

## Proyecto 1 Etapa 2

### Automatización y uso de modelos de analítica de textos

Andrea Galindo Cera – 202122477

Santiago Navarrete Varela – 202211202

Luis Fernando Ruiz – 202211513

### Descripción del Proceso de Automatización

#### Preparación de datos y construcción del modelo

El proceso de preparación de datos y construcción del modelo se implementó utilizando un pipeline compuesto por varias etapas de preprocesamiento de texto y un modelo de clasificación final. A continuación, se describen los pasos incluidos en el pipeline, que fue diseñado para transformar texto crudo en características aptas para el modelo de aprendizaje automático y generar predicciones precisas.

- **ExpandContractions:** Esta etapa se encarga de expandir contracciones en el texto (por ejemplo, "it's" se convierte en "it is"). Esto mejora la consistencia del texto y asegura que el modelo interprete las contracciones de manera uniforme.



```
1 class ExpandContractions(BaseEstimator, TransformerMixin):
2     def __init__(self):
3         pass
4
5     def fit(self, X, y=None):
6         return self
7
8     def transform(self, X):
9         X = X.copy()
10        X['expanded_text'] = X['Textos_espanol'].apply(contractions.fix)
11        return X
```

- **Tokenizer:** Después de expandir las contracciones, se aplica un tokenizador que divide el texto en palabras individuales o tokens. Esto es esencial para que el modelo pueda trabajar con palabras específicas y extraer características relevantes de cada una.

```

1 class Tokenizer(BaseEstimator, TransformerMixin):
2     def __init__(self):
3         pass
4
5     def fit(self, X, y=None):
6         return self
7
8     def transform(self, X):
9         X = X.copy()
10        X['tokenized_words'] = X['expanded_text'].apply(word_tokenize)
11        return X

```

- **NormalizeText:** Esta fase normaliza el texto, lo que incluye convertir todo el texto a minúsculas, eliminar caracteres no deseados, y asegurar una representación consistente. Este paso es importante para eliminar variaciones en la escritura que podrían confundir al modelo (por ejemplo, "Hola" y "hola" serían tratados como diferentes sin normalización).

```

1 class NormalizeText(BaseEstimator, TransformerMixin):
2     def __init__(self):
3         self.infect_engine = inflect.engine()
4         self.stop_words_spanish = set(stopwords.words('spanish'))
5         self.stop_words_english = set(stopwords.words('english'))
6
7     def fit(self, X, y=None):
8         return self
9
10    def transform(self, X):
11        X = X.copy()
12        X['words_normalize'] = X['tokenized_words'].apply(self.normalize)
13        return X
14
15    def remove_non_ascii(self, words):
16        new_words = []
17        for word in words:
18            new_word = unicodedata.normalize('NFKD', word).encode('ascii', 'ignore').decode('utf-8', 'ignore')
19            new_words.append(new_word)
20        return new_words
21
22    def to_lowercase(self, words):
23        new_words = []
24        for word in words:
25            new_words.append(word.lower())
26        return new_words
27
28    def remove_punctuation(self, words):
29        new_words = []
30        for word in words:
31            new_word = re.sub(r'^\w+$', '', word)
32            if new_word != '':
33                new_words.append(new_word)
34        return new_words
35
36    def replace_numbers(self, words):
37        new_words = []
38        for word in words:
39            if word.isdigit():
40                new_word = self.infect_engine.number_to_words(word)
41                new_words.append(new_word)
42            else:
43                new_words.append(word)
44        return new_words
45
46    def remove_stopwords(self, words):
47        new_words = []
48        for word in words:
49            if word not in self.stop_words_spanish and word not in self.stop_words_english:
50                new_words.append(word)
51        return new_words
52
53    def normalize(self, words):
54        words = self.remove_non_ascii(words)
55        words = self.to_lowercase(words)
56        words = self.remove_punctuation(words)
57        words = self.replace_numbers(words)
58        words = self.remove_stopwords(words)
59        return words
60

```

- **StemAndLemmatize:** En este paso, se aplica un proceso de stemming y lematización, que reduce las palabras a su forma raíz o lema (por ejemplo, "corriendo" se convierte en "correr"). Esto permite que el modelo identifique relaciones entre palabras derivadas de una misma raíz.

```

1 class StemAndLemmatize(BaseEstimator, TransformerMixin):
2     def __init__(self):
3         self.stemmer = LancasterStemmer()
4         self.lemmatizer = WordNetLemmatizer()
5
6     def fit(self, X, y=None):
7         return self
8
9     def transform(self, X):
10        X = X.copy()
11        X['s&l'] = X['words_normalize'].apply(self.stem_and_lemmatize)
12        return X
13
14    def stem_words(self, words):
15        stems = [self.stemmer.stem(word) for word in words]
16        return stems
17
18    def lemmatize_verbs(self, words):
19        lemmas = [self.lemmatizer.lemmatize(word, pos='v') for word in words]
20        return lemmas
21
22    def stem_and_lemmatize(self, words):
23        stems = self.stem_words(words)
24        lemmas = self.lemmatize_verbs(words)
25        return stems, lemmas

```

- **CombineOriginalStemLemma:** Esta etapa combina las versiones originales de las palabras con sus formas derivadas después del stemming y la lematización, creando una representación más rica y completa del texto.

```

1 class CombineOriginalStemLemma(BaseEstimator, TransformerMixin):
2     def __init__(self):
3         pass
4
5     def fit(self, X, y=None):
6         return self
7
8     def transform(self, X):
9         X = X.copy()
10        X['text_clean'] = X.apply(lambda row: ' '.join(self.combine_original_stem_lemma(row['words_normalize'], row['s&l'][0], row['s&l'][1])), axis=1)
11        return X
12
13    def combine_original_stem_lemma(self, original, stems, lemmas):
14        combined = []
15        for orig, stem, lemma in zip(original, stems, lemmas):
16            if lemma.endswith('ar') or lemma.endswith('er') or lemma.endswith('ir'):
17                combined.append(lemma)
18            elif len(stem) < len(orig) / 2:
19                combined.append(stem)
20            else:
21                combined.append(orig)
22        return combined

```

- **TextVectorizer:** El vectorizador de texto transforma las palabras en vectores numéricos que el modelo puede interpretar. Utiliza técnicas como TF-IDF (Term Frequency-Inverse Document Frequency) para asignar pesos a las palabras según su frecuencia en el documento y en el corpus general.



```

1 class TextVectorizer(BaseEstimator, TransformerMixin):
2     def __init__(self):
3         self.vectorizer = TfidfVectorizer()
4
5     def fit(self, X, y=None):
6         self.vectorizer.fit(X['text_clean'])
7         return self
8
9     def transform(self, X):
10        X_tfidf = self.vectorizer.transform(X['text_clean'])
11        return X_tfidf
12

```

- **SVC:** El modelo de clasificación utilizado es una Máquina de Vectores de Soporte (SVM) lineal, implementada mediante SVC. Este modelo separa las clases (en este caso, las categorías de texto relacionadas con los ODS 3, 4 y 5) generando un hiperplano óptimo que maximiza la separación entre las clases. El SVM se entrenó con el objetivo de maximizar la precisión en la clasificación de opiniones ciudadanas, basándose en las características textuales extraídas a lo largo del pipeline de preprocesamiento.



```

1 ('model', SVC(probability=True))

```

- **MultinomialNB:** Este modelo de clasificación, basado en Naive Bayes, se utiliza exclusivamente en la opción de reentrenamiento incremental. MultinomialNB es un modelo estadístico que asume independencia condicional entre las características y es especialmente adecuado para problemas de clasificación de texto. En este caso, se utiliza para clasificar opiniones ciudadanas relacionadas con los ODS 3, 4 y 5. Su enfoque probabilístico lo hace eficiente en escenarios donde los datos llegan de manera continua, permitiendo ajustar el modelo rápidamente sin necesidad de reentrenarlo desde cero. Sin embargo, al ser un modelo más simple, puede no capturar todas las complejidades de las opiniones textuales como lo haría un modelo más avanzado.




Este pipeline está completamente automatizado para llevar a cabo el preprocesamiento de textos de manera eficiente y consistente, asegurando que los datos de entrada estén listos para el entrenamiento o la predicción con el modelo de aprendizaje automático. Además, el pipeline permite que las mismas transformaciones se apliquen tanto a los datos de entrenamiento como a los datos de prueba, manteniendo la consistencia en todo el proceso.

### Persistencia del modelo

El modelo entrenado se guardó utilizando la librería **dill**, que permite serializar el pipeline completo, incluyendo tanto las transformaciones de datos como el modelo de clasificación (SVC o MultinomialNB). Decidimos utilizar dill en lugar de joblib porque dill ofrece una mayor flexibilidad al permitir la serialización de objetos más complejos, como funciones personalizadas o clases, algo que joblib no puede hacer de manera tan efectiva y que es necesario en este caso ya que tenemos varias funciones personalizadas. Esta capacidad de dill es importante para garantizar que todo el pipeline de procesamiento de datos y el modelo se puedan almacenar y reutilizar sin problemas, facilitándonos así el reentrenamiento continuo y el despliegue del modelo en otros entornos. El siguiente código muestra cómo se guardó el pipeline en un archivo binario:

```
import dill
with open('model-group20.dill', 'wb') as f:
    dill.dump(pipeline_svm, f)
```

Para reutilizar el modelo en futuras predicciones, simplemente se carga desde el archivo guardado y se utiliza para realizar predicciones:



```
1 # Cargar el modelo desde el archivo .dill
2 with open("model-group20.dill", "rb") as f:
3     model = dill.load(f)
```

Este enfoque nos asegura que tanto las transformaciones como el modelo se mantengan consistentes y disponibles para predicciones futuras.

### Desarrollo de la API REST (primer endpoint)

Se implementó un endpoint para realizar predicciones a partir de una o más instancias de datos enviadas en formato JSON. El endpoint acepta una lista de objetos con las características necesarias, convierte los datos en un *DataFrame* y luego utiliza el modelo cargado para generar las predicciones correspondientes.

El proceso es el siguiente:

1. **Entrada:** Se reciben uno o más objetos JSON con los datos de entrada, que serían las opiniones que se buscan clasificar.
2. **Transformación:** Los datos se convierten en un *DataFrame* para ser procesados por el pipeline.
3. **Salida:** El modelo realiza las predicciones y la probabilidad de que pertenezca a cada ODS y devuelve los resultados en formato JSON.

El siguiente código muestra la implementación del endpoint:

```

1 @app.post("/predict")
2 def make_predictions(dataModels: List[DataModel]):
3     # Convertir los datos recibidos a un DataFrame
4     df = pd.DataFrame([data.model_dump() for data in dataModels])
5
6     with open("model-group20.dill", "rb") as f:
7         model = dill.load(f)
8
9     # Realizar las predicciones y calcular las probabilidades
10    predicciones = model.predict(df)
11    probabilidades = model.predict_proba(df) # Esto devuelve las probabilidades
12
13    # Crear una lista de diccionarios para devolver tanto la predicción como la probabilidad
14    resultados = [
15        {
16            "prediccion": int(pred), # La predicción de la clase
17            "probabilidades": prob.tolist() # Las probabilidades para todas las clases
18        }
19        for pred, prob in zip(predicciones, probabilidades)
20    ]
21
22    return {"resultados": resultados}

```

Este enfoque asegura que se puedan realizar predicciones sobre múltiples instancias a la vez, devolviendo las predicciones correspondientes para cada una.

## Desarrollo de la API REST (segundo endpoint)

El segundo endpoint de la API REST tiene como objetivo permitir que los usuarios envíen nuevos datos para reentrenar el modelo de clasificación de opiniones sobre los ODS (3, 4 y 5). Este reentrenamiento puede realizarse en tres modalidades diferentes, según las necesidades del usuario. A continuación, se explica cómo funciona el endpoint:

### 1. Recepción de instancias para reentrenar el modelo:

El endpoint acepta nuevos datos enviados por el usuario en forma de archivo (CSV o Excel), los cuales son utilizados para reentrenar el modelo. Dependiendo de la opción seleccionada, el reentrenamiento puede realizarse con:

- **Naive Bayes** de manera incremental.
- **SVC** añadiendo nuevos datos a los antiguos.
- **SVC** solo con los nuevos datos.

### 2. Cálculo y devolución de métricas de desempeño:

Tras el reentrenamiento del modelo con los nuevos datos, el sistema calcula y devuelve tres métricas de desempeño clave:

- **Precision:** Mide la proporción de verdaderos positivos sobre el total de positivos predichos.
- **Recall:** Indica la capacidad del modelo para encontrar todos los verdaderos positivos.

- **F1-score:** Es una métrica que combina la precisión y el recall en un solo valor.

Estas métricas permiten a los usuarios evaluar el rendimiento del modelo después del reentrenamiento y verificar si ha mejorado o se ha mantenido en el tiempo.

### 3. Actualización del archivo del modelo:

Una vez que el modelo ha sido reentrenado, el archivo del modelo (**model-group20.dill**) se actualiza y guarda en disco. Esto garantiza que el modelo siempre esté actualizado con la información más reciente.

Además, se guarda un historial de los datos utilizados en el reentrenamiento en un archivo CSV (HistoryData.csv), que contiene tanto las nuevas instancias de datos como las antiguas (si corresponde), junto con la variable objetivo (la clasificación ODS).

```
1 # Método para reentrenar el pipeline y guardar el modelo, devolviendo métricas
2 def retrain_model(self, X, y, save_path="model-group20.dill", X_old=None, y_old=None):
3
4     if self.retraining_option == 'naive_bayes_incremental':
5         # Cambiar el modelo a Naive Bayes si no está ya configurado
6         if not isinstance(self.pipeline.named_steps['model'], MultinomialNB):
7             self.pipeline.named_steps['model'] = MultinomialNB()
8
9         # Transformar los datos nuevos con las partes previas del pipeline (sin incluir el modelo)
10        steps_before_model = Pipeline(self.pipeline.steps[:-1])
11        X_transformed = steps_before_model.fit_transform(X)
12
13        # Ajustar el modelo Naive Bayes de manera incremental
14        classes = np.unique(y)
15        self.pipeline.named_steps['model'].partial_fit(X_transformed, y, classes=classes)
16
17        # Guardar el dataframe con la variable objetivo en un csv
18        X_combined = pd.concat([X_old, X], axis=0)
19        y_combined = np.concatenate([y_old, y], axis=0)
20        X_combined['sdg'] = y_combined
21        X_combined.to_csv('HistoryData.csv', index=False)
22
23    elif self.retraining_option == 'svc_add_new':
24        # Asegurarse de que los datos nuevos y viejos tengan las mismas columnas
25        if not np.array_equal(X_old.columns, X.columns):
26            raise ValueError("Las columnas de los datos antiguos y nuevos no coinciden")
27
28        X_combined = pd.concat([X_old, X], axis=0)
29        y_combined = np.concatenate([y_old, y], axis=0)
30
31        #
32
33        self.pipeline.fit(X_combined, y_combined)
34
35        # Guardar el dataframe combinado con la variable objetivo en un csv
36        X_combined['sdg'] = y_combined
37        X_combined.to_csv('HistoryData.csv', index=False)
38
39    else:
40
41        self.pipeline.fit(X, y)
42
43        # Guardar el dataframe con la variable objetivo en un csv
44        X['sdg'] = y
45        X.to_csv('HistoryData.csv', index=False)
46
47    y_pred = self.pipeline.predict(X)
48
49    precision = precision_score(y, y_pred, average='weighted')
50    recall = recall_score(y, y_pred, average='weighted')
51    f1 = f1_score(y, y_pred, average='weighted')
52
53    with open(save_path, 'wb') as file:
54        dill.dump(self.pipeline, file)
55
56    return {
57        "precision": precision,
58        "recall": recall,
59        "f1_score": f1
60    }
61 }
```

## Definiciones de Reentrenamiento



## Reentrenamiento Incremental con Naive Bayes

- **Descripción:** En esta opción, se utiliza el algoritmo Naive Bayes con un enfoque de reentrenamiento incremental. Esto significa que el modelo ajusta sus parámetros basándose únicamente en los nuevos datos de opiniones ciudadanas relacionadas con los ODS. El reentrenamiento se realiza sin volver a entrenar desde cero, lo que permite que el modelo procese los nuevos datos rápidamente y ajuste su clasificación sin perder los conocimientos previos.
- **Ventaja:** Es eficiente en términos de procesamiento y tiempo, lo que es ideal cuando los datos sobre opiniones llegan en tiempo real o de manera continua. Los funcionarios pueden recibir resultados más rápido sin necesidad de esperar que el modelo procese todos los datos históricos. Naive Bayes es especialmente útil en contextos donde las relaciones entre las variables son más simples y puede aprender a partir de datos nuevos rápidamente.
- **Desventaja:** Naive Bayes es un modelo estadístico simple que puede no capturar las complejidades en el lenguaje o los matices de las opiniones sobre los ODS, lo que podría afectar su precisión en clasificaciones más complejas, como en opiniones que combinan múltiples aspectos de los ODS 3, 4 y 5.

## 2. Reentrenamiento desde cero con SVC utilizando los datos antiguos y nuevos

- **Descripción:** En esta opción, el modelo SVC se reentrena desde cero utilizando tanto los datos antiguos como los nuevos, lo que implica que el modelo se ajusta tomando en cuenta todo el historial de opiniones clasificadas en los ODS. Esto es útil cuando los funcionarios necesitan un análisis más profundo de las opiniones de los ciudadanos sobre los ODS, ya que el modelo puede aprender tanto de los datos históricos como de los nuevos.
- **Ventaja:** Al utilizar toda la información disponible, el modelo es más robusto y puede mejorar su precisión en la clasificación de opiniones complejas. Esto es ideal cuando los funcionarios del UNFPA requieren análisis detallados y basados en datos completos para tomar decisiones más informadas sobre el progreso hacia los ODS 3, 4 y 5.
- **Desventaja:** El reentrenamiento completo puede ser más lento y costoso en términos de tiempo, lo que puede retrasar el procesamiento si se requiere una actualización rápida del modelo. Además, el modelo SVC no soporta reentrenamiento incremental, por lo que siempre debe reentrenarse desde cero con todos los datos disponibles.

## 3. Reentrenamiento desde cero con SVC utilizando solo los nuevos datos

- **Descripción:** En este escenario, se entrena el modelo SVC desde cero, pero solo con las opiniones más recientes. El modelo no utiliza los datos históricos, lo que puede ser útil cuando se quiere dar mayor relevancia a las tendencias más recientes en las opiniones de los ciudadanos sobre los ODS.
- **Ventaja:** Este enfoque es más rápido que utilizar todos los datos (antiguos y nuevos) y permite a los funcionarios obtener resultados basados únicamente en los últimos datos recolectados, lo que puede ser útil cuando las prioridades o contextos de los ciudadanos cambian rápidamente en relación a los ODS 3, 4 y 5.
- **Desventaja:** Al no incluir los datos antiguos, el modelo puede perder información valiosa sobre tendencias pasadas, lo que podría afectar su capacidad para generalizar y reconocer patrones consistentes en las opiniones. Esto puede llevar a una menor precisión en clasificaciones que dependen de un contexto más amplio o histórico.

## Desarrollo de la aplicación y justificación

La aplicación está dirigida a los funcionarios del Fondo de Poblaciones de las Naciones Unidas (UNFPA), cuya responsabilidad principal es analizar y evaluar las opiniones de los ciudadanos en relación con los Objetivos de Desarrollo Sostenible (ODS). Estos funcionarios necesitan herramientas que les permitan procesar de manera eficiente grandes volúmenes de información textual para identificar los problemas relacionados con la salud (ODS 3), la educación (ODS 4) y la igualdad de género (ODS 5).

Además, tomando en cuenta la retroalimentación del mapa de actores, confirmamos que los roles, la importancia y los beneficios asociados a cada actor, como el UNFPA, la población beneficiada, las Naciones Unidas, y el equipo de desarrolladores y científicos de datos, estaban correctamente definidos desde el inicio. No fue necesario realizar ajustes, ya que se mantuvieron intactos los riesgos y beneficios establecidos.

La conexión con el proceso de negocio se centra en la automatización del análisis de opiniones ciudadanas, reduciendo significativamente el tiempo y los recursos necesarios para realizar evaluaciones manuales. Esto puede optimizar la toma de decisiones, permitiendo que los funcionarios identifiquen de forma más eficiente los patrones de opinión que requieren atención, lo que es clave para diseñar políticas públicas efectivas y planificar intervenciones precisas relacionadas a cada ODS. Además, la aplicación cuenta con funcionalidades que permiten reentrenar el modelo de análisis, asegurando que este se mantenga actualizado y refleje mejor las

inquietudes actuales de los ciudadanos. Este reentrenamiento constante mejora la capacidad del UNFPA para adaptar sus acciones a medida que evoluciona la opinión pública, maximizando el impacto de sus intervenciones y garantizando que sus decisiones estén siempre basadas en los datos más recientes.

## Resultados

Se desarrolló una aplicación web para interactuar con el modelo analítico, permitiendo tanto la **predicción** de los textos ingresados por el usuario como la opción de **reentrenar el modelo** con nuevos datos. La aplicación muestra el resultado de la predicción junto con la probabilidad asociada y ejecuta correctamente el proceso de reentrenamiento, asegurando que el modelo se actualice con nuevas opiniones cuando sea necesario.

En el video entregado, se muestra cómo la aplicación opera con todas sus funcionalidades de forma correcta con una interfaz de usuario intuitiva y que permite a los funcionarios interactuar de forma directa con los resultados del modelo.

## Trabajo en Equipo

**Santiago, Ingeniero de Software responsable del segundo endpoint:** Santiago fue responsable de implementar el segundo endpoint, que permite el reentrenamiento del modelo y devuelve métricas de desempeño como Precision y Recall. Dedicó aproximadamente 5 horas en la realización del proyecto.

**Luis, Ingeniero de Software Responsable del Desarrollo de la Aplicación:** Luis fue responsable de la creación de la aplicación web, asegurando que la interfaz permitiera la interacción del usuario con el modelo analítico e integrando ambos endpoints en la aplicación. Dedicó aproximadamente 5 horas en la realización del proyecto.

**Andrea, Ingeniera de Software Responsable del Primer Endpoint:** Andrea estuvo a cargo de la gestión del proyecto, organizando las reuniones. Se encargó de la implementación del primer endpoint, que recibe las instancias de datos en formato JSON y devuelve las predicciones generadas por el modelo, incluyendo las probabilidades. Dedicó aproximadamente 5 horas en la realización del proyecto.

**Trabajo en equipo para el modelo y pipeline:** Los tres integrantes además tomaron el rol de **ingeniero de datos** y colaboraron para desarrollar el modelo de analítica de textos y el pipeline de preprocesamiento,

En cuanto a la distribución de puntos, decidimos asignar 33,3 puntos a cada uno, reflejando una contribución equitativa y significativa al éxito del proyecto. Para futuras

entregas, identificamos oportunidades de mejora en cuanto la optimización de nuestros tiempos de reunión para acelerar la toma de decisiones. Esto nos permitirá aumentar la cohesión del equipo y la eficiencia del proyecto.