DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y COMPUTACIÓN
Inteligencia de negocios 202220 – Laboratorio 4
PROFESORA: Haydemar Núñez

Nombres	Apellidos	Código	Login
María Sofía	Álvarez López	201729031	ms.alvarezl
Brenda Catalina	Barahona Pinilla	201812721	bc.barahona
Álvaro Daniel	Plata Márquez	201820098	ad.plata

Proyecto 1 – Etapa 2: Automatización analítica de textos

El objetivo de este proyecto es automatizar un proceso replicable para aplicar la metodología de procesamiento de lenguaje natural en la construcción de un modelo analítico que pretende clasificar textos médicos en cinco categorías diferentes: neoplasias, enfermedades del sistema digestivo, enfermedades del sistema nervioso, enfermedades cardiovasculares y condiciones patológicas generales. Asimismo, se desarrolló una aplicación que utiliza dicho modelo para ser utilizada por los médicos de un hospital determinado (en este caso, sea este el Hospital de los Alpes) con el fin de apoyar a los médicos en el diagnóstico de pacientes. Este informe describe el proceso de automatización del proceso de preparación de datos, la construcción del modelo, la persistencia del modelo y el acceso por medio del API; así como el desarrollo de la aplicación y su importancia para la organización y sus médicos.

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

Al desplegar modelos de Machine Learning, uno de los aspectos más importantes resulta ser la automatización de la preparación de los datos, la construcción del modelo y su persistencia. Para facilitar el proceso, se utilizan arquitecturas como los pipelines de scikit-learn. Asimismo, es necesario desarrollar un API usando un Framework como FastAPI para el despliegue. Esta sección describe exhaustivamente este proceso.

Lo primero que automatizamos fue el preprocesamiento de los datos y su preparación, para que ya estuviesen listos para el modelo de Machine Learning a utilizar. Aquí vale la pena anotar que una descripción exhaustiva del perfilamiento de los datos se realizó en el informe anterior, pero puede encontrarse también en el notebook de perfilamiento en la carpeta notebooks.

La primera fase del preprocesamiento consiste en el manejo del ruido en textos. Esto constituye un reto gigante y, a pesar de que existen varias librerías que implementan funcionalidades para su manejo, es necesario implementar transformaciones individuales también. En este caso, se definieron 6 transformaciones individuales correspondientes a:

- Eliminación de caracteres no ASCII: Hace parte importante del preprocesamiento de las palabras. Con caracteres no-ASCII, el preprocesamiento puede verse terriblemente perjudicado.
- Conversión de mayúsculas en minúsculas: Asimismo, es importante que todas las palabras tengan una capitalización homogénea (en este caso, queremos que estén en minúscula).
- Eliminación de la puntuación: Por otro lado, consideramos que la puntuación no provee información adicional en este contexto. Adicionalmente, de no eliminarse, puede aumentar la dimensionalidad de los datos sin proveer más información. Por ejemplo, no tiene sentido pensar que "¡almuerzo!" y "almuerzo" sean palabras diferentes. Por ello removimos toda la puntuación usando expresiones regulares.

- Eliminación de los números: Los números no proveen información relevante para el problema en cuestión, puesto que no ayudan distinguir entre una enfermedad o la otra, pero sí pueden agregar dimensionalidad inútilmente al problema.
- Eliminación de las fechas (si las hay) también: las fechas son irrelevantes para el contexto del problema.
- Eliminación de las palabras vacías (artículos, pronombres, preposiciones y palabras irrelevantes en el contexto médico): Estas se denominan stop-words, en inglés. Son palabras que se usan en muchos contextos (como 'the') y no aportan información significativa en la construcción del modelo. Asimismo, definimos nuestras propias stop-words de acuerdo con el perfilamiento de datos realizado con las nubes de palabras, pues son palabras que no aportan significativamente al contexto, y eliminarlas podría ayudar a tener un mejor modelo.
- Arreglo de contracciones: Debido al carácter formal del contexto médico en el que se encuentra inmerso este proyecto, no esperamos que haya muchas contracciones (i.e. expresiones del estilo can't, doesn't, etc.). No obstante, se aplican las correcciones pertinentes.

Como fase final del manejo del ruido, se tokenizaron las palabras de los textos. Es decir, se separaron las palabras en tokens únicos – delimitados por el carácter espacio – y se creó una lista con todas las palabras de cada texto. Todas estas funcionalidades se implementaron en funciones de Python, las cuales a su vez se agruparon en una clase para poder ser añadidas al pipeline del modelo. Es importante anotar que, para que un pipeline de scikit-learn permita el uso de transformaciones personalizadas, es necesario usarlas en clases, que implementen tres métodos: `__init__`, `transform`, y `fit`. Como se ve en la imagen a continuación, la clase `Preprocessing()`, que encapsula todas las funcionalidades de limpieza de ruido, implementa los métodos mencionados previamente:

```
class Preprocessing():
    def __init__(self):
        pass
    def transform(self,X,y=None):
        X["medical_abstracts"] = X["medical_abstracts"].apply(contractions.fix)
        X["medical_abstracts"] = X["medical_abstracts"].apply(word_tokenize)
        X["medical_abstracts"] = X["medical_abstracts"].apply(noise_elimination)
        return X
    def fit(self, X, y=None):
        return self
```

Imagen 1: Implementación de una clase para ser posteriormente utilizada en el pipeline de preprocesamiento, con el fin de arreglar las contracciones, tokenizar las palabras y eliminar el ruido de los textos médicos recibidos.

Posteriormente, se procedió a realizar la fase de normalización de los datos, aplicando stemming y lematización sobre los textos. Allí, se realizó la eliminación de prefijos y sufijos, así como la lematización de los verbos. En el caso del Stemming, se consideró el uso de tres algoritmos: Porter, Snowball (Porter2) o Lancaster (Paice-Husk). La agresividad en el corte de raíces de las palabras de estos algoritmos aumenta, siendo Porter el menos agresivo y Lancaster el más agresivo. En este sentido, parece ser que Lancaster (a pesar de ser el más eficiente de todos), puede ser poco riguroso y así crear muchas ambigüedades. Asimismo, Porter2 es un poco más agresivo que Porter, sin perder mucho el origen de las palabras y con un tiempo de cómputo razonable. El mismo Porter, creador del algoritmo, argumenta que es una mejora de su algoritmo original. Con el fin de tener la mejor preparación de las palabras, en un tiempo de cómputo razonable, se usó Porter2. En el caso de la lematización, se utilizó `WordNetLemmatizer()` al ser el más usado en el mundo del procesamiento de textos. De forma similar, para esta fase se definió otra clase,

StemAndLemmatize(), que encapsula ambas funciones y las adecúa para ser utilizadas en el pipeline que se construyó posteriormente con scikit-learn.

```
class StemAndLemmatize():
    def __init__(self):
        pass
    def transform(self,X,y=None):
        X["medical_abstracts"] = X["medical_abstracts"].apply(stem_and_lemmatize)
        return X
    def fit(self, X, y=None):
        return self
```

Imagen 2: Implementación de una clase para ser posteriormente utilizada en el pipeline de preprocesamiento, con el fin de aplicar stemming y lematización sobre los textos médicos recibidos.

Antes de proseguir con el análisis, es importante señalar que todas estas clases se definen en un archivo externo al notebook, denominado clases.py, con el fin de evitar errores en la serialización al momento de guardar el modelo.

Con esto, ya se tenían definidas todas las funciones y clases necesarias para el preprocesamiento, con lo cual definimos el pipeline con dichos pasos, como se ve a continuación:

```
preproc = [
    ("preprocessing", Preprocessing()),
    ("stem_lemmatize", StemAndLemmatize())
]
```

```
pipe  = Pipeline(preproc)
```

Imagen 3: Creación del pipeline con los pasos de preprocesamiento descritos previamente.

Posteriormente, dicho pipeline fue ajustado a los datos con el fin de obtener el archivo preprocesado[1]. Una muestra de este archivo tras las transformaciones del pipeline puede verse en la siguiente imagen:

```
datos_train_procesados = pipe.fit_transform(data_train, data_train['problems_described'])
datos_train_procesados.head(5)
```

	medical_abstracts	problems_described
4888	[threedimension, system, longterm, cultur, col...	1
3247	[brainstem, auditori, evok, respons, acoust, n...	3
5261	[methodolog, mental, stress, test, cardiovascu...	4
9568	[applic, modifi, bioassay, monitor, serum, tei...	5
10074	[fine, surfac, structur, intraspin, neurenter,...	5

Imagen 4: Ajuste del pipeline a los datos. Puede apreciarse que cada uno de los textos médicos es ahora un conjunto de palabras, que ya han sido limpiadas de todo tipo de ruido, y a las que, además, se les ha aplicado el proceso de stemming y lematización (se ve que casi todas están reducidas a sus raíces).

Una vez finalizada la fase de **preprocesamiento,** se procedió a incluir al pipeline el mejor modelo que había resultado de la fase anterior del proyecto. En la iteración pasada, se probaron tres modelos de Machine Learning diferentes (Naïve-Bayes, OneVsRest y una red neuronal LSTM), bajo la métrica de la precisión. Se decidió usar dicha métrica puesto que uno de los mayores intereses del negocio es que los médicos puedan clasificar, sin invertir gran parte de su tiempo, los casos clínicos y los textos médicos que les llegan. Por tanto, debido a que se trata de cuestiones de salud, es importante que las clasificaciones sean

[1] Este es uno de los entregables del proyecto, el cual puede encontrarse en la siguiente ruta: https://github.com/Brenda-cbp/Proyecto1_fase2_BI/blob/main/notebooks/proyecto1_fase2_datos_preprocesados.csv

tan precisas como sea posible, con el fin de que los pacientes sean tratados por su enfermedad y no por alguna otra condición que no padezcan y que el algoritmo erróneamente haya detectado.

Ajustando los hiperparámetros de los algoritmos, se obtuvo que el algoritmo que arrojaba la mejor precisión para todas las clases era la LSTM. Por lo tanto, fue este el que decidimos poner en producción. Es importante resaltar que elegimos solamente el mejor modelo pues a los médicos realmente les interesa tener los mejores resultados y conceptos (no tres diferentes), particularmente porque es muy posible que las técnicas de Machine Learning sean desconocidas para ellos. Lo único que quieren lograr es clasificar adecuadamente sus textos.

Volviendo al modelo elegido, la LSTM (Long-Short Term Memory) es un tipo de red neuronal recurrente (RNN, por sus siglas en inglés) que se desempeña mejor que las RNN tradicionales en términos de memoria [1]. Una RNN es un tipo de red neuronal que permite a las salidas de capas previas ser utilizadas como entradas, teniendo estados ocultos [2]. Las LSTM tienen múltiples capas ocultas. A medida que se pasa a través de una capa, la información relevante se mantiene y la irrelevante se desecha en cada neurona (celda) individual [1]. Asimismo, las LSTM solucionan el problema de desvanecimiento de gradientes que las RNN enfrentan a menudo. El módulo de scikit-learn no implementa redes neuronales; por lo tanto, se utilizaron keras y tensorflow. Aunque son módulos muy interesantes, representaron algunos desafíos grandes (que pudieron ser solucionados) para la implementación de pipelines de scikit-learn.

La primera parte de la que debíamos encargarnos era la vectorización de los datos, la cual corresponde al embedding para los modelos de redes neuronales. De acuerdo con el ajuste de hiperparámetros del modelo anterior, la mejor opción de embedding fue BioSentVec: un módulo de Zhang et. Al. [3] que genera vectores de 700 dimensiones de los datos recibidos a partir de un modelo pre-entrenado sobre varias bases de datos médicas. De forma similar al preprocesamiento, se definió la clase `VectorizeLSTM()` para cargar el modelo pre-entrenado de BioSentVec y embeber los textos resultado de la fase de preprocesamiento en los vectores. Esta clase también se encarga de ajustar la forma de los datos para ser posteriormente recibidos por la red neuronal. La clase definida se encuentra en el **anexo 1** de este informe. Una vez finalizada esta fase, se agregó el paso al pipeline definido.

Finalmente, únicamente restaba la creación del modelo de Keras. De acuerdo con el ajuste de hiperparámetros del laboratorio anterior, el modelo de mejor desempeño era aquel que tenía dos capas LSTM, cada una de 64 neuronas, seguidas de sus capas de Dropout (i.e. pérdida de información) con un factor de apenas 0.1. La red tiene además una capa densa final de 5 neuronas (la cantidad de tipos de enfermedad) con función de activación Softmax. Este fue el mayor desafío de todo el proyecto, pues debía embeberse un modelo de Keras dentro de un clasificador de scikit-learn. Para ello, lo primero que se hizo fue crear el modelo en una clase, como se ve en el recuadro rojo de la figura 5 y, posteriormente, agregarlo al pipeline (recuadro azul) en el “envoltorio” KerasClassifier de scikit-learn (recuadro verde):

```
class LSTMBuilder():
    def __call__(self):
        output=5
        model = Sequential(name="LSTM")
        # Agregamos una capa LSTM con el tamanio de entrada de los embedded abstract
s y 64 neuronas en la capa
        model.add(LSTM(units=64, return_sequences=True,
                       input_shape=(1, 700)))
        model.add(Dropout(0.1))
        # Agregamos una segunda capa LSTM con 16 neuronas
        model.add(LSTM(units=64, return_sequences=False))
        # Con su respectiva capa de dropout
        model.add(Dropout(0.1))
        # Definimos la capa de salida
        model.add(Dense(output, activation='softmax'))
        # Compilo
        model.compile(loss='categorical_crossentropy', optimizer='adam',
                      metrics=[keras.metrics.Precision(name='precision')])
        return model
```

```
: 1 clf = KerasClassifier(LSTMBuilder(), verbose=1)
```

```
: 1 pipe.steps.append(('model', clf))
```

Imagen 5: Construcción del modelo usando dos capas LSTM, con sus respectivas capas de Dropout, y la capa densa con activación Softmax. Este modelo es envuelto en un *wrapper* de scikit-learn, denominado Keras Classifier, para ser agregado al pipeline.

Posterior a esto, el modelo fue ajustado a los datos (también se estudiaron algunas estadísticas – remítase al notebook para ver ello -) y, finalmente fue **persistido**. Debido a que se tiene un modelo de Keras embebido en un pipeline, era necesario guardar ambos archivos por aparte. Primero, se almacena el modelo de Keras en un archivo .h5, como se ve en la línea 1 de la figura 6. Posteriormente, se elimina el modelo del pipeline (línea 2) y, finalmente, se guarda el pipeline (sin el paso del modelo) en un archivo pickle (.pkl) en la línea 3. Usamos pickle en lugar de joblib para evitar problemas con la serialización del modelo. La idea es que, al leer el archivo en el API, se cargue primero el pipeline y, posteriormente, se agregue el modelo de Keras almacenado en el archivo .h5, para que funcione correctamente.

```
1 pipe_elegida.named_steps['model'].model.save('./assets/keras_model.h5')
2 pipe_elegida.named_steps['model'].model = None
3 dump(pipe_elegida, './assets/modelo.pkl')
```

Imagen 6: Almacenamiento del modelo de Keras (línea 1) y del pipeline (línea 3). Note que el modelo se elimina del pipeline antes de almacenarse para evitar errores de carga y serialización con los módulos de scikit-learn y keras.

Finalmente, con todos los pasos descritos previamente, el pipeline quedó como lo indica la figura 7:

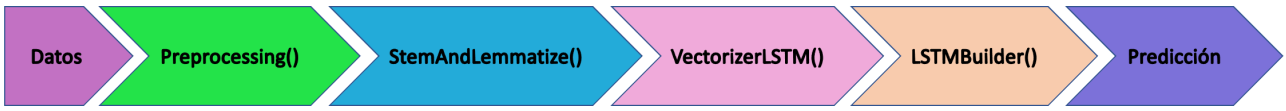

Imagen 7: Ilustración con el pipeline final construido.

Con el modelo almacenado, únicamente restaba desplegarlo para ser accedido por medio de un API. Para este fin, se utilizó FastAPI, el cual es un *framework* de Python que permite crear APIs de manera sencilla. En este caso, se desplegó un único endpoint con la información que se consideraba brindaba mayor valor a los médicos, en este caso: la categoría con la cual el modelo clasificó al texto, y la probabilidad asociada a que dicho texto pertenezca a cada una de las clases consideradas. Esta última información puede ser de especial importancia pues, en los casos en los que más “se confunde el modelo” (i.e. en la categoría 5, como se vio en la entrega pasada), el médico pueda ver rápidamente una segunda categoría a la cual el texto podría pertenecer también.

Para el despliegue de esta API se crearon varios módulos necesarios para su funcionamiento, que en el repositorio se encuentran dentro de la carpeta “back”. El primero de ellos corresponde a “clases.py”, que cuenta con las dependencias y funciones propias

necesarias para realizar toda la etapa de preprocesamiento de textos que lleguen a la API y que necesitará el modelo, como “remove_non_ascii”, “to_lowercase” o “remove_punctuation”. Con todo lo anterior se crea una clase llamada “Preprocessing” que estará encargada de realizar el preprocesamiento de los textos. Siguiendo con la lógica del pipeline descrito anteriormente, se crean también en este archivo las clases “StemAndLemmatize”, “VectorizeLSTM” y “LSTMBuilder” que cuentan con las funciones necesarias para cumplir con las etapas descritas en el pipeline mencionado.

En la misma carpeta “back” se creó también el archivo dataModel.py, que cuenta con la clase DataModel. Esta clase corresponde al tipo de datos en el que serán mapeados los datos recibidos desde el FrontEnd. Este paso se realiza para que un objeto de esta clase pueda ser analizado por el modelo. La clase DataModel tendrá los mismos atributos que el dataset con el que se entrenó el modelo, que en este caso corresponde únicamente a ”medical_abstracts”

Con estos módulos se realizó el despliegue de esta API, la cual puede ser accedida por la aplicación FrontEnd a través de un endpoint, cuya función más importante es recibir peticiones HTTP POST en la ruta “/predict” donde desde la aplicación recibirá un objeto de tipo DataModel. Una vez se reciba la petición, se crea un dataframe con los datos y se carga el pipeline guardado previamente como “modelo.pkl” y se agrega el modelo de Keras necesario para su funcionamiento. Teniendo el modelo en los pasos del pipeline, se procede a predecir las probabilidades de pertenecer a cada categoría de enfermedades con el modelo. Una vez terminado, se retorna a la aplicación el resultado, que consiste en el nombre de la categoría con mayor probabilidad según el modelo, y las probabilidades de pertenecer a las demás categorías, para que esta información pueda ser desplegada en la página web.

```python
@app.post("/predict")
def make_predictions(dataModel: DataModel):
    df = pd.DataFrame(dataModel.dict(), columns=dataModel.dict().keys(), index=[0])
    df.columns = DataModel.columns()
    model = load("../notebooks/assets/modelo.pkl")
    # Agrego el modelo de Keras que almacene previamente
    model.named_steps['model'].model = load_model('../notebooks/assets/keras_model.h5')
    result = model.predict_proba(df)
    # Sumo uno a la prediccion porque el modelo retorna como si las clases fueran de 0 a 4, no de 1 a 5.
    lista = (np.argmax(result)+1).tolist()
    json_prediccion = json.dumps(lista)
    # Ahora, envio todas las probabilidades en otro campo. Es el campo 0 para no enviar doble lsita
    lista_2 = result.tolist()[0]
    json_probabilidades = json.dumps(lista_2)
    return {"predict": json_prediccion, "probabilities": json_probabilidades}
```

Imagen 8: Función encargada de atender las peticiones POST de la API. Se encuentra en el archivo main.py de la carpeta back

Desarrollo de la aplicación y justificación

Para poder dejar el modelo a disposición de un usuario, la API desplegada debía conectarse con una aplicación web tal que se pudiera interactuar con los resultados del modelo. No obstante, antes de ello, es importante reconocer los usuarios y roles de la organización que usarán la aplicación.

Debido a que el modelo que construimos busca asesorar al personal médico en el diagnóstico de pacientes (sobre todo en situaciones en las que hay mucha demanda de los servicios de salud), consideramos que la aplicación podría ser usada por dos usuarios/roles de negocio.

a) Secretario/a, interno/a y/o residente: este usuario, al asistir al doctor o médico principal en sus labores diarias, sería el encargado de poner el texto en la aplicación y esperar a la obtención de las clasificaciones para que el médico pueda interpretarlos después. La idea sería que, usando los sistemas de bases de datos propios del hospital, las historias clínicas tengan un nuevo campo para guardar las posibles predicciones del algoritmo, tal que el médico pueda accederlas posteriormente. La ventaja de esto es que el doctor no se tendría que encargar de subir los textos directamente, sino de interpretar los resultados obtenidos por el modelo, y reducir el espectro de enfermedades particulares a diagnosticar (dentro de las categorías que el modelo concibe). Esta ventaja implica un ahorro de tiempo para el doctor en dos momentos: El primero, el ingreso de la historia clínica y espera de respuesta del modelo / herramienta. El segundo, facilidad y rapidez para realizar un diagnóstico final apoyado en los resultados obtenidos.
b) Médico: El médico podría utilizar la aplicación y sus resultados en dos circunstancias particulares. Primero, en ausencia de un asistente (bien sea un/a secretario/a o estudiante de medicina), el médico mismo podría subir el texto del cual quiere conocer una clasificación en particular. Este no es el escenario ideal, puesto que la idea es justamente reducir el tiempo que el doctor gasta clasificando textos, y sería más útil si ya tuviera a su disposición los que subió su secretario/a, interno/a o residente. El descrito anteriormente sería el segundo escenario: los textos han sido previamente clasificados por el asistente del doctor y este solamente debe encargarse de ver los diagnósticos, corroborarlos (en caso de ser necesario), hacer un diagnóstico más preciso en una categoría del algoritmo (e.g. si el paciente tiene apendicitis, gastroenteritis o ileítis cuando se diagnostica una enfermedad del sistema digestivo), recetar a los pacientes para sus respectivas enfermedades o remitir a un paciente determinado a otro especialista, de acuerdo con la categoría predicha por el modelo.

De esta forma, el usuario (sea el asistente del doctor o él/ella mismo/a) podrá subir un texto de un paciente del que se quiera conocer un diagnóstico, limitar una serie de posibles enfermedades para un paciente (i.e. reducir el espectro de enfermedades a un único grupo – por ejemplo enfermedades neurológicas o del cardiovascular -), o acelerar el diagnóstico de pacientes en momentos en que los servicios de salud (específicamente de urgencias) puedan encontrarse saturados, con el fin de “despachar” casos clínicos rápidamente, pero con el respaldo de una inteligencia artificial. El proceso de uso de la aplicación por parte de ambos roles es equivalente: Primero, se abre la aplicación. Segundo, se carga el texto del que quiere conocerse un diagnóstico. Tercero, se espera a que el modelo realice su predicción. Cuarto, se visualiza la predicción del modelo y la probabilidad de que una enfermedad en particular pertenezca a una clase determinada. En unión con procesos de negocio preexistentes, se propone que las predicciones sean almacenadas en la base de datos de las historias clínicas para su posterior lectura por parte de otros médicos, o para su consulta cuando sea de interés.

Ahora, nos interesa conocer la conexión entre la aplicación desarrollada y los procesos de negocio propios del hospital. Lo primero es que la aplicación busca facilitar tres procesos a los médicos del hospital, en particular en cuanto a sus funciones de diagnóstico de pacientes. En este orden de ideas, el proceso de negocio que se busca apoyar es el diagnóstico de las enfermedades de los pacientes que lleguen al hospital. Particularmente, esta aplicación apoyaría los siguientes procesos para los médicos:

1. El más importante de todos: brindar diagnósticos de pacientes. El proceso principal que busca apoyar nuestro sistema es el diagnóstico de enfermedades a partir de historias clínicas. Con los diagnósticos, muchas veces los médicos tendrían que

limitarse únicamente a corroborar los resultados (en ciertos casos) y recetar los medicamentos apropiados para cada enfermedad.

2. Reduce el tiempo de lectura y diagnóstico de los casos clínicos. La razón principal por la que se propuso este problema es que a los médicos les consume muchísimo tiempo el hecho de estudiar y analizar historias clínicas todos los días, cuando les ayudaría por lo menos tener una herramienta que, a priori, clasificara las enfermedades (al menos en familias grandes – como neurológicas o gastrointestinales -) por ellos. Esto sería particularmente útil en casos en que los servicios médicos de urgencias están llenos (o, incluso, colapsados). En estas ocasiones, usar el modelo en cuestión permitiría a los médicos "despachar" una gran cantidad de pacientes, con diagnósticos certeros, o por lo menos reducir el trabajo de tener que leer muchas historias clínicas y sólo tener que determinar una enfermedad particular (e.g. migraña) en un grupo de enfermedades predichas por el modelo (e.g. enfermedades del sistema nervioso).
3. Reducir el espectro de enfermedades a considerar para un paciente: usando nuestro modelo, el médico podrá obtener un grupo de enfermedades para poder realizar un diagnóstico más preciso. Como se dijo previamente, si el modelo predice enfermedades del sistema digestivo, el médico puede concentrarse en buscar patologías asociadas a dicho sistema (e.g. apendicitis, peritonitis, etc.) y dejar de preocuparse por buscar enfermedades de otros sistemas (e.g. cardiovascular o del sistema nervioso).
4. Para médicos generales e internos: remitir a los pacientes a otro tipo de especialista, de acuerdo con la categoría que haya sido obtenida con el modelo, para el tratamiento del paciente. Esto también es útil en los contextos donde los servicios médicos están colapsados y se necesita saber qué especialista debería atender cada caso.

Con esto, puede decir que la aplicación construida es fundamental para el médico: le permitirá ahorrar tiempo que actualmente consume leyendo historias clínicas y obtener diagnósticos precisos para las enfermedades de sus pacientes. Asimismo, nuestra aplicación no solo brinda el diagnóstico más preciso para un texto médico en particular, sino que además le da al médico la probabilidad de que el texto pertenezca a cualquier otra de las categorías definidas. Esto es particularmente útil en el caso de las patologías generales. Nuestro modelo tiende a confundirse en esta categoría debido a que las palabras pueden pertenecer a otra de las categorías contempladas en el modelo. No obstante, conociendo la probabilidad, los médicos pueden tener una "segunda idea" de cuál es el diagnóstico que debe darse a un paciente en particular, si el primero no le convence en su totalidad. Es así como este modelo es fundamental para ahorrar tiempo del médico (el propósito principal de su construcción), brindando diagnósticos precisos para las enfermedades de los clientes.

Por último, la figura 9 muestra un resumen de la interacción de los roles (médicos y/o asistentes) con la aplicación y su utilidad para el hospital y sus funcionarios:

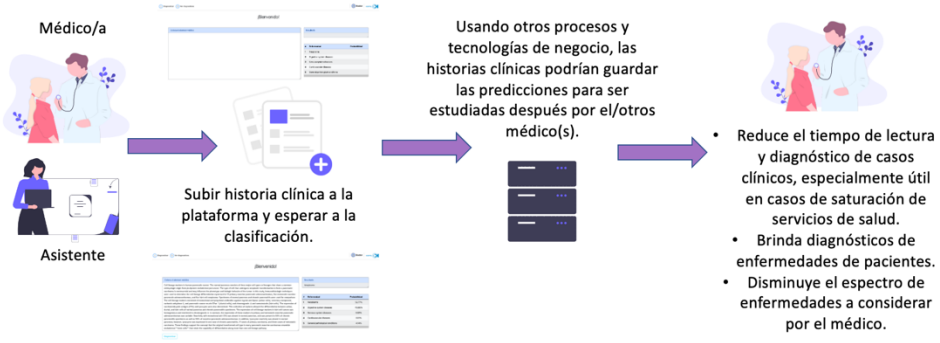

Imagen 9: Interacción de los roles (médicos, asistentes) con la aplicación, y beneficios para los médicos.

Trabajo en equipo.

a. **Roles:**
   I. Sofía Álvarez: Líder de proyecto e Ingeniera de datos
   II. Brenda Barahona: Ingeniera de software responsable del diseño de la aplicación y resultados
   III. Alvaro Plata: Ingeniero de software responsable de desarrollar la aplicación final

b. **Retos enfrentados en el proyecto**
   a. Nuestro principal reto fue colocar el modelo en un pipeline. Este reto se especificó anteriormente en el proceso de automatización y de la preparación de los datos. En resumen, nosotros usamos un modelo de Keras, el cual debíamos embeber dentro de un clasificador de Scikit-Learn. Para dar solución a esto, se creó el modelo en una clase y este fue agregado por medio de KerasClassifier de scikit-learn
   b. Otro desafío fue desplegar el modelo en FastApi. Se nos presentaban errores de serialización ya que teníamos funciones o transformaciones personalizadas para la realización del modelo. Para solucionar esto colocamos las mismas clases en el back y en el notebook.

c. **Tareas realizadas:**
   a. Conjunto de datos resultado de la fase de entendimiento y preparación de los datos: Brenda, Sofía y Álvaro – 2.5 horas
   b. Preparación del pipeline con el modelo listo: Brenda, Sofía y Álvaro – 10 horas.
   c. Desarrollo de la aplicación (Backend, Frontend y conexión entre estas): Brenda, Sofía y Álvaro -8 horas
   d. Video: Brenda, Sofía y Álvaro – 0.5 horas
   e. Informe y presentación: Brenda, Sofía y Álvaro – 2.5 horas

d. **Repartición de puntos:**

Si tuviéramos que repartir 100 puntos entre los integrantes del grupo, repartiríamos 33.33 a cada uno.

e. **Puntos para mejorar en la siguiente entrega:**
   a. Manejo y distribución del tiempo. Distribuir el trabajo en el tiempo de manera más homogénea, para no tener una gran cantidad de trabajo por hacer poco tiempo antes de la fecha de entrega.
   b. Antes de empezar el proyecto, tener claridad sobre los roles, las actividades que cada uno debe hacer y las actividades que haremos todos en grupo. Esto nos permitirá un mejor enfoque al momento de ejecutar el proyecto y nos permitirá trabajar más en paralelo

f. **Bono de expertos:**
   a. Realizamos una reunión inicial con los expertos. Con ellos pudimos conocer más sobre los tipos de enfermedades que había y lo que implicaban estas. Adicional a esto, les explicamos el proyecto y los adelantos que teníamos. Les explicamos nuestra métrica y que significaba, además, les comentamos que

teníamos problemas con la clasificación con las enfermedades de tipo 5, esto debido a que este conjunto de enfermedades es muy general. Un consejo que ellos nos dieron fue que, si el modelo lanzaba la predicción de una categoría tipo 5, se mostrara también las probabilidades de los otros 4 tipos de enfermedades. Esto con el objetivo de dar más información al doctor para que tenga una segunda opción en caso de que la enfermedad del paciente no clasifique como tipo 5.

Conclusiones:

- Evidenciamos una gran oportunidad de mejora en el proceso de diagnóstico gracias al uso de inteligencia artificial para crear una herramienta que permita generar un prediagnóstico en base al historial médico del peciente, reduciendo así opciones de enfermedades y facilitándole el proceso de detección al médico
- Hay que tener en cuenta que la herramienta no busca reemplazar la función de diagnóstico de los médicos, pero si les puede brindar un soporte para que esta tarea sea más rápida y sencilla
- Podemos ver que la herramienta se puede integrar fácilmente con los modelos ya existentes del negocio, los cuales usan otras tecnologías.
- El impacto de la aplicación no solo se vería en la efectividad de los médicos al hacer el diagnóstico, si no en la rapidez de sacar un prediagnóstico por parte de los internos o residentes.

Anexos:

Anexo 1: Definición de la clase VectorizeLSTM():

```
class VectorizeLSTM():
    def __init__(self):
        pass
    def transform(self,X,y=None):
        # Union de todos los tokens.
        X["medical_abstracts"] = X["medical_abstracts"].apply(lambda x: ' '.join(map(str, x)))
        # Inicializacion del modelo para realizar el embedding.
        model_path = 'BioSentVec_PubMed_MIMICIII-bigram_d700.bin'
        model = sent2vec.Sent2vecModel()
        try:
            # Carga del modelo.
            model.load_model(model_path)
            print('Model successfuly loaded')
        except Exception as e:
            print(e)
        # Embebiendo las frases de los datos como vectores de 700 dimensiones bajo el modelo cargado.
        new_X = model.embed_sentences(X["medical_abstracts"])
        # Ajuste de la forma de los datos para ser recibidos por la red neuronal.
        new_X = new_X.reshape(-1, 1, new_X.shape[1])
        return new_X
    def fit(self, X, y=None):
        return self
```

Anexo 1: Esta clase carga el modelo pre-entrenado para generar vectores de palabras de dimensionalidad 700, y ajusta los datos obtenidos a la forma requerida para la entrada de la red neuronal.

Bibliografía:

[1] https://www.analyticsvidhya.com/blog/2021/06/lstm-for-text-classification/

[2] https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks

[3] Zhang Y, Chen Q, Yang Z, Lin H, Lu Z. BioWordVec, improving biomedical word embeddings with subword information and MeSH. Scientific Data. 2019.