TP2: Story Points - Grupo 2

Célestine Raveneau, Florian Escaffre, Juan Gomez, Luis Condori

Introducción

En este trabajo práctico el objetivo es de entrenar varios modelos a predecir un número de story point asignado a una user story, es decir la complejidad de la tarea.
Tenemos dos datasets

- Train: uno etiquetado y que vamos a utilizar para desarrollar nuestro modelos
- Test: uno no etiquetado del cual intentaremos predecir los *story points*.

Ambos conjuntos de datos tienen para cada user story, un título y una descripción en lenguaje natural y una variable categórica del proyecto al cual la user story pertenece.

Descripción de columnas

- title: Título de la user story.
- description: Descripción de la user story.
- project: Nombre del proyecto en el cual esa user story se crea.
- storypoint: Puntaje (nivel de complejidad) de la user story.

En total tuvimos que desarrollar 5 modelos

- Bayes Naive
- Random Forest
- XGBoost
- Red Neuronal
- Ensamble

Preprocesamiento:

El dataset no cuenta con valores faltantes, raros, tampoco se encontró outliers significativos.

Las tareas de preprocesamiento son vitales para el entrenamiento de modelos, ya que ayudan a reducir la dimensionalidad de las columnas y eliminar el ruido (como signos de puntuación, stopwords, URLs, etc.). Esto mejora la calidad de los datos, permitiendo que el modelo se enfoque en información relevante y, por lo tanto, mejore su rendimiento.

Preparar y limpiar los textos en las columnas **"title"** y **"description".** Se creó una función “clean_text” usando la librería **SpaCy**.

- Eliminación de múltiples espacios
- Eliminación de bloques de código y contenido entre llaves.
- Transformación a minúsculas.
- Tokenización y Lematización:
- Filtrado de tokens:
    - Espacios: `token.is_space`
    - Signos de puntuación: `token.is_punct`
    - Números: `token.like_num`
    - Palabras comunes (stopwords): `token.is_stop`
    - URLs y correos electrónicos: `token.like_url`, `token.like_email`
    - Palabras muy cortas (menos de 3 caracteres): `len(token) > 2`

Pipeline: En la mayoría de los modelos siguientes se construyen utilizando un **pipeline**, que asegura que los pasos de preprocesamiento (como la vectorización y codificación) y el modelo se entrenen y evalúen de manera consistente.

- ('preprocessor', preprocessor): Primero, se aplica una serie de transformaciones a los datos mediante un objeto `ColumnTransformer`. Estas transformaciones incluyen la vectorización de texto y la codificación de variables categóricas.
- ('model', Modelo()): Luego se aplica el modelo (ej clasificador Naive Bayes).

Dentro del preprocesamiento, se lleva a cabo una optimización de los **hiperparámetros** del `TfidfVectorizer`, y del modelo involucrado, que es el encargado de convertir los textos en vectores numéricos. Los hiperparámetros que se ajustan son:

- max_features: Número máximo de características (palabras) a considerar en el modelo.
- max_df y min_df: Controlan la frecuencia mínima y máxima de los términos, respectivamente. Filtran palabras demasiado comunes o demasiado raras.
- ngram_range: Permite capturar secuencias de palabras (n-gramas) en lugar de solo palabras individuales. El rango (1, 1) captura solo palabras individuales, mientras que (1, 2) captura tanto palabras como combinaciones de dos palabras.
- stop_words: Se puede especificar un conjunto de palabras que deben ser ignoradas (como "the", "and", etc.).
- binary: Si es True, convierte las características en valores binarios (1 si la palabra está presente, 0 si no lo está).

Si no se realiza la limpieza de datos con la función “clean_text”, ni se limita el hiperparemetro max_features la cantidad de columnas asciende a 41533, generando que los modelos basados en árboles demoren en una única iteración entre 8 min y 12 min.

Entrenamiento y Validación: se emplea KFold Cross Validation cambiando el número de particiones en cada modelo para disminuir el tiempo de entrenamiento.

- **KFold Cross Validation**: Utiliza “n” particiones para validar el modelo, asegurando que se entrenen y validen los modelos en diferentes subconjuntos de los datos para obtener una evaluación más robusta.

Descripción de Modelos

Naive Bayes (MultinomialNB):

Modelo Naive Bayes (MultinomialNB): El **Naive Bayes Multinomial** es un clasificador probabilístico basado en el teorema de Bayes, que asume que las características (en este caso, las palabras en el texto) son independientes entre sí.

Hiperparámetros para MultinomialNB():

- alpha: Suavizado de Laplace. Controla la suavización de las probabilidades.

- fit_prior: Si debe ajustar las probabilidades a priori. Si es True, se ajustan las probabilidades de clase, de lo contrario, se asume una distribución uniforme de las clases.

```
Best RMSE: 2.8197174864304535

Best hyperparameters: {'preprocessor__title_bow__stop_words':
'english', 'preprocessor__title_bow__ngram_range': (1, 1),
'preprocessor__title_bow__min_df': 2,
'preprocessor__title_bow__max_features': 100,
'preprocessor__title_bow__max_df': 1.0,
'preprocessor__title_bow__binary': False,
'preprocessor__description_bow__stop_words': None,
'preprocessor__description_bow__ngram_range': (1, 1),
'preprocessor__description_bow__min_df': 3,
'preprocessor__description_bow__max_features': None,
'preprocessor__description_bow__max_df': 0.9,
'preprocessor__description_bow__binary': True, 'model__fit_prior':
True, 'model__alpha': 1000}

TRAIN RMSE: 2.823980236497805
TEST RMSE : 2.8346857035612754
```

Random Forest

Random Forest Regressor: El modelo **Random Forest** es un algoritmo basado en un conjunto de árboles de decisión, que mejora el rendimiento mediante el ensamblaje de múltiples árboles y la votación sobre las predicciones de cada uno.

Hiperparámetros de Random Forest:

- **`n_estimators`**: Especifica el número de árboles en el bosque. Cuantos más árboles, mayor será la capacidad del modelo para generalizar, aunque también aumentará el tiempo de entrenamiento.
- **`max_depth`**: Controla la profundidad máxima de cada árbol, limitando el número de nodos en cada árbol. Limitar la profundidad puede ayudar a evitar el sobreajuste.
- **`min_samples_split`**: Especifica el número mínimo de muestras necesarias para dividir un nodo. Aumentar este valor puede reducir el sobreajuste.
- **`min_samples_leaf`**: Establece el número mínimo de muestras requeridas para ser una hoja en el árbol. Aumentar este valor puede evitar que los árboles se ajusten demasiado a los datos de entrenamiento.
- **`ccp_alpha`**: Es un parámetro de complejidad que controla la poda de los árboles, reduciendo la complejidad del modelo y evitando el sobreajuste.

```
Mejor RMSE: 2.5581231484256706
```

```
Mejores hiperparámetros: {'preprocessor__title_bow__stop_words':
'english', 'preprocessor__title_bow__ngram_range': (1, 1),
'preprocessor__title_bow__max_features': 300,
'preprocessor__description_bow__stop_words': 'english',
'preprocessor__description_bow__ngram_range': (1, 1),
'preprocessor__description_bow__max_features': 5000,
'model__n_estimators': 200, 'model__min_samples_split': 2,
'model__min_samples_leaf': 4, 'model__max_depth': 30,
'model__ccp_alpha': 0.0}

RMSE en el conjunto de train: 1.8038251538330836

RMSE en el conjunto de prueba: 2.688588059735108
```

XGBRegressor

XGBoost Regressor: XGBoost es un algoritmo de **Gradient Boosting** muy eficiente que se utiliza para problemas de regresión y clasificación. En lugar de entrenar un único árbol de decisión, construye múltiples árboles secuenciales donde cada árbol intenta corregir los errores de los anteriores. Además, XGBoost incluye regularización (a través de `alpha` y `lambda`) que ayuda a prevenir el sobreajuste.

Hiperparámetros de XGBRegressor:

- **`n_estimators`**: Número de árboles en el modelo. Un mayor número de árboles puede mejorar la precisión, pero también aumenta el riesgo de sobreajuste y el tiempo de entrenamiento.
- **`max_depth`**: Profundidad máxima de cada árbol. Limitar la profundidad de los árboles puede evitar que el modelo se sobreajuste a los datos de entrenamiento.
- **`learning_rate`**: Tasa de aprendizaje que controla cuánto afectan los nuevos árboles a la predicción final. Valores más pequeños requieren más árboles para obtener un buen rendimiento.
- **`subsample`**: Proporción de muestras utilizadas para entrenar cada árbol. Esto puede ayudar a evitar el sobreajuste y mejorar la generalización.
- **`colsample_bytree`**: Proporción de características a usar para entrenar cada árbol. Esto también puede ayudar a evitar el sobreajuste y mejorar la variabilidad de los árboles.
- **`gamma`**: Parámetro de regularización que controla la complejidad de los árboles. Un valor mayor restringe más la construcción de los árboles.
- **`alpha`** y **`lambda`**: Son términos de regularización L1 y L2, respectivamente, que ayudan a controlar el sobreajuste penalizando los coeficientes de los árboles.

```
Mejor RMSE: 2.6930071931161126
```

```
Mejores hiperparámetros: {'preprocessor__title_bow__stop_words':
'english', 'preprocessor__title_bow__ngram_range': (1, 1),
'preprocessor__title_bow__max_features': 500,
'preprocessor__description_bow__stop_words': 'english',
'preprocessor__description_bow__ngram_range': (1, 2),
'preprocessor__description_bow__max_features': 1000,
'model__subsample': 0.7, 'model__n_estimators': 200,
'model__max_depth': 20, 'model__learning_rate': 0.01, 'model__lambda':
1, 'model__gamma': 0.2, 'model__colsample_bytree': 0.6, 'model__alpha':
0}

RMSE en el conjunto de train: 1.6637862023003536

RMSE en el conjunto de prueba: 2.5762766406793247
```

Red Neuronal

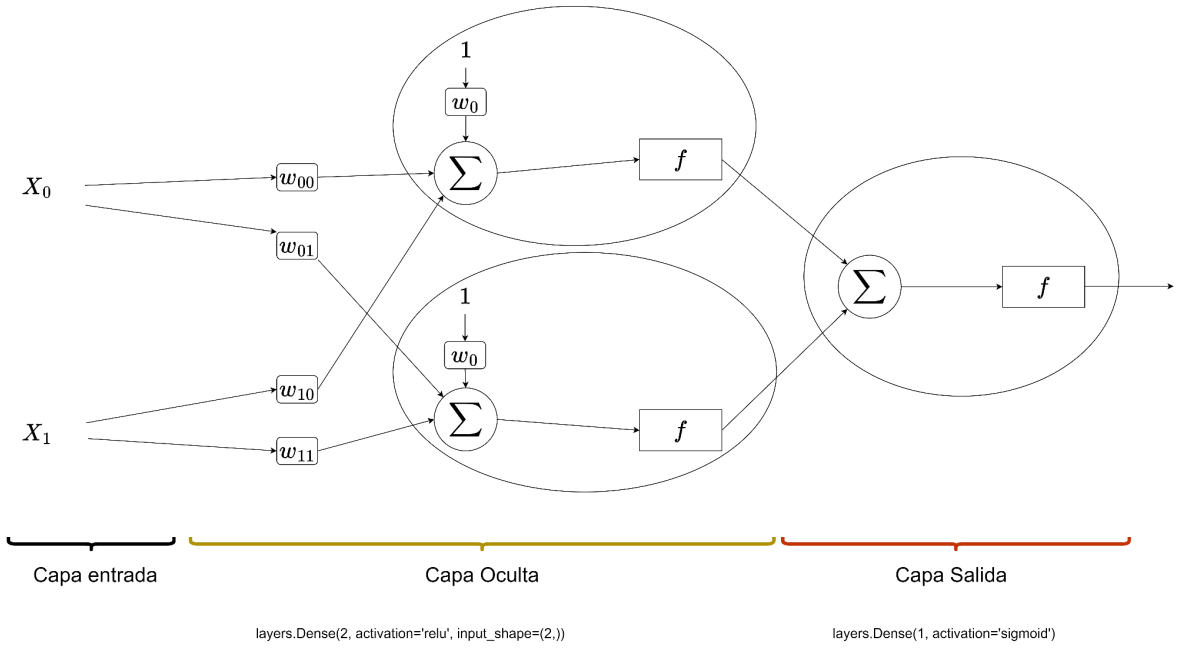

Fig 1: Red neuronal simple

Red Neuronal N1:

Definimos un modelo de red neuronal simple usando **Keras** de TensorFlow, con las siguientes características:

Arquitectura:

- Capa densa de 32 neuronas con activación `ReLU` y regularización L2 para evitar sobreajuste.
- Capa densa adicional con 16 neuronas, también con activación `ReLU`.
- Capa de salida con una sola neurona para regresión (predicción de `storypoint`).

Compilación:

- Se utiliza el optimizador **SGD** (Descenso por Gradiente Estocástico) con una tasa de aprendizaje de 0.001.

Entrenamiento:

- Se entrenó el modelo durante 20 épocas con un tamaño de lote de 16, utilizando 20% de los datos de entrenamiento para validación.
- Se evalúa el rendimiento del modelo en los conjuntos de prueba y entrenamiento utilizando métricas como MSE, RMSE, MAE y R^2.

Red 2:

Empleamos **Keras Tuner** para realizar la búsqueda de hiperparámetros. Los parámetros que optimizamos incluyen:

1. **Número de capas** (`num_layers`): Entre 1 y 5 capas ocultas.
2. **Unidades por capa** (`units_{i}`): Número de neuronas en cada capa (de 32 a 512).
3. **Función de activación** (`activation_{i}`): `ReLU` o `tanh`.
4. **Optimizador**: `Adam` o `SGD`.
5. **Tasa de aprendizaje** (`learning_rate`): 0.001, 0.01 o 0.1.

El proceso de búsqueda de hiperparámetros está basado en **Hyperband**, que realiza un ajuste eficiente explorando distintas combinaciones de estos parámetros. Además, se incorpora **EarlyStopping** para evitar el sobreajuste y detener el entrenamiento cuando no se observa mejora en la validación

```
Trial 76 Complete [00h 03m 21s]

val_root_mean_squared_error: 2.8543992042541504

Best val_root_mean_squared_error So Far: 2.588426351547241

Total elapsed time: 00h 35m 48s

Los mejores hiperparámetros son: {'num_layers': 5, 'units_0': 160,
'activation_0': 'relu', 'optimizer': 'sgd', 'learning_rate': 0.01,
'units_1': 480, 'activation_1': 'tanh', 'units_2': 416, 'activation_2':
'tanh', 'tuner/epochs': 4, 'tuner/initial_epoch': 2, 'tuner/bracket':
```

```
3, 'tuner/round': 1, 'units_3': 32, 'activation_3': 'relu', 'units_4':
32, 'activation_4': 'relu', 'tuner/trial_id': '0002'}
```

Ensamble

El **modelo combinado (ensemble)** está compuesto por tres modelos base: **RandomForestRegressor**, **XGBoost** y una **Red Neuronal**. Estos modelos se combinan utilizando un **Stacking Regressor**, donde las predicciones de los tres modelos base se utilizan como entrada para un modelo final (en este caso, **Ridge Regression**), que aprende a hacer las predicciones finales a partir de estas predicciones intermedias.

Modelos Base:

- **RandomForestRegressor**:
  - Un modelo de regresión basado en un conjunto de árboles de decisión. Se configuró con 300 árboles y una profundidad máxima de 20, lo que ayuda a que el modelo sea lo suficientemente complejo para aprender patrones, pero sin llegar a sobreajustarse.
- **XGBoost**:
  - **XGBoost** es un modelo de aprendizaje basado en árboles que utiliza el enfoque de boosting. Se configuró con el objetivo de minimizar el error cuadrático medio.
- **Red Neuronal (NN)**:
  - Se implementó una red neuronal simple con dos capas ocultas: una capa con 64 neuronas y una capa con 32 neuronas. Ambas capas usan la función de activación **ReLU** (Rectified Linear Unit), que es eficiente para evitar problemas como el desvanecimiento del gradiente. La capa de salida tiene una sola neurona sin activación (función lineal), ya que el objetivo es predecir valores continuos (storypoints).
  - El modelo de red neuronal se entrenó utilizando **Adam** como optimizador y el **MSE (Error Cuadrático Medio)** como función de pérdida, lo cual es común en tareas de regresión.

Stacking Regressor:

- **Stacking** es una técnica en la que los modelos base se entrenan de forma independiente, y sus predicciones se combinan para hacer una predicción final.
- **Meta-modelo (final_estimator)**: El modelo final que hace la predicción combinada es una **regresión Ridge**. Este es un modelo lineal que intenta predecir los **storypoints** a partir de las salidas de los tres modelos base. La regresión Ridge es útil para regularizar el modelo final y evitar el sobreajuste.

Entrenamiento y Evaluación:

- Los modelos base se entrenaron de manera independiente, y luego el Stacking Regressor combinó sus predicciones. La evaluación del rendimiento del modelo final

se realizó utilizando el **RMSE (Root Mean Squared Error)** y el **R² (Coeficiente de Determinación)**, dos métricas comunes para problemas de regresión.

Cuadro de Resultados

Los siguientes resultados son los mejores que se obtuvieron a lo largo del mes que duró la competencia.

Modelo	MSE	RMSE	R²	Kaggle
Bayes Naive	8.03	2.83	0.14	2.96
Random Forest	7.22	2.69	0.28	2.69
XG Boost	6.63	2.69	0.24	2.77
Red Neuronal	6.86	2.62	0,27	2.77
Ensamble 1 XGBoost + Adaboost + Naive Bayes	11.09	3.33	-0.18	3.31
Ensamble 2 RF + XGBoost + red neuronal	8.35	2.89	0.11	3.09

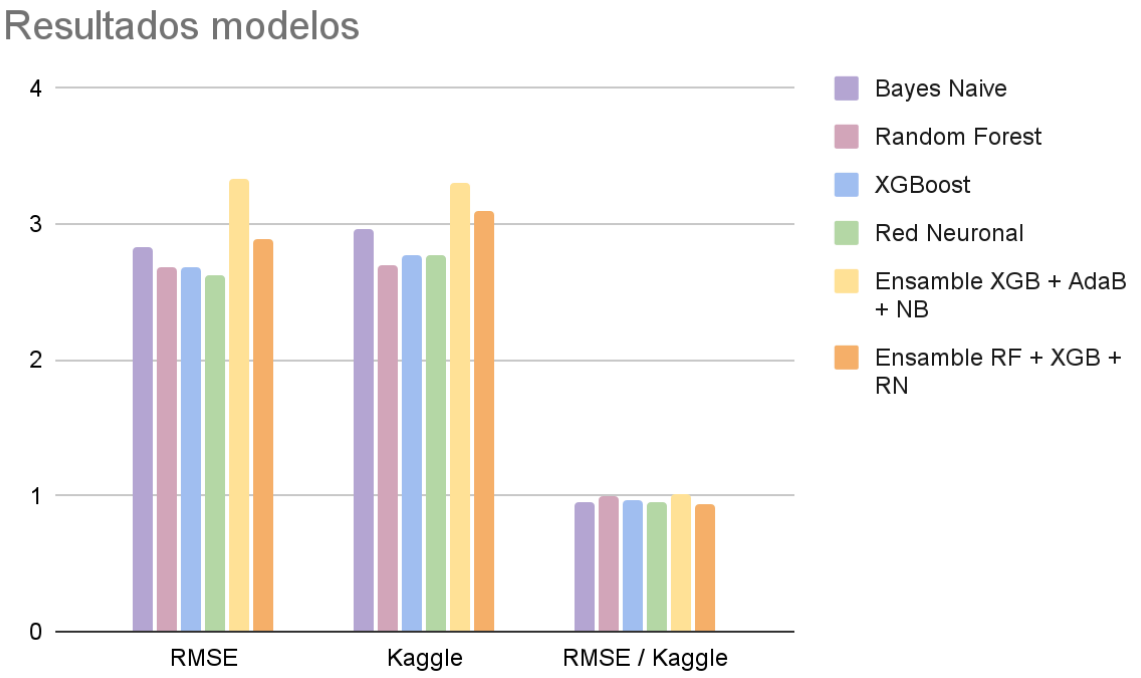

Fig 2: Resultados obtenidos.

Modelos Extras

En una fase inicial, se implementaron modelos de base (Regresión lineal, Perceptrón, Self-Organizing Map) para establecer un benchmark de desempeño. Estos modelos, aunque informativos, no fueron considerados para la solución final.

Posteriormente, se llevó a cabo una exhaustiva exploración de distintas técnicas de preprocesamiento de texto, incluyendo la eliminación de términos vacíos (stop words), puntuación, enlaces web (URLs), entre otros. Además, se evaluaron diferentes estrategias de vectorización, como CountVectorizer y TF-IDF, con el objetivo de representar el texto de manera numérica y adecuada para los modelos de aprendizaje automático.

La configuración final del modelo corresponde a la combinación de preprocesamiento y vectorización que arrojó los mejores resultados en las métricas de evaluación seleccionadas.

Conclusiones generales

A través de la implementación y evaluación de varios modelos, se logró identificar que el **Random Forest** fue el modelo más efectivo, destacándose tanto en el conjunto de test como en la competencia de Kaggle.

Las tareas de preprocesamiento fueron fundamentales para mejorar el rendimiento del modelo. La limpieza de los textos, que incluyó la eliminación de stopwords, puntuación y URLs, ayudó a reducir la dimensionalidad y a focalizarse en las características relevantes. Además, la optimización del **TfidfVectorizer** incrementó la eficiencia de la vectorización del texto, mejorando la precisión y reduciendo el tiempo de ejecución.

Aunque **XGBoost** y la **red neuronal** también mostraron buenos resultados, especialmente en términos de capacidad de modelar relaciones no lineales, el **Random Forest** demostró ser el más equilibrado en cuanto a precisión y eficiencia computacional. A pesar de su simplicidad, **Naive Bayes** no alcanzó el rendimiento esperado, lo que resalta la importancia de usar modelos más complejos para este tipo de tareas.

En resumen, el trabajo mostró que un buen preprocesamiento y la correcta selección de modelos son clave para obtener buenos resultados. Pero siempre hay margen para seguir mejorando los modelos, explorando diferentes técnicas de procesamiento de datos y de optimización de modelos. Es importante buscar un equilibrio entre efectividad y rapidez, ya que un modelo complejo puede mejorar el rendimiento pero aumentar los tiempos de ejecución.

Tareas Realizadas

El trabajo práctico tuvo una duración de 9 (nueve) semana.

Intégrante	Principales Tares Realizadas
Célestine Raveneau	• Optimisation de los hiper-parámetros del modelo de naive bayes

	• Escritura y optimización del modelo de red neuronal. • Optimización del modelo Random Forest. • Escritura de un ensamble que combina RF + XGBoost + red neuronal
Florian Escaffre	• Optimizar los hiperparametros del modelo de random forest • Armado un ensamble de 3 modelos (random forest, xgboost y red neuronal) y Celestine hizo uno con mejores performancias.
Juan Gomez	• Armado inicial de la Notebook con los 3 modelos inicial (naive Bayes, xgboost y random forest), sin buscar hiperparametros optimos, random forest • Perceptron • Modelo SOM (sin optimizar parámetros) • Armado del ensamble de 3 modelos (xgboost, adaboost y naive Bayes) con los hiperparametros optimos.
Luis Condori	• Naive bayes • Random Forest • XGBoost