Proceso de automatización de la preparación de datos, construcción del modelo, persistencia del modelo y acceso por medio de API

Inteligencia de negocios – ISIS3301

Contenido

1 Automatización de la preparación de datos

1.1 División de datos

Para comenzar, se tomó la decisión de repartir los datos en baches de 2400 filas para cada división. Note que, obviamente, se tomaron los datos etiquetados que fueron otorgados por el negocio. Del mismo modo, estos datos fueron usados para entrenar el modelo que posteriormente es guardado en un pipeline para poder ser persistido. A continuación, se presenta como se hizo esta carga y división de datos. Recuerde que todo lo sustentado aquí, hace parte del notebook desarrollado con el nombre de 'Proyecto_1_etapa_2'.

```
# Usamos la libreria pandas para leer el archivo excel
data = pd.read_excel("./data/cat_6716.xlsx")

# Dividimos el conjunto de datos
X_train = data.loc[:2400, 'Textos_espanol'].values
y_train = data.loc[:2400, 'sdg'].values
X_test = data.loc[2400:, 'Textos_espanol'].values
y_test = data.loc[2400:, 'sdg'].values
```

1.2 Preprocesador para la transformación de datos

Para tokenizar los datos de entrada, se define la función tokenizer y se tomó la decisión de usar la librería spacy, junto con sus lemas y sus non stop words, puesto que ofrece el idioma español para estos propósitos. La tokenizacion es sumamente importante en este proyecto para el procesamiento de lenguaje natural, ya que divide el texto en unidades significativas, lo que permite el análisis, la normalización y el procesamiento eficiente, además de ser fundamental en diversas aplicaciones de NLP. Finalmente, también se hace el preprocesador para limpiar los datos, aplicar flexiones gramaticales y otros cambios necesarios para poder hacer uso de los datos de entrada.

```
# Definimos la funcion para tokenizar el texto con la libreria en
español spacy
def tokenizar(texto):
    return [token.text for token in nlp(texto) if not token.is_stop]

# Definimos el preprocessador para transformar los datos antes de
ajustarlos al modelo
def preprocessor(tokens):
```

```
    # Funciones de limpieza para preprocesar una lista de tokens
    tokens = [re.sub('[\W]+', ' ', token.lower()) for token in tokens]
    tokens = [unidecode(token) for token in tokens]
    tokens = ['[NUM]' if re.match(r'\d+(\.\d+)?', token) else token for
token in tokens]
    tokens = [token for token in tokens if token not in
nlp.Defaults.stop_words]  # Filtrar stop words

    # Aplicamos las limpiezas adicionales
    for i in range(len(tokens)):
        tokens[i] = tokens[i].replace('Ã¡', 'a')
        tokens[i] = tokens[i].replace('Ã©', 'e')
        tokens[i] = tokens[i].replace('Ã³', 'o')
        tokens[i] = tokens[i].replace('Ã±', 'ñ')
        tokens[i] = tokens[i].replace('Ã', 'i')

    return tokens

# Creamos el pipeline para la preparación de datos
data_prep_pipeline = Pipeline([
    ('preprocessor', FunctionTransformer(func=preprocessor,
validate=False)),
])

# Aplicamos la transformación al conjunto de entrenamiento
X_train_transformed = data_prep_pipeline.fit_transform(X_train)
```

2 Automatización del modelo entrenado

Para automatizar el modelo, que es construido usando el modelo de tf-idf desarrollado en la etapa 1, se usaron los mismos parámetros encontrados con el cross validation de dicha etapa. En el pipeline se guarda el vectorizador Tfidf y la regresión logística que emplea el algoritmo. Finalmente ajustamos y entrenamos el modelo que esta siendo persistido en el pipeline.

```
best_params = {
    'vect': {
        'ngram_range': (1, 1),
        'tokenizer': None,
    },
    'clf': {
        'C': 100.0,
        'penalty': 'l2',
    }
}
```

```
# Creamos un nuevo pipeline con TfidfVectorizer y LogisticRegression
con los mejores parámetros
model_pipeline = Pipeline([
    ('vect', TfidfVectorizer(**best_params['vect'])),  # Configuramos
el vectorizador con los mejores parámetros
    ('clf', LogisticRegression(**best_params['clf']))  # Configuramos
el clasificador con los mejores parámetros
])

# Ajustamos el modelo al conjunto de entrenamiento transformado
model_pipeline.fit(X_train_transformed, y_train)
```

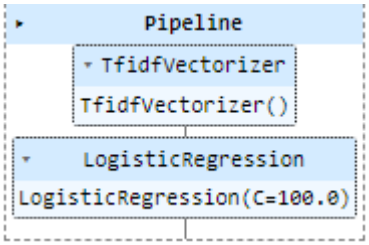

ILUSTRACIÓN 1: OUTPUT PIPELINE MODELO

3 Persistencia de los pipelines

En conclusión, hasta el momento tenemos dos (2) pipelines para persistir y usar como procesamiento en el API que diseñamos. La exportación de los pipelines se hizo gracias a la librería *joblib* la cual empaqueta estos procesos y transformaciones en archivos *.pkl* que después vamos a desempaquetar y usar en nuestro API. Así pues, a continuación, se muestra como se hizo dicho proceso de empaquetamiento y exportación en el notebook, el cual realmente, fue el proceso más sencillo del proyecto.

```
import joblib

# Guardamos el modelo entrenado en un archivo
joblib.dump(model_pipeline, 'modelo_logreg_0.pkl')
# Exportamos tambien el pipeline de transformación de datos
joblib.dump(data_prep_pipeline, 'data_prep_pipeline_0.pkl')
```

4 Acceso del API

Flask es un framework de desarrollo web para Python que permite crear aplicaciones web de manera sencilla y eficiente. Note que todo lo se va a explicar a continuación hace referencia al código desarrollado en el Backend del github adjunto en este proyecto. En este contexto, hemos utilizado Flask para desarrollar una aplicación web con las siguientes características:

- Configuración de la aplicación: Hemos configurado una aplicación Flask llamada app y definido las rutas a las que los usuarios pueden acceder.

- Ruta para cargar archivos (end-point de la API): Hemos creado una ruta /predict que permite a los usuarios subir su texto. Esta ruta es accesible mediante una solicitud POST, y cuando un usuario carga un archivo, el código dentro de la función predict() maneja la carga del archivo.

- Procesamiento de archivos: El código dentro de la función upload_file() verifica el texto que se envió mediante el POST, aplica las transformaciones, hace uso de los pipelines importados y finalmente imprime en un template renderizado el resultado de la predicción.

- Plantillas HTML: Flask permite renderizar plantillas HTML para presentar información en una interfaz web. Se han definido plantillas personalizadas para brindar una breve experiencia al usuario y una interfaz agradable para poder usar la api diseñada.

- Integración con librerías: Hemos utilizado librerías como pandas, joblib y otras para cargar modelos y realizar el procesamiento de datos. Note que todas estas dependencias son instaladas localmente en la maquina que despliega el servicio (endpoint) para la API.

```
app = Flask(__name__)
model = joblib.load('model.pkl')

@app.route('/')
def hello():
    return render_template('index.html')

@app.route('/predict', methods=['POST'])
def predict():
    if request.method == 'POST':
        input_texts = request.form.getlist('text')

        # Realiza las predicciones para cada texto
        predictions = model.predict(input_texts)

        # Devuelve las predicciones en un formato adecuado
        response = {'predictions': predictions.tolist()}

        return render_template('data.html',
predictions=response['predictions'])
    else:
        return 'Unsupported method'
```