

print("Listo. JSONs filtrados guardados en 'resultados_filtrados/'.")
```

```
Listo. JSONs filtrados guardados en 'resultados_filtrados/'.
```

[12]:
```
# Cargar entidades útiles
df_entidades = pd.read_csv("entidades_top100_limpias.csv")
```

```
entidades_significativas = set(df_entidades["entidad"])  # ya están en␣
 ↪minúsculas

# Rutas
carpeta_jsons = "resultados_filtrados"
salida_csv = "train_significativo.csv"

# Preparar filas
filas = []

for subdir in ['izquierda', 'derecha']:
    label = 0 if subdir == 'izquierda' else 1
    ruta_subcarpeta = os.path.join(carpeta_jsons, subdir)

    for archivo in os.listdir(ruta_subcarpeta):
        if not archivo.endswith(".json"):
            continue

        ruta = os.path.join(ruta_subcarpeta, archivo)
        with open(ruta, "r", encoding="utf-8") as f:
            data = json.load(f)

        for frase in data.get("frases", []):
            entidades = frase.get("entidades_detectadas", [])
            if any(ent in entidades_significativas for ent in entidades):
                filas.append({
                    "archivo": archivo,
                    "texto": frase["frase"],
                    "label": label
                })

# Guardar CSV
df_final = pd.DataFrame(filas)
df_final.to_csv(salida_csv, index=False, encoding="utf-8")
print(f"Dataset generado con {len(df_final)} frases en {salida_csv}")
```

```
Dataset generado con 29083 frases en train_significativo.csv
```

4 PARTE 3: Fine-tuning de BERT

[2]:
```
# Verificar y mostrar dispositivo
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print(f"Usando dispositivo: {device}")

# Cargar tokenizer y modelo base
tokenizer = BertTokenizer.from_pretrained("dccuchile/
 ↪bert-base-spanish-wwm-cased")
```

```
model = BertForSequenceClassification.from_pretrained("dccuchile/
 ↪bert-base-spanish-wwm-cased", num_labels=2)

# Función para chunkear texto
def chunkear(texto, tokenizer, max_length=510):
    tokens = tokenizer.tokenize(texto)
    return [tokenizer.convert_tokens_to_string(tokens[i:i + max_length])
            for i in range(0, len(tokens), max_length)] or [texto]

# Leer CSV y generar chunks
df = pd.read_csv("train_significativo.csv")
data_rows = []
for _, row in df.iterrows():
    chunks = chunkear(row["texto"], tokenizer)
    for chunk in chunks:
        if chunk.strip():
            data_rows.append({"text": chunk, "label": row["label"]})

# Crear dataset y dividir
dataset = Dataset.from_dict({
    "text": [r["text"] for r in data_rows],
    "label": [r["label"] for r in data_rows]
})

def tokenize(batch):
    return tokenizer(batch["text"], padding="max_length", truncation=True,␣
 ↪max_length=512)

tokenized_dataset = dataset.map(tokenize, batched=True)
split = tokenized_dataset.train_test_split(test_size=0.2, seed=42)

# Configurar entrenamiento
training_args = TrainingArguments(
    output_dir="./bert_significativo",
    eval_strategy="epoch",
    save_strategy="epoch",
    learning_rate=2e-5,
    per_device_train_batch_size=8,
    per_device_eval_batch_size=8,
    num_train_epochs=3,
    weight_decay=0.01,
    load_best_model_at_end=True,
    metric_for_best_model="eval_loss"
)

def compute_metrics(eval_pred):
    logits, labels = eval_pred
```

```
    preds = logits.argmax(axis=1)
    precision, recall, f1, _ = precision_recall_fscore_support(labels, preds,␣
 ↪average='weighted')
    return {
        "accuracy": accuracy_score(labels, preds),
        "precision": precision,
        "recall": recall,
        "f1": f1
    }

# Entrenar (reanudar si hay checkpoint)
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=split["train"],
    eval_dataset=split["test"],
    tokenizer=tokenizer,
    compute_metrics=compute_metrics
)

trainer.train(resume_from_checkpoint=True)

# Guardar modelo fine-tuneado
model.save_pretrained("modelo_finetuneado_significativo")
tokenizer.save_pretrained("modelo_finetuneado_significativo")
```

```
Usando dispositivo: cuda
```

```
Some weights of BertForSequenceClassification were not initialized from the
model checkpoint at dccuchile/bert-base-spanish-wwm-cased and are newly
initialized: ['bert.pooler.dense.bias', 'bert.pooler.dense.weight',
'classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able to use it
for predictions and inference.
```

```
Map:   0%|          | 0/40800 [00:00<?, ? examples/s]
```

```
C:\Users\fedez\AppData\Local\Temp\ipykernel_10652\3883577624.py:62:
FutureWarning: `tokenizer` is deprecated and will be removed in version 5.0.0
for `Trainer.__init__`. Use `processing_class` instead.
  trainer = Trainer(
```

```
<IPython.core.display.HTML object>
```

[2]:
```
('modelo_finetuneado_significativo\\tokenizer_config.json',
 'modelo_finetuneado_significativo\\special_tokens_map.json',
 'modelo_finetuneado_significativo\\vocab.txt',
 'modelo_finetuneado_significativo\\added_tokens.json')
```

5 PARTE 4: Análisis de texto con el modelo fine-tuneado

[5]:
```
class AnalizadorIdeologico:
    def __init__(self, config):
        self.device = 'cuda' if torch.cuda.is_available() and config.
↪get('device') != 'cpu' else 'cpu'
        model_path = config['model_path']
        if not os.path.isdir(model_path) or not os.path.exists(os.path.
↪join(model_path, 'config.json')):
            raise FileNotFoundError(f"No se encontró el modelo BERT␣
↪fine-tuneado en {model_path}")

        self.tokenizer = BertTokenizer.from_pretrained(model_path)
        self.model_cls = BertForSequenceClassification.
↪from_pretrained(model_path).to(self.device)
        self.model_cls.eval()

    def _chunkear(self, texto, max_length=510):
        tokens = self.tokenizer.tokenize(texto)
        return [self.tokenizer.convert_tokens_to_string(tokens[i:i +␣
↪max_length])
                for i in range(0, len(tokens), max_length)] or [texto]

    def _predecir_chunk(self, chunk_text):
        try:
            inputs = self.tokenizer(chunk_text, max_length=512,␣
↪padding='max_length', truncation=True, return_tensors='pt')
            inputs = {k: v.to(self.device) for k, v in inputs.items()}
            with torch.no_grad():
                outputs = self.model_cls(**inputs)
                return torch.softmax(outputs.logits, dim=1)[0][1].item()
        except:
            return 0.5

    def analizar_frases(self, frases):
        frases_detalle = []

        for frase in frases:
            chunks = self._chunkear(frase["frase"])
            chunk_scores = [self._predecir_chunk(chunk) for chunk in chunks if␣
↪chunk.strip()]
            if chunk_scores:
                score_frase = sum(chunk_scores) / len(chunk_scores)
                etiqueta = "derecha" if score_frase > 0.5 else "izquierda"
                frases_detalle.append({
                    "frase": frase["frase"],
                    "score": round(score_frase, 3),
```

```
                    "etiqueta": etiqueta
                })

        if not frases_detalle:
            return None

        scores = [f["score"] for f in frases_detalle]
        conteo_izq = sum(1 for s in scores if s <= 0.5)
        conteo_der = len(scores) - conteo_izq
        score_promedio = sum(scores) / len(scores)
        etiqueta_final = "derecha" if score_promedio > 0.5 else "izquierda"

        return {
            "perfil_global_del_texto": {
                "conteo": {"izquierda": conteo_izq, "derecha": conteo_der},
                "score_promedio": round(score_promedio, 3),
                "etiqueta_final": etiqueta_final
            },
            "frases": frases_detalle,
            "detalles": {"frases_analizadas": len(frases_detalle)}
        }
```

[6]:
```
# Cargar clases necesarias
analizador = AnalizadorIdeologico({"model_path":␣
 ↪"modelo_finetuneado_significativo"})

# Directorios
input_dir = "resultados_filtrados/neutral"
output_base = "resultados_significativos"
Path(os.path.join(output_base, "izquierda")).mkdir(parents=True, exist_ok=True)
Path(os.path.join(output_base, "derecha")).mkdir(parents=True, exist_ok=True)

# Procesar cada archivo
for file in os.listdir(input_dir):
    if not file.endswith(".json"):
        continue

    with open(os.path.join(input_dir, file), "r", encoding="utf-8") as f:
        data = json.load(f)

    frases = data.get("frases", [])
    if not frases:
        continue

    resultado = analizador.analizar_frases(frases)
    if resultado is None:
        continue
```

```
    resultado["etiqueta_original"] = data.get("etiqueta_original", "")
    output_path = os.path.join(output_base,␣
 ↪resultado["perfil_global_del_texto"]["etiqueta_final"], file)

    with open(output_path, "w", encoding="utf-8") as f_out:
        json.dump(resultado, f_out, ensure_ascii=False, indent=2)

print("Clasificación completada. JSONs guardados en resultados_significativos/
 ↪izquierda o /derecha.")
```

```
Clasificación completada. JSONs guardados en resultados_significativos/izquierda
o /derecha.
```