```python
# preprocesamiento.py
# Este script se encarga de generar documentos de texto a partir de plantillas,
# sustituyendo los marcadores de posición (placeholders) por datos específicos
# cargados desde archivos JSON. Vamos a generar 20,000 documentos para tener
# suficientes datos de facturas para entrenar nuestro modelo de SPACY.

import os
import json
import random

# Directorios de entrada y salida...
# Cargamos las plantillas que hemos generado y donde tenemos las etiquetas
llamadas placeholders.
# Estos placeholders serán sustituidos por los datos de los archivos JSON.
# Así, podemos generar una cantidad infinita de plantillas con los datos
disponibles.
plantillas_dir = "C:/Users/34670/Desktop/python/Hack a
boss/proyecto_decide/libretas/plantillas/"
json_dir = "C:/Users/34670/Desktop/python/Hack a
boss/proyecto_decide/libretas/json_categoria/"
output_dir = "C:/Users/34670/Desktop/python/Hack a
boss/proyecto_decide/libretas/datos/"

# Diccionario de placeholders...
# Este diccionario mapea los placeholders a sus correspondientes archivos JSON.
json_files = {
    "placeholder_01": "nombre_cliente",
    "placeholder_02": "dni_cliente",
    "placeholder_03": "calle_cliente",
    "placeholder_04": "cp_cliente",
    "placeholder_05": "poblacion_cliente",
    "placeholder_06": "provincia_cliente",
    "placeholder_07": "nombre_comercializadora",
    "placeholder_08": "cif_comercializadora",
    "placeholder_09": "direccion_comercializadora",
    "placeholder_10": "cp_comercializadora",
    "placeholder_11": "poblacion_comercializadora",
    "placeholder_12": "provincia_comercializadora",
    "placeholder_13": "numero_factura",
    "placeholder_14": "inicio_periodo",
    "placeholder_15": "fin_periodo",
    "placeholder_16": "importe_factura",
    "placeholder_17": "fecha_cargo",
    "placeholder_18": "consumo_periodo",
    "placeholder_19": "potencia_contratada",
}

# Función para cargar datos de JSON...
# Aquí cargamos los datos desde los archivos JSON en un diccionario.
def cargar_datos_json(json_dir, json_files):
    datos = {}
    for placeholder, key in json_files.items():
```

```
            file_name = f"{key}.json"
            with open(os.path.join(json_dir, file_name), 'r', encoding='utf-8') as f:
                datos[placeholder] = json.load(f)
    return datos

# Función para generar un texto con datos aleatorios y capturar las entidades...
# Sustituimos los placeholders en la plantilla por datos aleatorios del JSON
correspondiente
# y capturamos la posición de las entidades para usarlas en el entrenamiento del
modelo...
def generar_texto_con_datos(plantilla, datos_json):
    entities = []
    texto_final = ""
    offset = 0

    for placeholder, key in json_files.items():
        while placeholder in plantilla:
            valor = random.choice(datos_json[placeholder])
            start = plantilla.find(placeholder)
            end = start + len(valor)

            texto_final += plantilla[:start] + valor
            plantilla = plantilla[start + len(placeholder):]

            # Ajuste de entidades y desplazamiento de posiciones
            for entity in entities:
                if entity[0] >= offset + start:
                    entity[0] += len(valor) - len(placeholder)
                if entity[1] >= offset + start:
                    entity[1] += len(valor) - len(placeholder)

            entities.append([offset + start, offset + start + len(valor), key])
            offset += start + len(valor)

    texto_final += plantilla

    return texto_final, entities

# Función principal para generar documentos...
# Esta función coordina la carga de datos, selección de plantillas, generación de
textos y escritura en archivos.
def generar_documentos(plantillas_dir, json_dir, output_dir, json_files,
num_docs=20000):
    # Cargar datos JSON...
    datos_json = cargar_datos_json(json_dir, json_files)

    # Obtener plantillas...
    plantillas = [f for f in os.listdir(plantillas_dir) if f.endswith('.txt')]

    # Generar documentos...
    documentos_descartados = 0
    for i in range(num_docs):
        plantilla_file = random.choice(plantillas)
        with open(os.path.join(plantillas_dir, plantilla_file), 'r',
```

```
encoding='utf-8') as f:
            plantilla = f.read()

        texto, entities = generar_texto_con_datos(plantilla, datos_json)

        if not entities:
            documentos_descartados += 1
            continue

        data_entry = {
            "text": texto,
            "entities": entities
        }

        output_file = os.path.join(output_dir, f"documento_{i+1}.json")
        with open(output_file, 'w', encoding='utf-8') as f:
            json.dump(data_entry, f, ensure_ascii=False, indent=4)

    print(f"Total de documentos descartados: {documentos_descartados}")

# Ejecutar la función principal...
# Creamos el directorio de salida si no existe y llamamos a la función para
generar los documentos.
if not os.path.exists(output_dir):
    os.makedirs(output_dir)

generar_documentos(plantillas_dir, json_dir, output_dir, json_files,
num_docs=20000)



_____
_____



# entrenamiento.py
# Este script procesa documentos de texto y sus entidades para crear archivos
binarios
# que pueden ser utilizados para entrenar y validar un modelo de spaCy en
español...
# Usamos plantillas generadas previamente y etiquetas para entrenar el modelo...

import os
import json
import spacy
from spacy.tokens import DocBin
import random

# Ruta a la carpeta de los textos extraídos y los JSON con las etiquetas...
data_folder = 'C:/Users/34670/Desktop/python/Hack a
boss/proyecto_decide/libretas/datos/'

# Inicializar spaCy...
# Usar un modelo en blanco para español, que entrenaremos desde cero con nuestros
```

```
datos.
nlp = spacy.blank('es')

# Recopilar y procesar todos los archivos JSON en el directorio de datos...
all_files = [file_name for file_name in os.listdir(data_folder) if
file_name.endswith('.json')]

# Barajar los archivos para una distribución aleatoria...
random.shuffle(all_files)

# Dividir los archivos en conjuntos de entrenamiento y validación (80%-20%)...
split_index = int(len(all_files) * 0.8)
train_files = all_files[:split_index]
val_files = all_files[split_index:]

# Función para procesar archivos y crear DocBin...
# DocBin es una estructura de datos eficiente para almacenar múltiples objetos
Doc, que representa nuestros documentos procesados con spaCy.
# Procesa cada archivo, crea el objeto Doc de spaCy y añade las entidades.
# Los documentos inválidos son descartados y se cuentan... así podemos evaluar si
hay muchos errores en el procesamiento del entrenamiento.
def process_files(file_list, docbin):
    discarded_count = 0
    for file_name in file_list:
        file_path = os.path.join(data_folder, file_name)

        with open(file_path, 'r', encoding='utf-8') as f:
            data = json.load(f)

        text = data['text']
        entities = data['entities']

        # Crear ejemplo de spaCy...
        doc = nlp.make_doc(text)
        spans = []
        valid = True
        for start, end, label in entities:
            if start < len(doc.text) and end <= len(doc.text):
                entity_text = text[start:end]
                # Verificar que el texto de la entidad coincida con lo que debería
ser...
                if entity_text == text[start:end]:
                    span = doc.char_span(start, end, label=label)
                    if span is not None:
                        spans.append(span)
                    else:
                        valid = False
                        print(f"Entidad inválida: {label} ({start}-{end}) en el
archivo {file_name}, texto: '{entity_text}'")
                        break
                else:
                    valid = False
                    print(f"Texto de entidad no coincide: {label} ({start}-{end})
en el archivo {file_name}, texto entidad: '{entity_text}', texto real:
```

```
'{text[start:end]}'")
                    break
            else:
                valid = False
                print(f"Índice fuera de rango: {label} ({start}-{end}) en el
archivo {file_name}, longitud del texto: {len(doc.text)}, texto:
'{text[start:end]}'")
                break

        if valid:
            doc.ents = spans
            docbin.add(doc)
        else:
            discarded_count += 1
            print(f"Documento inválido descartado: {file_name}")

    return discarded_count

# Crear DocBin para entrenamiento y validación...
train_db = DocBin()
val_db = DocBin()

# Procesar archivos de entrenamiento...
train_discarded = process_files(train_files, train_db)
# Procesar archivos de validación...
val_discarded = process_files(val_files, val_db)

# Guardar los datos de entrenamiento y validación en formato binario de spaCy
train_output_path = 'train_data.spacy'
val_output_path = 'val_data.spacy'
train_db.to_disk(train_output_path)
val_db.to_disk(val_output_path)
print(f"Datos de entrenamiento guardados en '{train_output_path}'")
print(f"Datos de validación guardados en '{val_output_path}'")

# Imprimir resumen de archivos descartados. Así vemos si hay muchos errores a la
hora del entrenamiento al montar los archivos del DocBin
print(f"Archivos descartados durante el entrenamiento: {train_discarded}")
print(f"Archivos descartados durante la validación: {val_discarded}")


_____
_____


# modelo.py
# Este script utiliza spaCy para entrenar un modelo de reconocimiento de entidades
(NER).
# El objetivo es extraer automáticamente campos específicos de facturas eléctricas
en PDF.
# Dado que las facturas pueden variar en formato y disposición de los campos,
necesitamos
# un método genérico que pueda manejar distintos tipos de plantillas de facturas y
extraer
```

```
# la información necesaria de manera consistente.

import spacy
from spacy.tokens import DocBin
from spacy.training import Example
from sklearn.metrics import classification_report
from spacy.util import compounding, minibatch
from spacy.lookups import load_lookups

# Cargar los datos de entrenamiento y validación
train_data_path = 'train_data.spacy'
val_data_path = 'val_data.spacy'

# Función para cargar datos binarios...
# Cargamos los datos de entrenamiento y validación desde archivos binarios de
spaCy (DocBin).
def load_data(data_path, nlp):
    doc_bin = DocBin().from_disk(data_path)
    return list(doc_bin.get_docs(nlp.vocab))

# Inicializar spaCy con un modelo preentrenado en español...
# Usamos 'es_core_news_md' que es un modelo de tamaño medio para probar su
eficacia.
# Existe también el spacy.load('es_core_news_lg') para modelo grande
# y también el spacy.load('es_core_news_sm') para un modelo más pequeño.
nlp = spacy.load('es_core_news_md')

# Eliminar el componente 'matcher' si existe en el pipeline... me ha dado varias
veces error
# y eliminándolo se soluciona el problema y se ejecuta el modelo.
if 'matcher' in nlp.pipe_names:
    nlp.remove_pipe('matcher')

# Cargar las tablas de lemas necesarias para el lematizador...
# Esto mejora la precisión al normalizar palabras a su forma base.
lookups = load_lookups(lang="es", tables=["lemma_rules", "lemma_index",
"lemma_exc", "lemma_rules_groups"])
nlp.get_pipe("lemmatizer").initialize(nlp.vocab, lookups=lookups)

# Cargar los datos de entrenamiento y validación...
train_data = load_data(train_data_path, nlp)
val_data = load_data(val_data_path, nlp)

# Añadir el componente NER al pipeline si no existe...
if 'ner' not in nlp.pipe_names:
    ner = nlp.add_pipe('ner', last=True)
else:
    ner = nlp.get_pipe('ner')

# Añadir las etiquetas al componente NER...
# Aquí añadimos las etiquetas de las entidades que queremos que el modelo
reconozca.
for doc in train_data:
    for ent in doc.ents:
```

```
        ner.add_label(ent.label_)

# Configurar los parámetros de entrenamiento...
# Ajustamos la tasa de aprendizaje para mejorar la convergencia del modelo.
optimizer = nlp.create_optimizer()
optimizer.learn_rate = 0.0005 # He probado con varias... con 0.001, con 0.0001...

# Función para evaluar el modelo...
# Esta función evalúa la precisión del modelo comparando las entidades predichas
con las reales.
# Esto es importante porque bueno, una de las veces estuve más de 4 horas
esperando que el modelo terminara
# de entrenar y luego dio error en las métricas.
def evaluate_model(nlp, data):
    y_true = []
    y_pred = []
    labels = list(nlp.get_pipe("ner").labels)
    for doc in data:
        gold_entities = [(ent.start_char, ent.end_char, ent.label_) for ent in
doc.ents]
        pred_doc = nlp(doc.text)
        pred_entities = [(ent.start_char, ent.end_char, ent.label_) for ent in
pred_doc.ents]

        y_true.extend([label for _, _, label in gold_entities])
        y_pred.extend([label for _, _, label in pred_entities])

    # Asegurar que y_true y y_pred tengan la misma longitud ¡Importante!
    min_length = min(len(y_true), len(y_pred))
    y_true = y_true[:min_length]
    y_pred = y_pred[:min_length]

    report = classification_report(y_true, y_pred, labels=labels, zero_division=0)
    return report

# Entrenar el modelo...
# Aumentamos el número de iteraciones para asegurar una mejor convergencia. He
probado con 10, 20... 100...
n_iter = 50
# El evaluate_every en 1 lo he puesto para saber rápidamente si las evaluaciones
las hacía bien y que no hubiese ningún error.
# Quizás es más interesante poner 10 psi vas a hacer 50 iteraciones para ir viendo
la evaluación o 5 si haces 20 iteraciones...
evaluate_every = 1
best_f1_score = 0.0
patience = 8 # esto es para que el modelo se detenga si no mejora... así no hay
que esperar hasta 50 si ya en el 30 ve que no mejora
no_improvement_counter = 0

for itn in range(n_iter):
    losses = {}
    # Ajustamos el tamaño del batch para mejorar el rendimiento del entrenamiento.
    batches = minibatch(train_data, size=compounding(4.0, 32.0, 1.001))
    for batch in batches:
```

```
        examples = []
        for doc in batch:
            example = Example.from_dict(doc, {"entities": [(ent.start_char,
ent.end_char, ent.label_) for ent in doc.ents]})
            examples.append(example)
        # Ajustamos el dropout para prevenir sobreajuste. Igual, he probado con
0.3, 0.5...
        nlp.update(examples, sgd=optimizer, drop=0.4, losses=losses)

    print(f"Iteración {itn + 1}, Pérdidas: {losses}")

    # Evaluar el modelo cada iteración...
    report = evaluate_model(nlp, val_data)
    print(f"Reporte de clasificación para la iteración {itn + 1}:\n{report}")

    # Medir F1-score y comparar con el mejor
    lines = report.split('\n')
    avg_line = [line for line in lines if 'avg' in line.lower()][0]
    current_f1_score = float(avg_line.split()[-2])
    if current_f1_score > best_f1_score:
        best_f1_score = current_f1_score
        no_improvement_counter = 0
        # Guardar el mejor modelo
        nlp.to_disk('best_model')
    else:
        no_improvement_counter += 1

    if no_improvement_counter >= patience:
        print("Early stopping debido a la falta de mejora en el rendimiento.")
        break

# Guardar el modelo entrenado final
output_dir = 'modelo_entrenado'
nlp.to_disk(output_dir)
print(f"Modelo guardado en '{output_dir}'")

# Evaluar el modelo final
nlp_trained = spacy.load(output_dir)
final_report = evaluate_model(nlp_trained, val_data)
print(f"Reporte de clasificación final:\n{final_report}")
```

```
# validacion.py
# Este script se utiliza para evaluar un modelo de reconocimiento de entidades
(NER) entrenado con spaCy.
# Cargará el modelo que ya hemos entrenado y los datos de validación para evaluar
el rendimiento del modelo.

import spacy
from spacy.tokens import DocBin
```

```
from sklearn.metrics import classification_report

# Función para cargar datos binarios...
# Cargamos los datos de validación desde un archivo binario de spaCy (DocBin).
def load_data(data_path, nlp):
    doc_bin = DocBin().from_disk(data_path)
    return list(doc_bin.get_docs(nlp.vocab))

# Inicializar spaCy y cargar el modelo entrenado...
# Asegúrate de que el directorio 'modelo_entrenado' contenga tu modelo entrenado.
output_dir = 'modelo_entrenado'
nlp = spacy.load(output_dir)

# Cargar los datos de validación
val_data_path = 'val_data.spacy'
val_data = load_data(val_data_path, nlp)

# Función para evaluar el modelo...
# Esta función evalúa la precisión del modelo comparando las entidades predichas
con las reales.
def evaluate_model(nlp, data):
    y_true = []
    y_pred = []
    labels = list(nlp.get_pipe("ner").labels)
    for doc in data:
        gold_entities = [(ent.start_char, ent.end_char, ent.label_) for ent in
doc.ents]
        pred_doc = nlp(doc.text)
        pred_entities = [(ent.start_char, ent.end_char, ent.label_) for ent in
pred_doc.ents]

        y_true.extend([label for _, _, label in gold_entities])
        y_pred.extend([label for _, _, label in pred_entities])

    # Asegurar que y_true y y_pred tengan la misma longitud ¡Importante!
    min_length = min(len(y_true), len(y_pred))
    y_true = y_true[:min_length]
    y_pred = y_pred[:min_length]

    report = classification_report(y_true, y_pred, labels=labels, zero_division=0)
    return report

# Evaluar el modelo final
final_report = evaluate_model(nlp, val_data)
print(f"Reporte de clasificación final:\n{final_report}")



_____
_____


# analizar_texto.py
# Este script permite ingresar texto por terminal y devuelve las entidades
reconocidas
```

```
# por el modelo de reconocimiento de entidades (NER) entrenado con spaCy.
# Esto lo hago para resolver una primera duda: ¿qué tan bien funciona el modelo si
le
# devuelvo un texto que yo escribo introduciendo datos manualmente?

import spacy

# Cargar el modelo entrenado...
# Asegúrate de que el directorio 'modelo_entrenado' contenga tu modelo entrenado.
output_dir = 'modelo_entrenado'
nlp = spacy.load(output_dir)

# Lista de categorías de entidades que queremos encontrar...
# Estas son las etiquetas que esperamos que el modelo reconozca en el texto.
categorias_entidades = [
    "nombre_cliente", "dni_cliente", "calle_cliente", "cp_cliente",
    "poblacion_cliente", "provincia_cliente", "nombre_comercializadora",
    "cif_comercializadora", "direccion_comercializadora", "cp_comercializadora",
    "poblacion_comercializadora", "provincia_comercializadora", "numero_factura",
    "inicio_periodo", "fin_periodo", "importe_factura", "fecha_cargo",
    "consumo_periodo", "potencia_contratada"
]

# Función para analizar el texto ingresado por terminal...
# Procesa el texto con el modelo spaCy y extrae las entidades reconocidas.
def analizar_texto(texto):
    doc = nlp(texto)
    entidades_encontradas = []
    for ent in doc.ents:
        if ent.label_ in categorias_entidades:
            entidades_encontradas.append((ent.text, ent.label_))
    return entidades_encontradas

# Ingresar texto por terminal...
# Permite al usuario introducir un texto y analiza las entidades presentes en él.
if __name__ == "__main__":
    texto_usuario = input("Introduce el texto a analizar: ")
    entidades = analizar_texto(texto_usuario)
    if entidades:
        print("Entidades reconocidas:")
        for entidad, etiqueta in entidades:
            print(f"{etiqueta}: {entidad}")
    else:
        print("No se encontraron entidades reconocidas.")
```

```
# extraccion_pdf.py
# Este script se utiliza para extraer texto de archivos PDF de facturas y
```

```
limpiarlo.
# Una vez entrenado el modelo y confirmado que reconoce las entidades
correctamente,
# procedemos con la extracción de los datos de las facturas originales.
# Vamos a utilizar la librería fitz (PyMuPDF) que parece dar buen rendimiento...

import fitz  # PyMuPDF
import os
import re
from dateutil.parser import parse, ParserError

# Función para extraer texto de un PDF usando PyMuPDF (fitz)
# Abrimos el PDF y extraemos el texto de cada página.
def extract_text_from_pdf(pdf_path):
    doc = fitz.open(pdf_path)
    text = ''
    for page_num in range(len(doc)):
        page = doc.load_page(page_num)
        text += page.get_text("text")
    return text

# Función para normalizar fechas
def normalize_dates(text):
    # Expresión regular para encontrar fechas en varios formatos
    date_patterns = [
        (r'\b(\d{1,2})[\/\-\.](\d{1,2})[\/\-\.](\d{2,4})\b', "%d/%m/%Y"),  #
Formatos: 15/09/2024, 15-01-21, 15.01.21
        (r'\b(\d{2,4})[\/\-\.](\d{1,2})[\/\-\.](\d{1,2})\b', "%d/%m/%Y"),  #
Formato: 2024/09/17, 2024-09-17, 2024.09.17
        (r'\b(\d{8})\b', "%d/%m/%Y"),                                      #
Formato: 27092018
        (r'\b(\d{1,2})\s+de\s+(\w+)\s+de\s+(\d{4})\b', "%d/%m/%Y")         #
Formato: 2 de octubre de 1991
    ]

    def replace_date(match, date_format):
        date_str = match.group()
        try:
            # Parse the date and format it as dd/mm/yyyy
            parsed_date = parse(date_str, dayfirst=True)
            return parsed_date.strftime(date_format)
        except (ParserError, ValueError):
            return date_str

    for pattern, date_format in date_patterns:
        text = re.sub(pattern, lambda match: replace_date(match, date_format),
text, flags=re.IGNORECASE)

    return text

# Función para limpiar el texto
# Aplicamos varias reglas de limpieza para preparar el texto extraído.
def clean_text(text):
    text = re.sub(r'http\S+|www\.\S+', '', text)  # Eliminar URLs
```

```
    text = re.sub(r'\.{2,}', '.', text)  # Eliminar secuencias repetidas de puntos
    text = re.sub(r'\s+', ' ', text).strip()  # Eliminar secuencias repetidas de
espacios
    text = re.sub(r'\bx,xx\b', '', text)  # Eliminar valores placeholder x,xx
    text = re.sub(r'\bpágina \d+\b', '', text)  # Eliminar números de página
    text = re.sub(r'(?<=\s)[\.\,](?=\s)', '', text)  # Eliminar puntos y comas
solitarios
    text = re.sub(r'[^\w\s.,€|-]', '', text)  # Eliminar caracteres no deseados,
mantener números, letras, puntos, comas, € y guiones, y el símbolo |
    return text

# Ruta a la carpeta de los PDFs y la carpeta de salida para los textos extraídos
pdf_folder = 'C:/Users/34670/Desktop/python/Hack a boss/proyecto_decide/training'
output_folder = 'C:/Users/34670/Desktop/python/Hack a
boss/proyecto_decide/libretas/facturas'
os.makedirs(output_folder, exist_ok=True)

# Procesar todos los PDFs en la carpeta
# Iteramos sobre todos los archivos PDF en la carpeta especificada.
for pdf_file in os.listdir(pdf_folder):
    if not pdf_file.endswith('.pdf'):
        continue

    pdf_path = os.path.join(pdf_folder, pdf_file)

    # 1. Extraer texto del PDF
    text = extract_text_from_pdf(pdf_path)

    # 2. Reemplazar saltos de línea con el símbolo |
    text = text.replace('\n', ' | ')

    # 3. Limpiar el texto
    text = clean_text(text)

    # 4. Normalizar las fechas en el texto
    text = normalize_dates(text)

    # Guardar el texto limpiado en un archivo
    output_file = os.path.join(output_folder, pdf_file.replace('.pdf', '.txt'))
    with open(output_file, 'w', encoding='utf-8') as f:
        f.write(text)

    print(f"Texto extraído y limpiado de '{pdf_file}' guardado en
'{output_file}'")

print("Procesamiento completado.")


_____
_____


# app.py
# Este será el archivo principal que configuraremos para realizar toda la
```

```
operación de extracción de datos de las facturas.
# Utilizaremos este script para cargar el modelo entrenado de spaCy, aplicar el
modelo a las facturas originales extraídas,
# detectar patrones específicos en el texto, y guardar los resultados en archivos
JSON.

import spacy
import os
import json
import re
from dateutil.parser import parse, ParserError

# Ruta del modelo entrenado
model_path = 'modelo_entrenado'

# Cargar el modelo entrenado
nlp = spacy.load(model_path)

# Directorios
input_dir = 'C:/Users/34670/Desktop/python/Hack a
boss/proyecto_decide/libretas/facturas/'
output_dir = 'C:/Users/34670/Desktop/python/Hack a
boss/proyecto_decide/libretas/validaciones/'

# Asegurarse de que el directorio de salida existe
os.makedirs(output_dir, exist_ok=True)

# Categorías de entidades
# Estas son las etiquetas que el modelo debe reconocer y extraer del texto de las
facturas
categories = [
    "nombre_cliente", "dni_cliente", "calle_cliente", "cp_cliente",
"población_cliente",
    "provincia_cliente", "nombre_comercializadora", "cif_comercializadora",
"dirección_comercializadora",
    "cp_comercializadora", "población_comercializadora",
"provincia_comercializadora", "número_factura",
    "inicio_periodo", "fin_periodo", "importe_factura", "fecha_cargo",
"consumo_periodo", "potencia_contratada"
]

# Para facilitar la ayuda al modelo, podemos de alguna forma "seleccionar" que
datos extraer de los textos
# que sean más reconocibles mediante expresiones regulares u otras.
# Lista de provincias españolas con sus variaciones
provincias_espanolas = [
    "Álava", "Araba", "Albacete", "Alicante", "Alacant", "Almería", "Asturias",
"Ávila", "Badajoz",
    "Baleares", "Illes Balears", "Barcelona", "Burgos", "Cáceres", "Cádiz",
"Cantabria", "Castellón",
    "Castelló", "Ciudad Real", "Córdoba", "Cuenca", "Gerona", "Girona", "Granada",
"Guadalajara",
    "Guipúzcoa", "Gipuzkoa", "Huelva", "Huesca", "Jaén", "La Coruña", "A Coruña",
"La Rioja", "Las Palmas",
```

```
    "León", "Lleida", "Lugo", "Madrid", "Málaga", "Murcia", "Navarra", "Nafarroa",
"Orense", "Ourense",
    "Palencia", "Pontevedra", "Salamanca", "Santa Cruz de Tenerife", "Segovia",
"Sevilla", "Soria",
    "Tarragona", "Teruel", "Toledo", "Valencia", "Valladolid", "Vizcaya",
"Bizkaia", "Zamora", "Zaragoza"
]

# Funciones para detectar patrones en el texto
def detect_dni(text):
    match = re.search(r'\b\d{8}[A-Z]\b', text, re.IGNORECASE)
    return match.group() if match else ""

def detect_cp(text):
    match = re.search(r'\b\d{5}\b', text)
    return match.group() if match else ""

def detect_nombre_cliente(text):
    # Eliminar ocurrencias de NIF y DNI antes de buscar el nombre
    text = re.sub(r'\bNIF\s*\d{8}[A-Z]\b', '', text, flags=re.IGNORECASE)
    text = re.sub(r'\bDNI\s*\d{8}[A-Z]\b', '', text, flags=re.IGNORECASE)
    # Expresión regular para nombres españoles con soporte para mayúsculas,
minúsculas y acentos, y sin números ni caracteres no válidos
    pattern = r'\b[A-ZÁÉÍÓÚÑ][a-záéíóúñ]+ [A-ZÁÉÍÓÚÑ][a-záéíóúñ]+ [A-ZÁÉÍÓÚÑ][a-
záéíóúñ]+\b|\b[A-ZÁÉÍÓÚÑ][a-záéíóúñ]+ [a-záéíóúñ]+ [A-ZÁÉÍÓÚÑáéíóúñ]+\b'
    matches = re.findall(pattern, text)
    for match in matches:
        if not re.search(r'\d', match):
            # Verificar que no haya letras sueltas separadas por espacios, puntos
o comas
            if not re.search(r'([A-Za-zÁÉÍÓÚÑáéíóúñ]\s{1}[A-Za-zÁÉÍÓÚÑáéíóúñ])|
([A-Za-zÁÉÍÓÚÑáéíóúñ]\.{1}[A-Za-zÁÉÍÓÚÑáéíóúñ])|([A-Za-zÁÉÍÓÚÑáéíóúñ],{1}[A-Za-
zÁÉÍÓÚÑáéíóúñ])', match):
                return match
    return ""

# Aquí para ayudar a detectar las provincias
def detect_provincia(text):
    for provincia in provincias_espanolas:
        if re.search(r'\b' + re.escape(provincia.lower()) + r'\b', text.lower()):
            return provincia
    return ""

# Aquí introducimos el patrón que siguen más o menos las facturas
def detect_numero_factura(text):
    match = re.search(r'\b[A-Z0-9]{10,13}\b', text, re.IGNORECASE)
    return match.group() if match else ""

# Aquí para la potencia contratada
def detect_potencia(text):
    match = re.search(r'\b\d{1,2},\d{3}\b', text)
    return match.group() if match else ""

# Esto es para el importe de las facturas...
```

```
def detect_importe(text):
    match = re.search(r'\b\d{1,3},\d{2}\b', text)
    return match.group() if match else ""

# Aquí para detectar fechas
def detect_fecha(text):
    try:
        parsed_date = parse(text, dayfirst=True)
        return parsed_date.strftime("%d.%m.%Y")
    except (ParserError, ValueError):
        return ""

# Este es para la detección del consumo, ya que hay muchos números enteros, le
ayudamos con el kWh
def detect_consumo(text):
    match = re.search(r'\b(\d{1,4}) kWh\b', text, re.IGNORECASE)
    return match.group(1) if match else ""

# Aquí para detectar direcciones
def detect_direccion(text):
    pattern = (

r'\b(?:C\/|Calle|Avenida|Avda\.|Av\.|Plaza|Paseo|Pje\.|Pl\.|Parque|Camino|Carreter
a|Cami|Urb\.)\s+[\w\s]+(?:,\s*\d+|\s+\d+)?\s*(?:[-ºªA-Za-z0-9\s,]*)?'
    )
    match = re.search(pattern, text, re.IGNORECASE)
    return match.group() if match else ""

# Función para limpiar el nombre de cliente, algunas veces se cuela la palabra NIF
def clean_nombre_cliente(nombre):
    # Eliminar cualquier ocurrencia de NIF o DNI y lo que venga después
    nombre = re.sub(r'\bNIF\s*\d{8}[A-Z]\b', '', nombre, flags=re.IGNORECASE)
    nombre = re.sub(r'\bDNI\s*\d{8}[A-Z]\b', '', nombre, flags=re.IGNORECASE)
    # Eliminar caracteres no deseados
    nombre = re.sub(r'[^\w\sÁÉÍÓÚÑáéíóúñ]', '', nombre)
    # Eliminar espacios extra
    nombre = re.sub(r'\s+', ' ', nombre).strip()
    return nombre

# Función para validar y ajustar entidades según patrones conocidos
def validate_and_adjust_entities(entities, text):
    # Detectar patrones en el texto
    detected_entities = {
        "dni_cliente": detect_dni(text),
        "cp_cliente": detect_cp(text),
        "nombre_cliente": detect_nombre_cliente(text),
        "provincia_cliente": detect_provincia(text),
        "provincia_comercializadora": detect_provincia(text),
        "número_factura": detect_numero_factura(text),
        "potencia_contratada": detect_potencia(text),
        "importe_factura": detect_importe(text),
        "inicio_periodo": detect_fecha(text),
        "fin_periodo": detect_fecha(text),
        "fecha_cargo": detect_fecha(text),
```

```
        "consumo_periodo": detect_consumo(text),
        "dirección_comercializadora": detect_direccion(text)
    }

    # Ajustar entidades extraídas con las detectadas por patrones
    for key in detected_entities:
        if not entities[key] and detected_entities[key]:
            entities[key] = detected_entities[key]

    # Validar y ajustar el CP de la comercializadora... para que no coja el del
cliente
    if detected_entities["dirección_comercializadora"]:
        context_text =
text[text.find(detected_entities["dirección_comercializadora"]):]
        cp_comercializadora = detect_cp(context_text)
        if cp_comercializadora:
            entities["cp_comercializadora"] = cp_comercializadora

    # Limpiar el nombre del cliente
    entities["nombre_cliente"] = clean_nombre_cliente(entities["nombre_cliente"])

    return entities

# Función para procesar el texto y extraer las entities...
# Esta función utiliza el modelo spaCy para extraer entidades específicas del
texto de las facturas
def extract_entities(text):
    doc = nlp(text)
    segments = text.split(' | ')
    entities = {category: "" for category in categories}

    for segment in segments:
        segment_doc = nlp(segment)
        for ent in segment_doc.ents:
            if ent.label_ in categories and not entities[ent.label_]:
                entities[ent.label_] = ent.text

    # Validar y ajustar las entities extraídas
    entities = validate_and_adjust_entities(entities, text)

    return entities

# Procesamos cada archivo en el directorio de entrada...
# Iteramos sobre todos los archivos en el directorio de entrada, leemos el texto,
# aplicamos el modelo para extraer entidades y guardamos los resultados en
archivos JSON.
for filename in os.listdir(input_dir):
    if filename.endswith('.txt') or filename.endswith('.json'):
        file_path = os.path.join(input_dir, filename)
        with open(file_path, 'r', encoding='utf-8') as file:
            if filename.endswith('.json'):
                data = json.load(file)
                text = data.get("text", "")
            else:
```

```
                text = file.read()

        # Extraer entidades del texto...
        extracted_entities = extract_entities(text)

        # Guardar los resultados en un archivo JSON
        output_filename = os.path.splitext(filename)[0] + '_result.json'
        output_path = os.path.join(output_dir, output_filename)
        with open(output_path, 'w', encoding='utf-8') as output_file:
            json.dump(extracted_entities, output_file, ensure_ascii=False,
indent=4)

        print(f"Resultados guardados en {output_path}")
```

```
# medicion.py
# Este script se utiliza para medir la precisión de nuestro modelo de extracción
de entidades (NER).
# Compara los resultados extraídos por el modelo con los datos originales y
calcula la tasa de acierto
# para cada categoría de entidad y una tasa de acierto global...

import os
import json
from dateutil.parser import parse

# Directorios
original_dir = 'C:/Users/34670/Desktop/python/Hack a
boss/proyecto_decide/training/'
extracted_dir = 'C:/Users/34670/Desktop/python/Hack a
boss/proyecto_decide/libretas/validaciones/'

# Categorías de entidades
# Estas son las etiquetas que esperamos que el modelo haya reconocido
correctamente en los textos.
categories = [
    "nombre_cliente", "dni_cliente", "calle_cliente", "cp_cliente",
"población_cliente",
    "provincia_cliente", "nombre_comercializadora", "cif_comercializadora",
"dirección_comercializadora",
    "cp_comercializadora", "población_comercializadora",
"provincia_comercializadora", "número_factura",
    "inicio_periodo", "fin_periodo", "importe_factura", "fecha_cargo",
"consumo_periodo", "potencia_contratada"
]

# Función para cargar JSON
# Cargamos los datos de un archivo JSON y los devolvemos como un diccionario.
def load_json(file_path):
```

```
    with open(file_path, 'r', encoding='utf-8') as file:
        return json.load(file)

# Función para normalizar fechas al formato dd.mm.yyyy
def normalize_date(date_str):
    try:
        date_obj = parse(date_str, dayfirst=True)
        return date_obj.strftime("%d.%m.%Y")
    except (ValueError, OverflowError):
        return date_str

# Función para comparar dos diccionarios y calcular la tasa de acierto
# Compara los valores de las entidades en los diccionarios original y extraído, y
cuenta los aciertos.
def compare_dicts(original, extracted):
    total = 0
    correct = 0
    for category in categories:
        total += 1
        original_value = str(original.get(category, "")).strip()
        extracted_value = str(extracted.get(category, "")).strip()

        # Normalizar fechas si el campo es una fecha
        if "fecha" in category or "periodo" in category:
            original_value = normalize_date(original_value)
            extracted_value = normalize_date(extracted_value)

        if original_value == extracted_value:
            correct += 1
    return correct, total

# Inicializar contadores
# Estos contadores llevarán un registro de los aciertos por categoría y
globalmente.
total_correct = {category: 0 for category in categories}
total_entries = {category: 0 for category in categories}
global_correct = 0
global_total = 0
document_accuracies = []

# Procesar cada archivo en el directorio de originales y extraídos
# Iteramos sobre cada archivo JSON en el directorio original y comparamos con el
archivo correspondiente en el directorio extraído.
for filename in os.listdir(original_dir):
    if filename.endswith('.json'):
        original_file_path = os.path.join(original_dir, filename)
        extracted_file_path = os.path.join(extracted_dir,
filename.replace('.json', '_result.json'))

        if os.path.exists(extracted_file_path):
            original_data = load_json(original_file_path)
            extracted_data = load_json(extracted_file_path)

            correct, total = compare_dicts(original_data, extracted_data)
```

```
            global_correct += correct
            global_total += total
            document_accuracies.append(correct / total if total > 0 else 0)

            for category in categories:
                total_entries[category] += 1
                original_value = str(original_data.get(category, "")).strip()
                extracted_value = str(extracted_data.get(category, "")).strip()

                # Normalizar fechas si el campo es una fecha
                if "fecha" in category or "periodo" in category:
                    original_value = normalize_date(original_value)
                    extracted_value = normalize_date(extracted_value)

                if original_value == extracted_value:
                    total_correct[category] += 1

# Calcular y mostrar los porcentajes de acierto
# Calculamos y mostramos la tasa de acierto para cada categoría de entidad.
print("Tasa de acierto por categoría:")
for category in categories:
    if total_entries[category] > 0:
        accuracy = (total_correct[category] / total_entries[category]) * 100
        print(f"{category}: {accuracy:.2f}%")
    else:
        print(f"{category}: No hay datos")

# Calcular y mostrar la tasa de acierto global
if global_total > 0:
    global_accuracy = (global_correct / global_total) * 100
    print(f"Tasa de acierto global: {global_accuracy:.2f}%")
else:
    print("No hay datos para calcular la tasa de acierto global")

# Calcular y mostrar los porcentajes de documentos con diferentes niveles de
acierto
thresholds = [0.20, 0.40, 0.50, 0.60, 0.75, 0.90, 1.00]
threshold_counts = {threshold: 0 for threshold in thresholds}

for accuracy in document_accuracies:
    for threshold in thresholds:
        if accuracy >= threshold:
            threshold_counts[threshold] += 1

print("Porcentaje de documentos con diferentes niveles de acierto:")
for threshold in thresholds:
    percentage = (threshold_counts[threshold] / len(document_accuracies)) * 100
    print(f"≥ {int(threshold * 100)}%: {percentage:.2f}%")



_____
_____

# levenshtein.py
```

```python
# Este script se utiliza para calcular el score de precisión de nuestro modelo de extracción de entidades (NER).
# La métrica utilizada es la distancia de Levenshtein, que mide la similitud entre dos cadenas de texto.
# Nos enviarán estos archivos JSON que hemos obtenido, y ellos usarán un script para calcular el score
# que hemos obtenido. Este score se expresará en porcentaje y se calcula como la media de una métrica basada
# en la distancia de Levenshtein de todos los campos de todos los documentos.

import os
import json
import Levenshtein
from dateutil.parser import parse

# Directorios
original_dir = 'C:/Users/34670/Desktop/python/Hack a boss/proyecto_decide/training/'
extracted_dir = 'C:/Users/34670/Desktop/python/Hack a boss/proyecto_decide/libretas/validaciones/'

# Categorías de entidades
categories = [
    "nombre_cliente", "dni_cliente", "calle_cliente", "cp_cliente", "población_cliente",
    "provincia_cliente", "nombre_comercializadora", "cif_comercializadora", "dirección_comercializadora",
    "cp_comercializadora", "población_comercializadora", "provincia_comercializadora", "número_factura",
    "inicio_periodo", "fin_periodo", "importe_factura", "fecha_cargo", "consumo_periodo", "potencia_contratada"
]

# Función para cargar JSON
def load_json(file_path):
    with open(file_path, 'r', encoding='utf-8') as file:
        return json.load(file)

# Función para normalizar fechas al formato dd.mm.yyyy
def normalize_date(date_str):
    try:
        date_obj = parse(date_str, dayfirst=True)
        return date_obj.strftime("%d.%m.%Y")
    except (ValueError, OverflowError):
        return date_str

# Función para calcular el score basado en la distancia de Levenshtein
# La métrica utilizada es:
# Score = Σ (1 - L(ŝ, s) / len(s)) / n
# Donde L(a,b) es la distancia de Levenshtein entre las cadenas a y b, s es el string del campo i-ésimo,
# ŝ es el string de nuestra predicción para el campo i-ésimo, y len() devuelve la longitud de un string.
def calculate_levenshtein_score(original, extracted):
```

```
    score_sum = 0
    n = len(categories)

    for category in categories:
        original_value = str(original.get(category, "")).strip()
        extracted_value = str(extracted.get(category, "")).strip()

        # Normalizar fechas si el campo es una fecha
        if "fecha" in category or "periodo" in category:
            original_value = normalize_date(original_value)
            extracted_value = normalize_date(extracted_value)

        levenshtein_distance = Levenshtein.distance(original_value,
extracted_value)
        max_len = max(len(original_value), 1)
        score_sum += (1 - (levenshtein_distance / max_len))

    return score_sum / n

# Inicializar la suma de scores
total_score = 0
num_files = 0

# Procesar cada archivo en el directorio de originales y extraídos
for filename in os.listdir(original_dir):
    if filename.endswith('.json'):
        original_file_path = os.path.join(original_dir, filename)
        extracted_file_path = os.path.join(extracted_dir,
filename.replace('.json', '_result.json'))

        if os.path.exists(extracted_file_path):
            original_data = load_json(original_file_path)
            extracted_data = load_json(extracted_file_path)

            file_score = calculate_levenshtein_score(original_data,
extracted_data)
            total_score += file_score
            num_files += 1

# Calcular y mostrar el score promedio
if num_files > 0:
    average_score = (total_score / num_files) * 100
    print(f"Score promedio: {average_score:.2f}%")
else:
    print("No hay archivos para calcular el score")
```