Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en computación

Biblioteca Python para el apoyo al desarrollo de pipelines de procesamiento de datos con Spark

Autor:
D. Óliver L. Sanz San José

Universidad de Valladolid

Escuela de Ingeniería Informática

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática
Mención en computación

Biblioteca Python para el apoyo al desarrollo de pipelines de procesamiento de datos con Spark

Autor:
D. Óliver L. Sanz San José

Tutor:
D. Carlos E. Vivaracho Pascual
D.ª Arancha Simón Hurtado

And the men who hold high places
Must be the ones to start
To mould a new reality
Closer to the Heart

The Blacksmith and the Artist
Reflect it in their art
Forge their creativity
Closer to the Heart

Philosophers and Ploughmen
Each must know his part
To sow a new mentality
Closer to the Heart

You can be the Captain
I will draw the Chart
Sailing into destiny
Closer to the Heart

Rush, Closer to the Hearth (1977)

Agradecimientos

Gracias a mis padres, Lourdes y Jose Luis, por vuestro apoyo incondicional.
Gracias a mi abuela, Magdalena, por tu fuerza y tu sonrisa.
Gracias a Miguel, por guiarme y confiar en mi trabajo.
Gracias a Jimena, por estar siempre dispuesta a ayudar.
Por último, gracias a mi sensei, Jose, por transmitirme tu espíritu de auto superación.

Índice

Índice de figuras

Resumen

Esta es la memoria del trabajo de fin de grado presentado por Óliver L. Sanz en el Grado en Ingeniería Informática de la Universidad de Valladolid en Julio de 2018. En ella, se describe el contexto, motivación, así como el proceso de diseño, desarrollo y validación de una biblioteca escrita en el lenguaje de programación Python para el desarrollo de pipelines (secuencias) de tratamiento de datos con el motor de procesamiento Spark, desarrollada para la empresa Luce Innovative Technologies. Los objetivos de diseño de esta biblioteca son que sea sencilla de utilizar, de forma que facilite iteraciones rápidas en el proceso de análisis de datos; y extensible, de forma que pueda adaptarse a necesidades futuras. Posteriormente, también se describe en esta memoria una necesidad de tratamiento de datos, así como la forma de cubrir esta necesidad aplicando la biblioteca desarrollada.

Capítulo 1 - Introducción

En la actualidad asistimos a la llamada transformación digital en la que términos como Big Data (obtención y explotación de grandes cantidades de datos) y Machine Learning (aprendizaje automático) están adquiriendo gran importancia, gracias a la inmensa generación de datos en las redes sociales, apps, sensores, wearables, e-commerce, banca electrónica etc. Existe un gran interés para aprovechar estas ingentes cantidades de datos (Big Data) generados de forma automática durante años para obtener extraer valor de ellos, es decir, ventajas competitivas en el mercado.

Debido a que la generación de datos es tan grande, requerimos de nuevas tecnologías y herramientas para poder trabajar con ellos. No sólo el volumen supone un reto, la variedad que tienen y la velocidad con la que se necesitan consumir suponen retos también. En la actualidad una tecnología que permite trabajar con dicha complejidad es Apache Spark, un motor unificado y eficiente de Big Data Open Source muy popular y estándar de facto actualmente en lo referente al procesamiento de datos masivos.

Una vez que disponemos de motores como Spark para trabajar con datos masivos, el aprendizaje automático toma un papel relevante en la transformación digital como una de las principales piezas para extraer ese valor o ventajas competitivas que prometen las grandes cantidades de datos. Árboles de Decisión, K-means, Redes Neuronales, etc., son técnicas típicas de aprendizaje automático. Todas tienen en común que necesitan los datos de una forma determinada, es decir, en formato tabular. Además, es común que la información que estas técnicas de aprendizaje automático pueden aprovechar esté oculta en los datos, enterrada entre otras informaciones irrelevantes.

La preparación de los datos (extraerlos de las fuentes, limpiarlos, extraer de ellos las características adecuadas y adecuarlos al formato tabular) consume la mayor parte del tiempo de los ingenieros de datos y analistas de datos [1], por tanto, **es de gran interés encontrar o desarrollar herramientas que agilicen y optimicen dicho trabajo.**

1.1 - Motivación

En equipos de trabajo de minería de datos se suele utilizar la metodología CRISP-DM [2]. Esta metodología es muy similar a otras metodologías de desarrollo software como Scrum o Kanban en su filosofía subyacente de agilidad para cambiar, y de hecho, tiene fases cíclicas, como puede verse en la Figura 1.

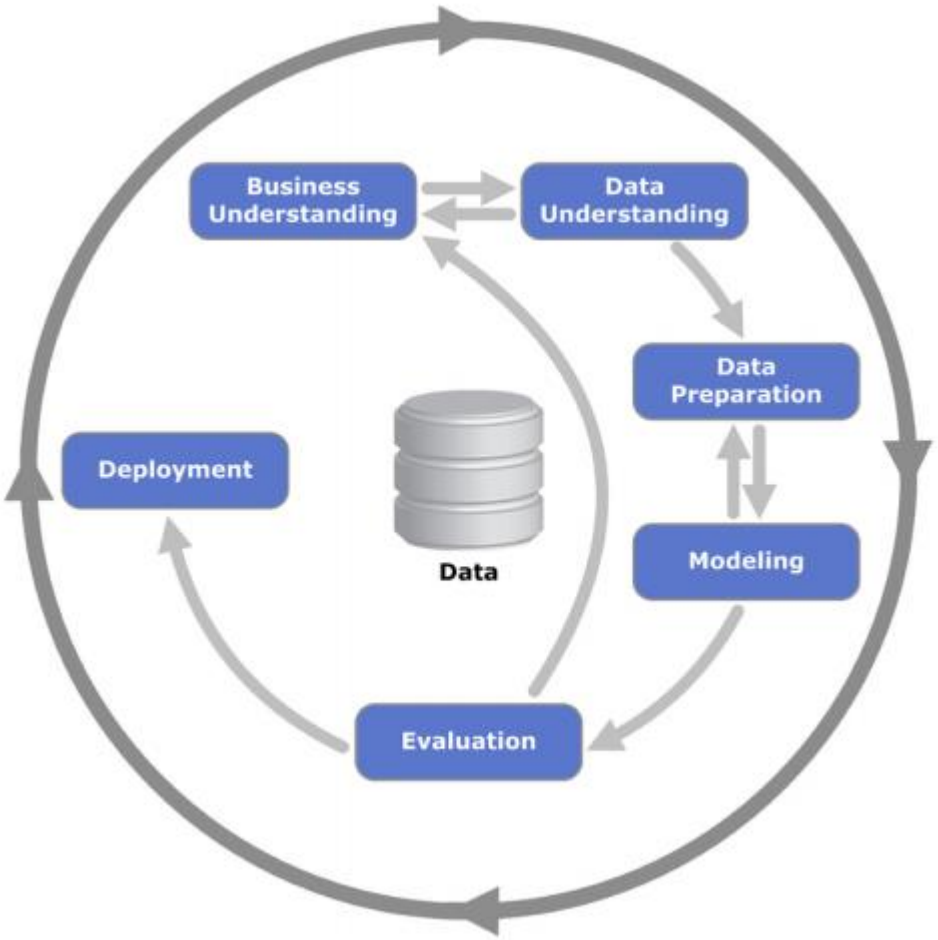


Figura 1: Fases de la metodología CRISP-DM [3]

En las primeras fases de proyecto de minería de datos, cuando son desconocidas las posibilidades que aportan, es posible que se cambien mucho los requisitos, redefinamos objetivos o incluso se desestimen muchas iniciativas propuestas con anterioridad. Es por esto que se requiere agilidad para tomar decisiones, y estas suelen estar soportadas por el análisis preliminar de los datos.

Para poder sacar conclusiones de los datos, en primer lugar hay que limpiarlos y ponerlos en un formato adecuado para el análisis. En algunos entornos de trabajo reales la forma de trabajar en las primeras fases es a partir de los llamados **notebooks** con Spark como motor de procesamiento. Los notebooks son documentos que incluyen celdas de código que pueden ser ejecutadas interactivamente, así como ecuaciones, visualizaciones y texto. Permiten el análisis interactivo de datos, y combinados con Spark permiten realizarlo sobre grandes cantidades de ellos.

Aunque los notebooks están preparados para el análisis ágil, no son adecuados para realizar procesados que requiere una gran cantidad de líneas de código. Esto es así porque los notebooks proveen poca funcionalidad para la gestión y reutilización del mismo, además de que la presencia de mucho código para el tratamiento de los datos perjudica la usabilidad del notebook y la legibilidad de los resultados del análisis que lleva a cabo.

Otro problema recurrente en el entorno del análisis de datos ocurre en la relación entre los ingenieros de datos y los analistas de datos. Los ingenieros de datos, con un perfil más técnico, están más preparados para implementar los algoritmos de procesado, así como los métodos de obtención y persistencia de los datos. Por otra parte, los analistas necesitan usar las piezas de código desarrolladas por los ingenieros para llevar a cabo el tratamiento y análisis. En algunos casos, la forma de compartir, usar y reutilizar estas piezas de código no está clara y puede llevar a una pérdida de recursos.

Si a la necesidad de facilitar el uso de las piezas de código implementadas por los ingenieros de datos, añadimos que la metodología CRISP-DM y la propia naturaleza de estos proyectos hace que haya muchos avances y retrocesos, resulta interesante disponer de herramientas que permitan reutilizar trabajo, que aporten agilidad y que reduzcan complejidad a la que tienen que enfrentarse los analistas de datos.

Durante este trabajo de fin de grado se desarrollará una herramienta que apoya al tratamiento de datos dentro de las primeras fases del análisis de los mismos. Para ello genera pipelines (secuencias) de procesado que transforman datos crudos en datos limpios y analizables en formato tabular.

1.2 - Objetivos

Los objetivos del proyecto son expuestos mediante un objetivo general, que expresa el resultado final que se busca conseguir. Este objetivo general se complementa con varios objetivos específicos, más concretos.

- Objetivo general: Desarrollar una herramienta que facilite las primeras fases de los proyectos de análisis de grandes cantidades de datos.
- Los objetivos específicos de la herramienta son:
    a. Definir un método de uso y reutilización de algoritmos de tratamiento de datos.
    b. Ayudar a hacer más conciso el código de un tratamiento de datos.
    c. Permitir a los analistas de datos utilizar algoritmos de procesado de datos sin conocer su implementación.
    d. Ayudar a los analistas de datos a dedicar sus esfuerzos a pensar la mejor manera de aprovechar los datos, evitando perder tiempo pensando cómo tratarlos.

1.3 - Estructura de la memoria

Esta es una breve descripción de los contenidos de cada capítulo:

Capítulo 1: Introducción

Se introduce el trabajo, dando el contexto del mismo y las motivaciones que llevan a su desarrollo.

Capítulo 2: Conceptos previos

Se presenta brevemente el concepto de pipeline y las tecnologías Python 3, Spark, PySpark y Tealeaf.

Capítulo 3: Desarrollo del proyecto

Se presenta el alcance esperado del proyecto y sus objetivos, así como la metodología a utilizar para conseguirlos. A continuación se realiza un análisis de los riesgos que se esperan encontrar a lo largo del proyecto, y se propone un plan de actuación para minimizar el impacto de los mismos.

Capítulo 4: Desarrollo de la biblioteca

Se exponen los requisitos elicitados para la biblioteca, así como una concreción y extensión de estos requisitos en forma de historias de usuario. Posteriormente se describe el proceso de diseño de la biblioteca y se justifican las decisiones tomadas durante el mismo, presentando diagramas de clases y de interacción del sistema. Finalmente, se mencionan y justifican las modificaciones que se deben realizar al diseño para adaptarlo a las herramientas usadas en su implementación.

Capítulo 5: Aplicación de la biblioteca

Se expone el desarrollo completo de una aplicación de la biblioteca utilizada. Se presentan unos datos en bruto y las características que se quiere obtener de ellos. A continuación se describe una

forma de utilizar la biblioteca para llevar a cabo la extracción de estas características. Finalmente, se incluye el código correspondiente a la implementación de esa extracción de características.

Capítulo 6: Pruebas

Se describe el proceso seguido y las pruebas utilizadas para validar el sistema implementado y sus distintos componentes.

Capítulo 7: Conclusiones y trabajo futuro

Se exponen brevemente unas posibles líneas de trabajo futuro para la mejora de la biblioteca implementada, así como unas conclusiones finales para la memoria.

Bibliografía

Anexos

Se incluyen tres anexos con esta memoria:

- Anexo A: Documentación de la biblioteca. Incluye la documentación de las interfaces de todos los componentes de la biblioteca implementada. Esta documentación ha sido generada automáticamente usando la herramienta Sphynx a partir de la que está presente en el propio código.
- Anexo B: Documentación de los componentes desarrollados para la aplicación de la biblioteca. Una documentación similar a la del anexo B, conteniendo la documentación de componentes desarrollados como extensión de la biblioteca para facilitar el desarrollo de la aplicación presentada en el capítulo 5.
- Anexo C: Índice de los contenidos del medio de almacenamiento adjunto con esta memoria.

Capítulo 2 – Conceptos previos

2.1 – Introducción a los Pipelines

En una parte importante de los casos de análisis datos, para obtener valor de los mismos es necesario llevar a cabo una serie de operaciones sobre ellos. Si cada una de estas operaciones se aísla en un elemento de procesado diferente, podemos recurrir al concepto de pipeline. Un pipeline es un conjunto de elementos de procesado, conectados en serie, de modo que la salida de uno es utilizado como entrada del siguiente (como puede verse en la Figura 2).

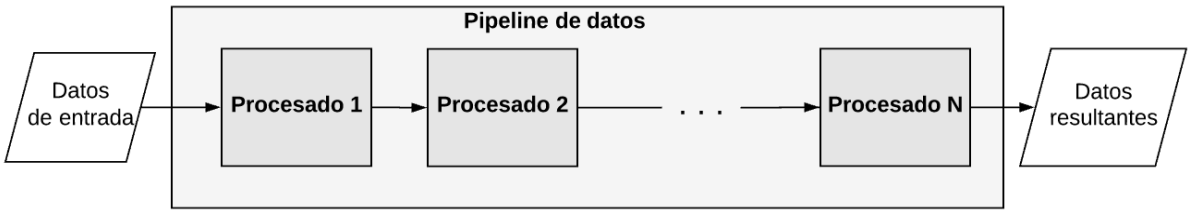


Figura 2: Pipeline de datos.

El uso de pipelines facilita la automatización y la reutilización de las herramientas de procesado, permitiéndonos usar los mismos elementos en distintas aplicaciones.

2.2 – Introducción a Python 3

Python es un lenguaje de programación interpretado de alto nivel. Creado por Guido van Rossum en 1991, tiene una filosofía de diseño que da gran importancia la legibilidad del código.

Python es un lenguaje de tipado débil y dinámico, con gestión de memoria automática. Soporta varios paradigmas de programación, incluyendo orientación a objetos y programación funcional. También dispone de una amplia y bien documentada biblioteca estándar.

Debido a su buena documentación, facilidad de lectura y características de alto nivel, Python también resulta fácil de aprender a profesionales cuya carrera no se basa en el desarrollo de software. Esto resulta especialmente útil en entornos multidisciplinares, como es el del análisis de datos.

Phytonic es un neologismo común entre la comunidad de desarrolladores Python, y se dice de un código que utiliza correctamente las herramientas que proporciona Python y sigue su filosofía minimalista y énfasis en la legibilidad. Estos principios de diseño están expuestos en el documento PEP-20, disponible en la página web del lenguaje [4].

Python es el lenguaje que será utilizado a lo largo de este proyecto debido a los requisitos del mismo, como se expondrá en el capítulo 4. El diseño y desarrollo se realizarán con el objetivo de que, en la medida de lo posible, el código resultante pueda considerarse *pythonic*.

2.3 – Introducción a Spark y Pyspark

Apache Spark es un framework open source de computación paralela en clusters (conjuntos de máquinas conectadas en red). Spark provee de una interfaz de programación que permite implementar tratamientos de datos para su ejecución en dichos clusters. Spark está desarrollado para el procesado paralelo de grandes cantidades de datos en lotes o en flujo contínuo, ya sean estructurados, semi-estructurados o no estructurados.

Todo el tratamiento realizado por Spark es implícitamente paralelo y tolerante a fallos, de modo que no es necesario especificar cómo se lleva a cabo la paralelización del procesado o la recuperación ante la caída de una máquina del cluster. A pesar de esto, la forma en la que cada

tratamiento de datos sea implementado afectará en gran medida a la efectividad de estas dos funcionalidades.

Otra de las características más importantes de Spark es su modelo de optimización DAG (Directed Acyclic Graph), que organiza todas las operaciones a realizar sobre los datos en un grafo dirigido acíclico a medida que estas son llamadas por el código cliente, en lugar de ejecutarlas. La Figura 3 muestra un ejemplo de DAG conteniendo estas operaciones:

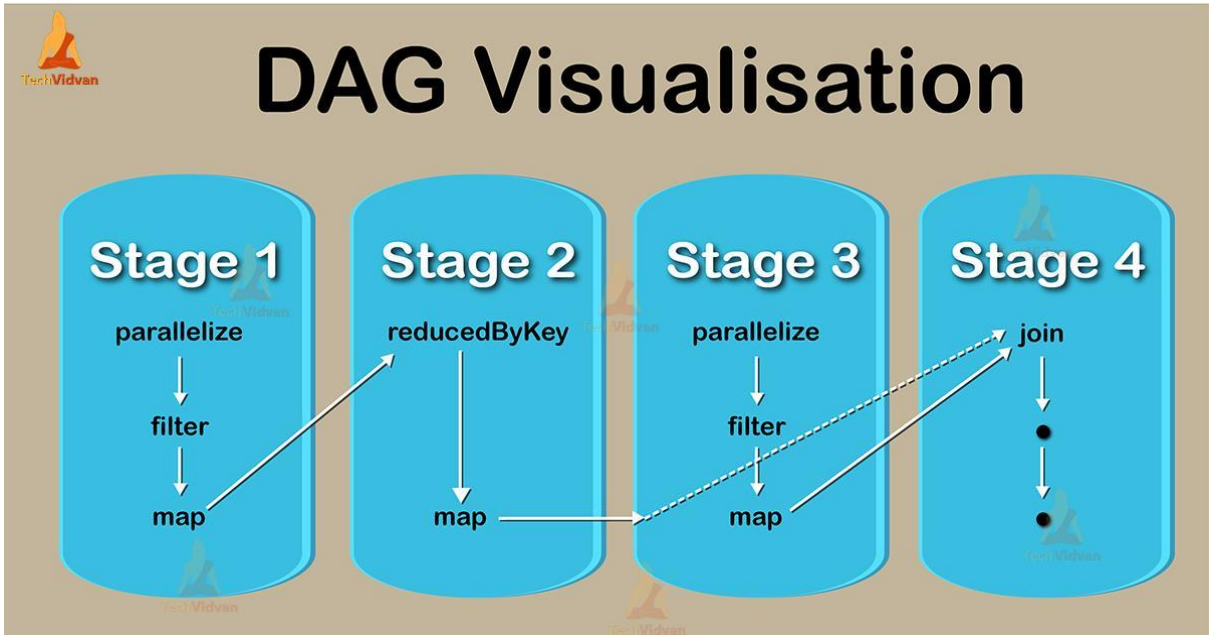


Figura 3: Visualización del DAG de Spark.

El DAG de Spark conecta distintas operaciones, especificando la salida de qué operaciones son necesarias como entrada de otra. De este modo salidas que se usan más de una vez pueden reutilizarse en lugar de volverse a calcular, y en caso de que unos datos se pierdan por la caída de un cluster, pueden volver a calcularse.

El DAG se sigue construyendo a medida que el código cliente solicita la realización de operaciones en el cluster, pero estas no se ejecutan hasta que el resultado no es estrictamente necesario. Esto ocurre, por ejemplo, cuando el código cliente solicita que los resultados se escriban o que se calcule y muestre una estadística de los mismos. Por esto, decimos que Spark tiene evaluación perezosa o *lazy*. Gracias a esta evaluación perezosa procesados complejos pueden ser optimizados por Spark, que solo realiza las operaciones necesarias para obtener la salida deseada.

Spark cuenta con interfaces de programación en varios lenguajes, entre ellos Java, Scala y Python. En la aplicación de Spark que se realizará a lo largo de este proyecto se utilizará la interfaz interfaz de programación que ofrece Spark en el lenguaje Python, con el nombre de PySpark.

2.4 – Introducción a Tealeaf

Tealeaf es una solución software para la realización de analíticas de aplicaciones web y móviles, creada en 1999 y cuya propietaria, desde 2012, es IBM.

Tealeaf captura los detalles de la interacción de los usuarios con la aplicación web mediante la observación pasiva del tráfico de paquetes entre cliente y servidor. De este modo puede aplicarse a aplicaciones web ya desarrolladas sin necesidad de modificarlas.

En sus inicios, Tealeaf fue concebido como una herramienta para ayudar al diagnóstico de errores en las aplicaciones. En muchos casos, los fallos detectados por los usuarios son difíciles de reproducir, y Tealeaf puede usarse como una caja negra que muestra lo que ocurrió antes del error.

Esto es así ya que permite reproducir paso a paso, incluso visualmente, todas las interacciones que un usuario lleva a cabo en la aplicación, así como la respuesta de la misma. Actualmente, Tealeaf también se utiliza para recoger datos que sirvan para el análisis de negocio, para obtener estadísticas sobre los usuarios de una aplicación, y en general para dar visibilidad a las interacciones de los usuarios con el sistema.

Una de las aplicaciones desarrolladas durante este proyecto utilizará datos recogidos mediante esta herramienta.

Capítulo 3 - Desarrollo del Proyecto

En este capítulo se describen los objetivos y alcance del proyecto, así como la metodología utilizada. Finalmente, se expone un análisis de los riesgos a los que se enfrenta el proyecto.

3.1 - Entregables

Los siguientes entregables deberán desarrollarse a lo largo del proyecto:

- **Análisis de riesgos**: Documento en el cual se detallarán los riesgos que predigo que pueden aparecer a lo largo del desarrollo del proyecto. La descripción de cada riesgo incluirá una estimación de su probabilidad de ocurrencia y del daño que puede causar. Esta información se utilizará para tomar las medidas pertinentes, evitando los riesgos más importantes.
- **Análisis de requisitos**: Documento en el que se expondrán los requisitos que debe cumplir el sistema a diseñar e implementar a lo largo del desarrollo del proyecto.
- **Documento de diseño**: Expondrá las características del sistema a implementar, diseñadas de acuerdo a los requisitos.
- **Documentación externa del sistema**: Describirá y explicará las interfaces externas de cada componente del sistema, de modo que pueda servir de referencia a los usuarios que lo utilicen.
- **Código fuente del sistema**, listo para ser utilizado.

3.2 - Descripción de la metodología utilizada

Este proyecto ha sido desarrollado con una estrecha colaboración con el cliente, la empresa vallisoletana Luce Innovative Technologies. A continuación se describe el método utilizado durante el desarrollo del proyecto.

3.2.1 - Scrum

La metodología Scrum es el modelo de referencia utilizado para el desarrollo de este proyecto.

Los roles principales en Scrum son:

- **Product Owner**: Representante de los *stakeholders* o interesados en el sistema, como el cliente.
- **Scrum Master**: Se encarga de facilitar la aplicación de Scrum y asegurarse de que sus reglas se cumplan, así como facilitar al equipo de desarrollo la consecución de sus objetivos.
- **Equipo de desarrollo**: El equipo de desarrolladores del proyecto.

El flujo de trabajo en Scrum ocurre siguiendo los siguientes eventos:

- **Sprint**: Periodos de trabajo, normalmente de una duración determinada y constante (entre una y cuatro semanas), durante la cual se espera cumplir ciertos requisitos y entregar una versión funcional del sistema.
- **Planificación de sprint**: evento que ocurre al inicio de un sprint, en el que el equipo se reúne para definir qué requisitos se intentarán cumplir durante el sprint actual.
- **Scrum diario**: reunión corta diaria en la que cada miembro del equipo informa del estado de su trabajo.
- **Revisión del sprint**: reunión al final de un sprint en la que se muestra al Product Owner los resultados obtenidos. De este modo se regulan las expectativas del cliente y se obtiene retroalimentación durante el proceso de desarrollo, que puede modificar gradualmente algunos requisitos.
- **Retrospectiva del sprint**: reunión al final de un sprint en la que el equipo comparte sus impresiones sobre el mismo, con objetivo de refinar el proceso para el sprint siguiente.

Estas características hacen que Scrum presente, entre otras, las siguientes ventajas:

- **Reducción del *Time to Market***, ya que el cliente puede empezar a utilizar las características más importantes del sistema antes de que esté completado.
- Ayuda a la creación de **software de alta calidad**. La necesidad de obtener en cada iteración una versión funcional del sistema que pueda modificarse ante cambios en los requisitos fuerza a cumplir buenas prácticas de desarrollo de software.
- **Aumenta la productividad**. Debido a que el equipo es autogestionado, se evita burocracia. Además, dado que es el propio equipo el que tiene que poner las medidas necesarias para el éxito del proyecto y mejorar el proceso de desarrollo, se consigue una mayor implicación por parte de sus miembros.
- **Reduce riesgos** relacionados con una comprensión inadecuada de los requisitos del sistema, ya que el cliente ve la evolución del mismo y define las características que necesita que se añadan al software a medida que comienza a utilizarlo.

Sin embargo, el uso de Scrum no es adecuado para cualquier proyecto, sino que se recomienda cuando:

- El cliente se compromete a implicarse en el proceso de desarrollo, designando un Product Owner.
- El equipo de desarrollo es pequeño, de menos de 10 miembros, que están dispuestos a auto-organizarse y superar por sí mismos los obstáculos que aparezcan en el desarrollo.
- El producto a desarrollar es usable aún cuando no está completo al 100%, y pequeños aumentos en su funcionalidad aportan valor.
- Los requisitos no están claros o es posible que cambien.

Ya que este proyecto cumple estas características, la elección de Scrum ha sido considerada adecuada. A pesar de que existen otras metodologías con características similares, Scrum fue elegida por la experiencia que los implicados en el proyecto tienen en su aplicación.

3.2.2 - Aplicación de Scrum al proyecto

En este caso, los roles de Scrum los llevan a cabo:

- **Product Owner**: El tutor dentro de la empresa cliente, que se encargará de revisar el sistema y especificar sus requisitos.
- **Scrum Master**: Óliver L. Sanz.
- **Equipo de desarrollo**: De nuevo, Óliver L. Sanz.

Este proyecto tiene características claves que nos llevan a no seguir al pie de la letra la metodología Scrum:

- El equipo de desarrollo tiene un solo miembro
- La disponibilidad del Product Owner es muy alta, y puede comunicarse con el equipo diariamente.
- El Product Owner tiene un perfil técnico, por lo que podrá asistir al equipo en cuestiones de diseño de software.
- Un aumento muy pequeño en la funcionalidad del sistema aporta valor y puede ser revisado por el Product Owner antes de la finalización del sprint.

Por esto, los eventos del flujo de trabajo Scrum han sido adaptados del siguiente modo:

- El trabajo será realizado en **Sprints** de duración corta y variable, con un máximo de una semana. Al final de cada Sprint se presentará al Product Owner una nueva versión funcional del sistema, que este evaluará. La retroalimentación del Product Owner será utilizado para modificar o añadir requisitos cuando sea conveniente.
- No se realizará **scrum diario**. Dado que el equipo solo tiene un miembro la coordinación inter-equipo no es necesaria. Sin embargo, habrá reuniones diarias con el Product Owner para informarle del estado del sprint y recibir retroalimentación sobre el trabajo que se está realizando.

Los sprints se sucederán hasta que se cumpla con la funcionalidad requerida o se agote el tiempo disponible para el desarrollo del sistema. En este segundo caso, ya que cada iteración aporta una versión funcional del sistema, se presentará la versión de la última iteración como versión final.

3.3 - Gestión de riesgos

A continuación se realiza un análisis de los riesgos a los que se enfrenta el proyecto. Posteriormente se presenta un plan para mitigar el impacto de los riesgos potencialmente más problemáticos.

Un riesgo es la probabilidad de que ocurra una situación concreta en el futuro, que de ocurrir afectaría negativamente a la capacidad para cumplir los objetivos del proyecto.

Para detallar los riesgos se utilizará la siguiente plantilla:

Identificador	Identificador de la forma R-000
Riesgo	Descripción del riesgo
Probabilidad	Estimación de la probabilidad de que el evento que genera el riesgo ocurra. Puede ser *raro*, *poco probable*, *posible*, *muy posible* y *casi seguro*.
Consecuencia	Descripción de las posibles consecuencias del riesgo.
Nivel de impacto	Abstracción de la gravedad de las consecuencias del riesgo. Puede ser *despreciable*, *menor*, *moderado*, *mayor* o *catastrófico*.
Notas	Información adicional.

3.3.1 - Identificación de los riesgos

Identificador	R-001
Riesgo	Cambios críticos o aumento repentino de los requisitos del sistema
Probabilidad	Poco probable
Consecuencia	Aumento del tiempo requerido para desarrollar al sistema.
Nivel de impacto	Mayor
Notas	En este caso es poco probable, debido a que la metodología a utilizar incluye un contacto constante con el cliente.

Identificador	R-002
Riesgo	Disponibilidad insuficiente del equipo
Probabilidad	Raro
Consecuencia	El equipo no dedica el tiempo necesario al proyecto antes de completarse los plazos del mismo.
Nivel de impacto	Catastrófico
Notas	

Identificador	R-003
Riesgo	Disponibilidad insuficiente del Product Owner
Probabilidad	Poco probable
Consecuencia	La comunicación con el cliente no es la adecuada y no se sigue la metodología elegida.
Nivel de impacto	Mayor
Notas	

Identificador	R-004
Riesgo	La dificultad de satisfacción de los requisitos supera las capacidades del equipo.
Probabilidad	Posible
Consecuencia	No se realiza una versión suficientemente completa del sistema en el tiempo previsto.
Nivel de impacto	Mayor
Notas	

Identificador	R-005
Riesgo	El tiempo de desarrollo es insuficiente
Probabilidad	Posible
Consecuencia	Alguno requisitos no pueden satisfacerse
Nivel de impacto	Moderado
Notas	En este caso el impacto es moderado, ya que la metodología utilizada permitirá tener una versión útil del sistema aunque falte tiempo para su desarrollo.

Identificador	R-006
Riesgo	Los requisitos están mal descritos, son ambiguos o contradictorios.
Probabilidad	Raro
Consecuencia	La herramienta desarrollada no satisface las necesidades del cliente
Nivel de impacto	Mayor
Notas	Debido a la metodología utilizada para describir los requisitos con la colaboración constante del cliente, es raro que este riesgo ocurra.

Identificador	R-007
Riesgo	El diseño del sistema es incorrecto y no puede trasladarse a la implementación.
Probabilidad	Posible
Consecuencia	El diseño debe ser desechado o modificado, aumentando el tiempo necesario para el desarrollo del proyecto. En su defecto, la implementación no dispondrá de un diseño adecuado y su calidad será menor.
Nivel de impacto	Mayor
Notas	

Identificador	R-008
Riesgo	Las herramientas utilizadas no disponen de alguna característica necesaria.
Probabilidad	Posible
Consecuencia	Aumento de la complejidad o imposibilidad para satisfacer algunos requisitos.
Nivel de impacto	Mayor
Notas	

Identificador	R-009
Riesgo	Pérdida de trabajo realizado.
Probabilidad	Muy posible
Consecuencia	Necesidad de llevarlo a cabo de nuevo, aumentando el tiempo necesario para la realización del proyecto
Nivel de impacto	Mayor

Notas	

Identificador	R-010
Riesgo	El sistema implementado no cumple alguno de los requisitos principales.
Probabilidad	Poco probable
Consecuencia	El sistema implementado es inútil para el cliente.
Nivel de impacto	Catastrófico
Notas	Es poco probable debido a la revisión constante del sistema que realizará el Product Owner.

3.3.2 - Plan de actuación

En función de la probabilidad y nivel de impacto de cada riesgo, se define su importancia en la medida de su capacidad para obstaculizar o impedir el desarrollo del proyecto. Para ello se usa la siguiente matriz:

Imp. \ Probabil.	**Raro**	**Poco probable**	**Posible**	**Muy probable**	**Casi seguro**
Despreciable	Baja	Baja	Baja	Media	Media
Menor	Baja	Baja	Media	Media	Media
Moderado	Media	Media	Media	Alta	Alta
Mayor	Media	Media	Alta	Alta	Muy alta
Catastrófico	Media	Alta	Alta	Muy Alta	Muy alta

En función del número de riesgos, la importancia de cada uno y los recursos disponibles para su gestión, se establecerán planes que mitiguen el impacto de los más importantes.

En este caso se llevará a cabo con los riesgos que tengan una importancia Alta o mayor, que son los que indica la matriz:

Imp. \ Probabil.	**Raro**	**Poco probable**	**Posible**	**Muy probable**	**Casi seguro**
Despreciable					
Menor					
Moderado			R-005		
Mayor	R-006	R-001, R-003	R-004 R-007 R-008	R-009	
Catastrófico	R-002	R-010			

El análisis pormenorizado y descripción del plan de acción ante cada riesgo se expone siguiendo esta plantilla:

Identificador	Identificador del riesgo en cuestión
Riesgo	Descripción del riesgo
Importancia	Su importancia según la matriz anterior. Solo se analizarán los riesgos con importancia Alta o mayor.
Minimización del riesgo	Plan para minimizar la probabilidad de que el riesgo ocurra.
Monitorización	Plan para detectar indicios de que el riesgo está ocurriendo.
Minimización del impacto	Plan para minimizar el impacto en caso de que el riesgo ocurra.

A continuación se detallan los riesgos con importancia Alta o Muy Alta:

Identificador	R-004
Riesgo	La dificultad de satisfacción de los requisitos supera las capacidades del equipo.
Importancia	Alta
Minimización del riesgo	Dado que los miembros del equipo no pueden sustituirse y no hay tiempo disponible para formación antes de comenzar el proyecto, no es posible minimizar este riesgo.
Monitorización	
Minimización del impacto	Se minimizará el impacto pidiendo asesoramiento a desarrolladores experimentados de la empresa cliente en caso de que algún reto exceda las capacidades del equipo.

Identificador	R-007
Riesgo	El diseño del sistema es incorrecto y no puede trasladarse a la implementación.
Importancia	Alta
Minimización del riesgo	Se minimizará el riesgo mediante un proceso de diseño iterativo implementando pruebas de concepto del diseño.
Monitorización	El Product Owner, que tiene un perfil técnico, revisará periódicamente el diseño y las pruebas de concepto implementadas.
Minimización del impacto	

Identificador	R-008
Riesgo	Las herramientas utilizadas no disponen de alguna característica necesaria.
Importancia	Alta
Minimización del riesgo	Se minimizará el riesgo utilizando las últimas versiones de las herramientas utilizadas.
Monitorización	
Minimización del impacto	

Identificador	R-009
Riesgo	Pérdida de trabajo realizado.
Importancia	Alta
Minimización del riesgo	Se minimizará el riesgo utilizando un sistema de control de versiones con repositorios locales y en la nube para el código fuente del sistema, y alojando la documentación y el resto de entregables en otro sistema en la nube.
Monitorización	
Minimización del impacto	

Identificador	R-010
Riesgo	El sistema implementado no cumple alguno de los requisitos principales.
Importancia	Alta
Minimización del riesgo	
Monitorización	El Product Owner tendrá en cuenta este riesgo a la hora de revisar las distintas versiones del sistema desarrolladas para evitar que sea irremediable si en algún momento comienza a ocurrir.
Minimización del impacto	

Capítulo 4 - Desarrollo de la biblioteca

4.1 - Requisitos

A lo largo de esta sección se definen las características que tendrá que presentar el sistema, tal y como han sido elicitadas del cliente.

En una primera aproximación se exponen los requisitos, tanto funcionales como no funcionales. Los requisitos funcionales describen a grandes rasgos la funcionalidad que debe tener el sistema, es decir: qué debe ser capaz de hacer. Por otra parte, los requisitos no funcionales marcan restricciones que el sistema debe cumplir, o la manera en la que es aceptable que lleguen a cumplirse los requisitos funcionales.

Cabe destacar que estos requisitos constituyen una visión preliminar y abstracta sobre las características que se esperan del sistema. Debido al carácter iterativo de el trabajo a realizar, es de esperar que a medida que se realice el diseño e implementación de la funcionalidad básica del sistema surjan nuevas necesidades, y las definidas en los requisitos se concreten más.

Posterior a esta exposición de requisitos no se encuentra un listado de casos de uso, sino que en su lugar se presenta una serie de historias de usuario. Mientras que un caso de uso es una descripción pormenorizada de la interacción y transacción entre el sistema y el usuario, la historia de usuario se centra en expresar lo que el usuario desea o necesita conseguir mediante el uso del sistema, concretando los requisitos en una descripción de la intención del usuario. Esto, en este caso, tiene múltiples ventajas respecto al caso de uso:

- **No concreta cómo debe ser la interacción con el usuario**. En el caso de que lo hiciera, tener esa restricción presente desde el principio del proyecto podría ser un gran estorbo para un desarrollo en el que los requisitos evolucionan a lo largo del tiempo, obligando a actualizar constantemente los casos de uso y desechar trabajo realizado.
- **Expresa la intención del usuario**. Dado que el objetivo principal de este sistema es ofrecer una mejor interfaz de programación a los usuarios, es primordial que el diseño esté centrado en ellos. Disponer de los requisitos concretados como historias de usuario ayuda a diseñar con una idea muy clara de lo que realmente necesitan, y por lo tanto facilita un buen diseño.
- **Expresa requisitos de una forma fácilmente comprensible** por el usuario y el cliente. Dado que en este proyecto se va a trabajar en una estrecha relación con ellos, es muy conveniente que la negociación y el acuerdo sobre los requisitos sea lo más sencilla posible. Las historias de usuario permiten al cliente comprender y opinar sobre las prioridades del desarrollo, así como definir él mismo nuevas historias de usuario a medida que la funcionalidad del sistema deba ampliarse.

Por estos motivos, para esta fase del proyecto se ha optado por una primera especificación de requisitos generales, seguida por la definición de las historias de usuario.

4.1.1 - Análisis de requisitos

Requisitos funcionales

La siguiente lista de requisitos funcionales define el comportamiento requerido por el sistema, y sigue la siguiente plantilla:

Identificador	Identificador del requisito
Dependencias	Dependencias con otros requisitos
Descripción	Descripción del requisito
Importancia	Importancia del requisito
Comentarios	Clarificaciones sobre el requisito

Los campos son los siguientes:

- El identificador del requisito es de la forma RF-000, correspondiendo RF a “Requisito Funcional”, y 000 a su identificación numérica de tres dígitos
- Las dependencias indican con qué otros requisitos está estrechamente relacionado este. Cambios en estos requisitos podrían hacer necesarias modificaciones en el requisito que está siendo descrito.
- La descripción especifica en qué consiste el requisito.
- La importancia tiene distintos valores en función de lo crítico que sea este requisito para el sistema. En este caso los valores serán: requerido, alta, media o baja.
- Los comentarios son anotaciones de interés con respecto al requisito.

Lista de requisitos funcionales

Identificador	RF-001
Dependencias	RF-002
Descripción	El sistema deberá permitir al usuario procesar un conjunto de datos tabulares según una lógica predefinida por el sistema o implementada por el propio usuario, obteniendo como resultado otro conjunto de datos tabulares.
Importancia	Requerido
Comentarios	

Identificador	RF-002
Dependencias	RF-001
Descripción	El sistema deberá permitir al usuario implementar sus propios algoritmos de procesado de datos, que podrán ser utilizados del mismo modo que los definidos por el sistema.
Importancia	Requerido
Comentarios	

Identificador	RF-003
Dependencias	RF-001
Descripción	El sistema deberá permitir al usuario concatenar varios algoritmos de procesamiento, utilizando los resultados de los algoritmos anteriores como entrada de los posteriores.
Importancia	Requerido
Comentarios	El resultado de esta concatenación será tratado como un solo algoritmo de procesamiento.

Identificador	RF-004
Dependencias	RF-001, RF-005
Descripción	El sistema deberá permitir al usuario ejecutar varios algoritmos de procesado de datos sobre un mismo conjunto de datos tabulares, de manera que los resultados de los diferentes procesados sean agregados en un solo conjunto de datos tabulares, mediante una lógica predefinida por el sistema o implementada por el usuario
Importancia	Alta
Comentarios	El resultado de esta agregación será tratado como un solo algoritmo de procesamiento.

Identificador	RF-005
Dependencias	RF-004
Descripción	El sistema deberá permitir al usuario implementar sus propios algoritmos para la agregación de varios conjuntos de datos tabulares en uno solo, de modo que puedan ser utilizados indistintamente de los ofrecidos por el sistema.
Importancia	Requerido
Comentarios	

Identificador	RF-006
Dependencias	RF-007
Descripción	El sistema deberá permitir al usuario cargar conjuntos de datos tabulares desde distintas fuentes, según una implementación ofrecida por el sistema o realizada por el usuario.
Importancia	Requerido
Comentarios	

Identificador	RF-007
Dependencias	RF-006
Descripción	El sistema deberá permitir al usuario implementar un método de lectura de conjuntos de datos tabulares, de modo que pueda ser utilizada por el sistema indistintamente de uno de los métodos predefinidos.
Importancia	Requerido
Comentarios	

Identificador	RF-008
Dependencias	RF-009
Descripción	El sistema deberá permitir al usuario persistir conjuntos de datos tabulares en diferentes medios, según una implementación ofrecida por el sistema o realizada por el usuario.
Importancia	Requerido
Comentarios	

Identificador	RF-009
Dependencias	RF-008
Descripción	El sistema deberá permitir al usuario implementar un método de persistencia de conjuntos de datos tabulares, de modo que pueda ser utilizada por el sistema indistintamente de uno de los métodos predefinidos.
Importancia	Requerido
Comentarios	

Identificador	RF-010
Dependencias	RF-001, RF-006, RF-008
Descripción	El sistema deberá permitir crear y ejecutar una cola de procesamiento especificando un método de lectura de datos, un algoritmo para su procesado y un método para la persistencia de los resultados.
Importancia	Requerido
Comentarios	

Identificador	RF-011
Dependencias	RF-001
Descripción	El sistema deberá implementar los algoritmos de procesamiento y agregación de datos cuya necesidad se detecte durante el desarrollo del mismo y estén definidos como historias de usuario.
Importancia	Alta
Comentarios	

Identificador	RF-012
Dependencias	RF-006, RF-008
Descripción	El sistema deberá implementar los algoritmos de lectura y persistencia datos cuya necesidad se detecte durante el desarrollo del mismo y estén definidos como historias de usuario.
Importancia	Alta
Comentarios	

Requisitos no funcionales

La siguiente lista de requisitos funcionales define el comportamiento requerido por el sistema, y sigue la siguiente plantilla:

Identificador	Identificador del requisito
Dependencias	Dependencias con otros requisitos
Descripción	Descripción del requisito
Importancia	Importancia del requisito
Comentarios	Clarificaciones sobre el requisito

Tiene los mismos campos que un requisito funcional:

- El identificador del requisito es de la forma RF-000, correspondiendo RF a “Requisito No Funcional”, y 000 a su identificación numérica de tres dígitos
- Las dependencias indican con qué otros requisitos está estrechamente relacionado este. Cambios en estos requisitos podrían hacer necesarias modificaciones en el requisito que está siendo descrito.
- La descripción especifica en qué consiste el requisito.
- La importancia tiene distintos valores en función de lo crítico que sea este requisito para el sistema. En este caso los valores serán: requerido, alta, media o baja.
- Los comentarios son anotaciones de interés con respecto al requisito.

Lista de requisitos no funcionales

Identificador	RNF-001
Dependencias	Ninguna
Descripción	El sistema deberá ofrecer una interfaz de programación en Python3.
Importancia	Requerido
Comentarios	

Identificador	RNF-002
Dependencias	RNF-001
Descripción	Mientras estén cargados en el sistema, los conjuntos de datos deberán ser objetos Python del tipo DataFrame, del framework PySpark.
Importancia	Requerido
Comentarios	

Identificador	RNF-003
Dependencias	RNF-002
Descripción	Los algoritmos de procesamiento implementados por el sistema deberán utilizar para la transformación de los conjuntos de datos solamente las operaciones provistas por el framework PySpark, en su versión 2.2.
Importancia	Requerido
Comentarios	Esto asegura que el procesamiento se puede acelerar mediante su paralelización en un cluster Spark.

Identificador	RNF-004
Dependencias	RNF-001
Descripción	Toda la información que provea en tiempo de ejecución el sistema sobre el estado del procesado y posibles errores será emitida mediante el paquete logging de Python [5], de modo que el destino de esta información pueda ser configurado por el usuario.
Importancia	Alta
Comentarios	De este modo lo que sería salida de la consola python puede dirigirse a cualquier otro destino, como un fichero de logs.

Identificador	RNF-005
Dependencias	RNF-001
Descripción	La interfaz de programación en Python 3 del sistema deberá seguir las convenciones de estilo propias de este lenguaje.
Importancia	Media
Comentarios	

Identificador	RNF-006
Dependencias	Ninguna
Descripción	El sistema deberá incluir una documentación externa que explique el propósito y uso de todas las piezas de las que esté compuesto.
Importancia	Requerido
Comentarios	

Identificador	RNF-007
Dependencias	RF-010, RNF-004
Descripción	Durante la ejecución de una cola de procesados, según se describe en el requisito RF-010, un fallo durante el procesado de un conjunto de datos provocará la emisión de un mensaje que de constancia del fallo, y el procesado continuará con el siguiente conjunto de datos.
Importancia	Alta
Comentarios	De este modo un procesado largo no se interrumpirá por un fichero de datos defectuoso.

4.1.2 - Historias de usuario

A continuación se presentan las historias de usuario, que muestran necesidades de los mismos que el sistema debería satisfacer. Su descripción sigue la siguiente plantilla:

Identificador	Identificador de la historia de usuario.
Tema	Tema sobre el que trata la historia de usuario.
Yo, como...	usuario con un rol determinado...
quiero...	llevar a cabo cierta acción...
para...	satisfacer una cierta necesidad.
Comentarios	Clarificaciones sobre la historia de usuario.
Prioridad	Prioridad de la historia.

Los campos son los siguientes:

- Un identificador para la historia de usuario, del formato: HU-000. HU viene de “Historia de Usuario”, y 000 es el identificador numérico propio de la historia.
- Tema: Una categoría que ayuda a organizar las historias de usuario.
- **Yo, como…** Este campo define el rol que tiene el usuario o interesado en el sistema al que pertenece esta historia, y es importante ya que ayuda a ponerse desde su punto de vista a la hora de considerar la historia de usuario.
- **quiero…** Este campo expresa la intención del usuario, la acción que quiere llevar a cabo por medio del sistema. Este campo es muy similar a un requisito funcional.
- **para…** Este campo expresa la necesidad que el usuario quiere satisfacer mediante el uso del sistema.
- Comentarios y otra información de utilidad sobre la historia.
- La prioridad de la historia, que indica la prontitud con la que cada historia debe ser satisfecha por la implementación. Es importante para que el usuario pueda ver cuanto antes la funcionalidad más básica y especificar nuevas historias de usuario, proporcionando nuevas prioridades en función de sus necesidades y facilitar el desarrollo iterativo. En este caso, las prioridades más altas se han asignado a las historias básicas de las que dependen las demás. Para las demás historias, el cliente ha especificado la prioridad en función de sus necesidades.

Lista de historias de usuario

Identificador	HU-001
Tema	Extensibilidad
Yo, como...	ingeniero de datos
quiero...	implementar diferentes algoritmos de procesamiento de datos
para...	para que puedan ser utilizados por usuarios que no conozcan los pormenores de su implementación
Comentarios	Esto permitirá separar la implementación y el uso de los diferentes algoritmos de procesamiento.
Prioridad	Máxima

Identificador	HU-002
Tema	Extensibilidad
Yo, como...	ingeniero de datos
quiero...	implementar diferentes algoritmos para la agregación de resultados de múltiples procesados
para...	que pueda ser utilizada por un usuario que no conozca los pormenores de su implementación
Comentarios	Esto permitirá separar la implementación y el uso de los diferentes algoritmos de agregación de resultados.
Prioridad	Alta

Identificador	HU-003
Tema	Lectura de datos, extensibilidad
Yo, como...	ingeniero de datos
quiero...	cargar datos desde distintas fuentes y siguiendo una lógica que yo mismo pueda definir
para...	que puedan ser procesados usando los algoritmos implementados y explotados por los analistas de datos.
Comentarios	
Prioridad	Media

Identificador	HU-004
Tema	Persistencia de datos, extensibilidad
Yo, como...	ingeniero de datos
quiero...	persistir datos en distintos medios y siguiendo una lógica que yo mismo pueda definir
para...	que los resultados de los procesados puedan ser explotados en el futuro
Comentarios	
Prioridad	Media

Identificador	HU-005
Tema	Orquestación
Yo, como...	ingeniero de datos
quiero...	necesito crear una cola de procesado en la que se obtengan conjuntos de datos, se procesen y se persistan los resultados según unas lógicas determinadas
para...	poder realizar procesados sobre muchos conjuntos de datos de forma automatizada
Comentarios	
Prioridad	Media

Identificador	HU-006
Tema	Procesado fácil
Yo, como...	analista de datos
quiero...	configurar y ejecutar algoritmos de procesado de datos sin conocer los pormenores de su implementación
para...	obtener información útil de los datos.
Comentarios	Para realizar esta historia el analista no tiene por qué conocer la implementación del algoritmo, sólo su interfaz de uso.
Prioridad	Máxima

Identificador	HU-007
Tema	Procesado fácil
Yo, como...	analista de datos
quiero...	comprender y modificar un procesado previamente configurado
para...	poder iterar sobre trabajo previo
Comentarios	Para satisfacer esta historia de usuario la configuración de un procesado deberá ser legible.
Prioridad	Alta

Identificador	HU-008
Tema	Procesado fácil
Yo, como...	analista de datos
quiero...	aplicar varios procesados sobre un mismo conjunto de datos y agregar los resultados
para...	para extraer de los datos información desde varias perspectivas, teniendo los resultados recogidos en un solo conjunto de datos
Comentarios	Esto permite crear algoritmos de procesado complejos a partir de otros maś sencillos.
Prioridad	Alta

Identificador	HU-009
Tema	Procesado fácil
Yo, como...	analista de datos
quiero...	realizar secuencias de procesados utilizando los resultados de procesados anteriores como entradas de los posteriores.
para...	obtener en cada paso una versión más resumida e informativa de los datos.
Comentarios	Esto permite crear algoritmos de procesado complejos a partir de otros maś sencillos.
Prioridad	Alta

Identificador	HU-010
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	obtener, a partir de un conjunto de datos, una columna que contenga la suma de los valores de dos o más columnas numéricas
para...	obtener una versión más resumida e informativa de los datos.
Comentarios	
Prioridad	Media

Identificador	HU-011
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	‘binarizar’ una columna de un conjunto de datos, es decir: sustituir por ceros los valores nulos, los ceros o las cadenas de texto vacías, y sustituir por unos cualquier otro valor
para...	obtener una versión más resumida e informativa de los datos.
Comentarios	
Prioridad	Media

Identificador	HU-012
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	a partir de un conjunto de datos, obtener otro que solo contiene un subconjunto de las columnas del primero.
para...	obtener una versión más resumida e informativa de los datos.
Comentarios	
Prioridad	Media

Identificador	HU-013
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	a partir de un conjunto de datos, obtener una nueva columna con la suma de todas las columnas numéricas cuyo nombre comparte un determinado prefijo
para...	obtener una versión más resumida e informativa de los datos.
Comentarios	
Prioridad	Muy baja

Identificador	HU-014
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	a partir de un conjunto de datos, obtener una columna con la suma ponderada de dos o más columnas numéricas.
para...	obtener una versión más resumida e informativa de los datos.
Comentarios	
Prioridad	Baja

Identificador	HU-015
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	realizar feature hashing sobre una columna de un conjunto de datos.
para...	obtener una versión más resumida e informativa de los datos.
Comentarios	
Prioridad	Baja

Identificador	HU-016
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	añadir un identificador único a cada columna de un conjunto de datos
para...	preparar a los datos para procesados posteriores.
Comentarios	
Prioridad	Muy baja

Identificador	HU-017
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	obtener un conjunto de datos tabular a partir de otro en formato JSON
para...	utilizar con esos datos las herramientas de explotación de datos tabulares.
Comentarios	
Prioridad	Muy baja

Identificador	HU-018
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	ante una columna que contiene listas, generar nuevas filas por cada valor de la lista, y una nueva columna que contenga el valor de la lista correspondiente a cada fila
para...	obtener una versión de los datos más fácil de tratar usando otras herramientas.
Comentarios	
Prioridad	Muy baja

Identificador	HU-019
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	sustituir por un valor determinado los valores nulos de una columna de un conjunto de datos
para...	obtener una versión de los datos más fácil de tratar usando otras herramientas.
Comentarios	
Prioridad	Media

Identificador	HU-020
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	realizar one hot encoding sobre una columna de un conjunto de datos
para...	obtener una versión de los datos más fácil de tratar usando otras herramientas.
Comentarios	
Prioridad	Baja

Identificador	HU-021
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	a partir de un conjunto de datos tabular con identificadores, obtener para cada identificador el número de filas que cumplen una cierta condición.
para...	obtener una versión más resumida e informativa de los datos.
Comentarios	
Prioridad	Media

Identificador	HU-022
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	pasar a minúsculas y eliminar los espacios al principio y al final de las cadenas de texto encontradas en una columna de un conjunto de datos
para...	obtener una versión más resumida e informativa de los datos.
Comentarios	
Prioridad	Media

Identificador	HU-023
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	dividir las cadenas de texto encontradas en una columna de un conjunto de datos en listas de palabras
para...	obtener una versión de los datos más fácil de tratar usando otras herramientas.
Comentarios	
Prioridad	Baja

Identificador	HU-024
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	a partir de un conjunto de datos tabular con identificadores, obtener para cada identificador una lista con todos los valores encontrados en cierta columna de cada fila asociada a ese identificador.
para...	obtener una versión de los datos más fácil de tratar usando otras herramientas.
Comentarios	
Prioridad	Muy baja

Identificador	HU-025
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	a partir de una columna que contiene cadenas de texto con una estructura de pares clave-valor, obtener una nueva columna que contenga los valores asociados a una clave dada.
para...	obtener una versión de los datos más fácil de tratar usando otras herramientas.
Comentarios	
Prioridad	Muy baja

Identificador	HU-026
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	a partir de una columna, obtener otra que contiene los valores de esa columna para las filas que cumplen una cierta condición.
para...	obtener una versión más resumida e informativa de los datos.
Comentarios	
Prioridad	Media

Identificador	HU-027
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	sustituir los valores de una columna por otros, siguiendo un criterio uno a uno de valor encontrado-valor sustituto.
para...	obtener una versión más resumida e informativa de los datos.
Comentarios	
Prioridad	Media

Identificador	HU-028
Tema	Algoritmo de procesado
Yo, como...	analista de datos
quiero...	a partir de un conjunto de datos tabular con identificadores, obtener para cada identificador la suma de todos los valores de una columna numérica, para las filas correspondientes a ese identificador.
para...	obtener una versión más resumida e informativa de los datos.
Comentarios	
Prioridad	Media

Identificador	HU-029
Tema	Método de persistencia de datos
Yo, como...	ingeniero de datos
quiero...	persistir conjuntos de datos en ficheros csv almacenados según una estructura de subdirectorios asociada a la fecha de recogida de los mismos
para...	poder explotar estos datos en el futuro y almacenarlos en una estructura de directorios determinada e informativa.
Comentarios	
Prioridad	Media

Identificador	HU-030
Tema	Método de lectura de datos
Yo, como...	ingeniero de datos
quiero...	leer conjuntos de datos desde ficheros csv almacenados según una estructura de subdirectorios asociada a la fecha de recogida de los datos
para...	explotar los datos almacenados.
Comentarios	
Prioridad	Media

Identificador	HU-031
Tema	Algoritmo de agregación de resultados
Yo, como...	ingeniero de datos
quiero...	agregar los resultados de varias agregaciones mediante una operación equi join
para...	disponer de los resultados de diversos procesamientos en un solo conjunto de datos.
Comentarios	
Prioridad	Media

4.2 - Diseño

Debido a la naturaleza de los requisitos y las necesidades de los usuarios, se toma la decisión de satisfacerlos mediante una biblioteca Python que encapsule y modularice las diferentes funciones requeridas para el tratamiento de datos. Esta biblioteca deberá servir como una herramienta sencilla de utilizar para los analistas, y al mismo tiempo un esqueleto que los ingenieros de datos podrán aprovechar para hacer más modular y reutilizable su código.

En esta sección se detallan los resultados del proceso de diseño de la biblioteca. Las subsecciones de las que consiste son las siguientes:

1. Arquitectura: Se presenta el proceso de diseño de la arquitectura del sistema, apoyado en las historias de usuario. Se exponen de forma incremental y abstracta las piezas por las que está formado el sistema, así como sus relaciones.
2. Diseño detallado: Explica las características de cada clase de la biblioteca, módulo a módulo, con sus correspondientes diagramas de clases UML.
3. Diseño de la interacción: Se expone el diseño de la interacción entre el usuario y el sistema, así como entre los módulos y clases de la propia biblioteca, mediante los correspondientes diagramas de interacción UML.

4.2.1 - Arquitectura

4.2.1.1 - Transformer

Las necesidades más básicas que debe satisfacer el sistema son:

- Permitir encapsular la implementación de un procesado de datos (HU-001)
- Permitir el uso de ese procesado sin que sea necesario conocer su implementación (HU-006)

Para satisfacer esta necesidad se introduce el concepto de **Transformer** (o transformador). Un Transformer es una pieza del sistema que toma como entrada un conjunto de datos, realiza procesados sobre él según su implementación, y devuelve el conjunto de datos resultante de ese procesado (como muestra la Figura 4).

Figura 4: Esquema del procesado realizado por un Transformer.

Utilizando este concepto, es posible encapsular un algoritmo de tratamiento de datos en un Transformer, para que otros usuarios puedan usarlo en sus procesados cuando sea necesario, atendiendo solo a su documentación externa e interfaz, y no a su implementación.

4.2.1.2 - Pipeline

Según la historia de usuario HU-009, en ocasiones es necesario encadenar múltiples operaciones sobre los datos, de modo que la salida de un algoritmo de procesado sea utilizado como entrada del siguiente. Dado que los algoritmos estarán a partir de ahora encapsulados en Transformers, esta necesidad muta en la de utilizar la salida de un Transformer como entrada del siguiente.

Para facilitar esto, se introduce el concepto de **Pipeline**. Un Pipeline es una pieza del sistema que, de acuerdo a una configuración previa, se encarga de dirigir la salida de cada Transformer a la entrada del siguiente, como muestra la Figura 5.

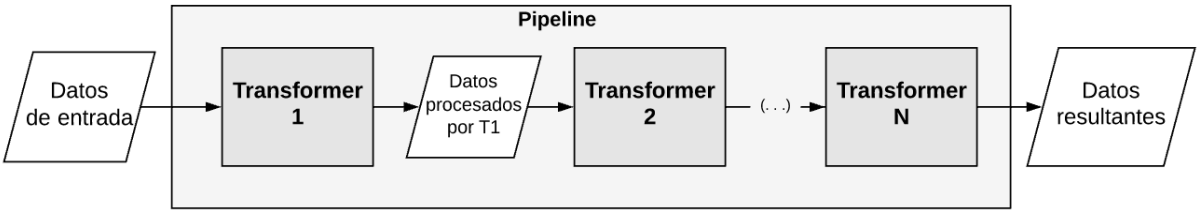


Figura 5: Esquema del procesado realizado por un Pipeline

4.2.1.3 - Step

Similar a la del procesado secuencial que satisfacen los Pipelines, también existe la necesidad de aplicar diferentes procesados sobre el mismo conjunto de datos, agregando los resultados de cada uno de ellos (HU-008). Utilizando de nuevo la abstracción que nos facilitan los Transformers, se puede reescribir lo anterior como: la necesidad de aplicar varios Transformers a un mismo conjunto de datos, y agregar los resultados de cada uno de ellos en un solo conjunto de datos resultado.

Para satisfacer esta necesidad se introduce el concepto de **Step** (o paso). Un Step es una pieza del sistema cuya función es aplicar a un conjunto de datos varios Transformers y agregar sus resultados. Dado que puede haber múltiples formas de agregar estos resultados, el Step utilizará una nueva pieza, el DataMerger (o agregador de datos). El **DataMerger** es, por lo tanto, el componente del sistema encargado de unir varios conjuntos de datos en uno solo, según su implementación.

Utilizando estos dos nuevos conceptos, el procesado realizado por un Step sería como muestra la figura 6.

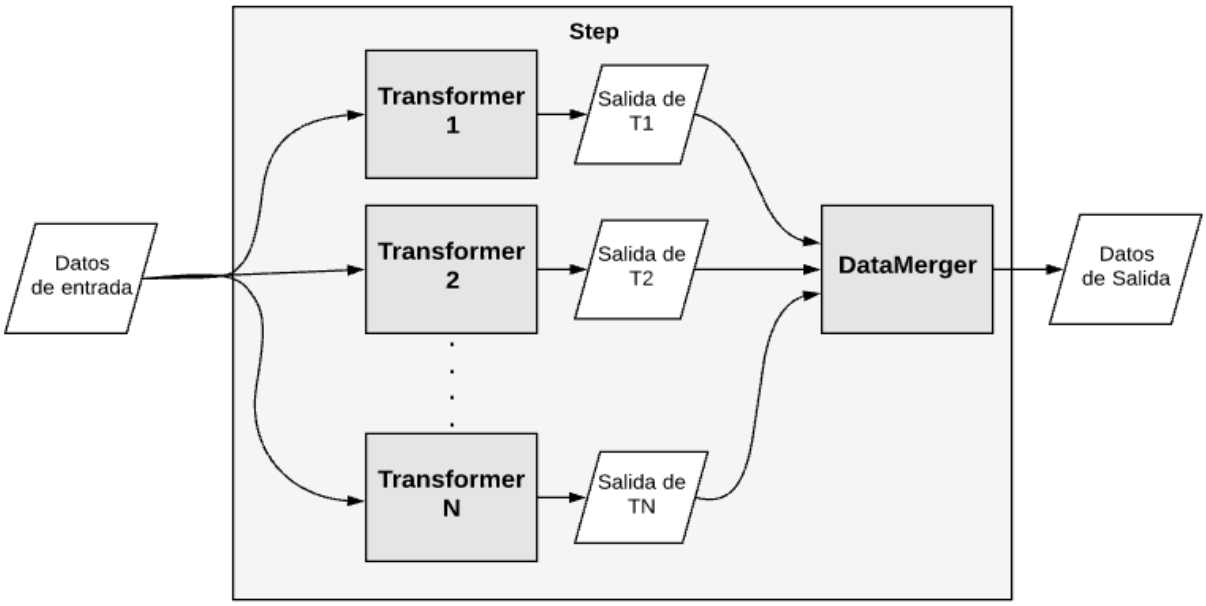


Figura 6: Esquema del procesado realizado por un Step

Además, el DataMerger satisface la historia de usuario HU-002, que precisamente describe la necesidad de una herramienta para agregar resultados de múltiples procesos.

4.2.1.4 - Transformer, Pipeline y Step como clases Python

Los tres componentes introducidos en este apartado serán implementados como clases Python. Por un lado, el Transformer será una interfaz, es decir, una clase que define métodos pero no los implementa. El único método definido por esta interfaz será el método *transform*, que recibirá un conjunto de datos como entrada y dará otro como salida. De este modo, cada vez que un desarrollador implementa un nuevo algoritmo de tratamiento de datos, este puede ser encapsulado en el método *transform* de una nueva clase que implemente la interfaz Transformer.

Por otro lado, tanto el Pipeline como el Step serán implementados como clases no abstractas. Estas dos clases incluirán un método *transform* que aplicará las operaciones correspondientes a un conjunto de datos y devolverá otro conjunto de datos resultante. Por esto, Pipeline y Step implementan la interfaz Transformer, como muestra el diagrama UML de la Figura 7.

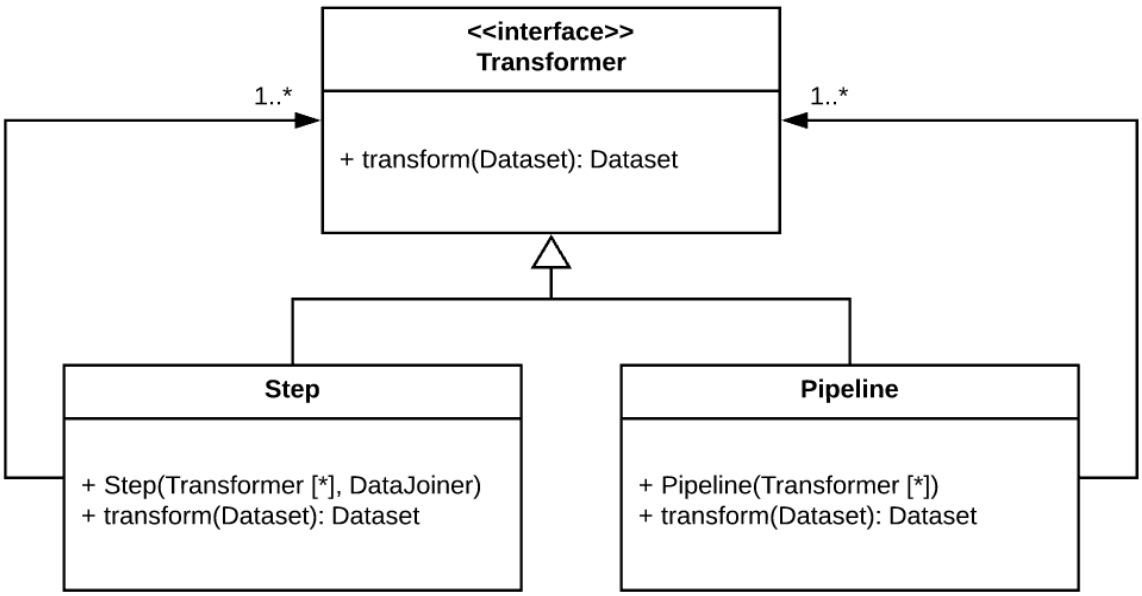


Figura 7: Diagrama de clases de Transformer, Step y Pipeline.

Esta relación entre Step, Pipeline y Transformer es muy potente ya que, según el principio de sustitución de Liskov (cada clase que hereda de otra puede usarse como su padre sin necesidad de conocer las diferencias entre ellas) [6], permite usar Steps y Pipelines allí donde se pueda utilizar un Transformer. Por lo tanto, un Pipeline podrá estar a su vez compuesto de otros Pipelines o Steps, y del mismo modo un Step podrá contener Pipelines.

Aunque hay muchas formas posibles de combinar Pipelines y Steps, lo más común será encontrar Pipelines compuestos de varios Steps, como muestra el ejemplo de la Figura 8 (en la que las salidas de los Transformers han sido omitidas), con un Pipeline formado por dos Steps.

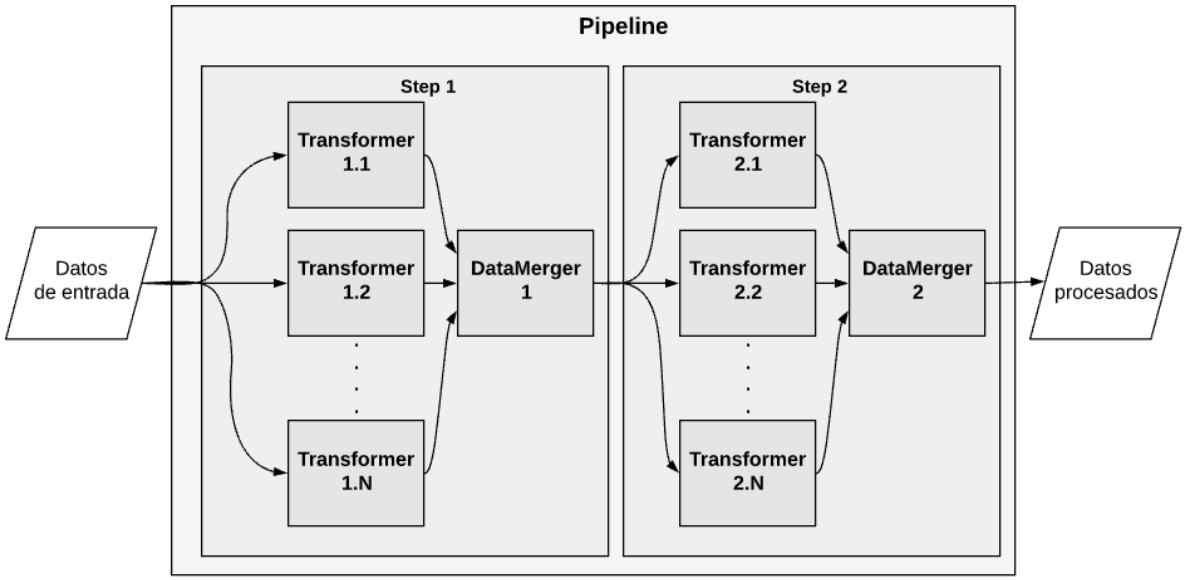


Figura 8: Esquema del proceso realizado por un Pipeline formado por Steps.

Utilizando cuatro conceptos sencillos (Transformer, Pipeline, Step y DataMerger) se obtiene una abstracción que permite encapsular algoritmos de procesado sencillos en Transformers, para posteriormente componer con ellos algoritmos más complejos utilizando Pipelines y Steps.

Esta estructura es la que seguirá, en líneas generales, el procesado a realizar por la herramienta aquí descrita. Para utilizarla, un usuario solo deberá disponer de la documentación de cada Transformer que desee utilizar, y conocer la función de los Pipelines y Steps. A partir de ese punto, podrá combinarlos como desee para construir algoritmos de procesado de datos arbitrariamente complejos, sin tener ninguna necesidad de conocer la implementación de los algoritmos base.

Cabe mencionar que para el diseño de esta parte del sistema se han seguido los siguientes patrones de diseño:

- **Interface** (Interfaz). Consiste en utilizar una clase abstracta para definir la interfaz que todos sus descendientes presentarán al exterior, ayudando a separar función de implementación [7]. En este caso, la clase Transformer es la que define la interfaz del Pipeline, el Step, y de todos los algoritmos de procesado de datos que se implementarán como descendientes del Transformer.
- **Strategy** (Estrategia). Consiste en definir una familia de algoritmos con la misma interfaz, encapsularlos y hacerlos intercambiables, pudiendo variar el comportamiento de los objetos de los que forman parte [8]. En este caso, los Transformers constituyen la encapsulación de los algoritmos de procesado de un conjunto de datos, que son intercambiables dentro de Pipelines y Steps.
- **Mediator** (Mediador). Consiste en definir un objeto mediador que regula la interacción entre otros, evitando que estos últimos estén referenciados entre sí, facilitando el desacoplamiento [9]. En este caso, el Pipeline sirve de mediador entre varios Transformers, y el Step sirve de mediador entre varios Transformers y un DataMerger.
- **Composite** (Contenedor). Consiste en componer objetos usando una estructura en árbol, de modo que un objeto pueda ser una hoja o un nodo del árbol, y en este último caso estará formado por otras hojas y/o nodos. El objetivo de este patrón es permitir a los clientes de un

objeto tratarlo del mismo modo sea compuesto o simple [10]. En este caso, tanto Pipelines como Steps contienen una colección de otros Transformers, que pueden ser tanto Transformers sencillos como otros Pipelines o Steps. Además, independientemente de cómo esté compuesto, todo Transformer, Pipeline o Step hace lo mismo de cara sus clientes: transformar un conjunto de datos en otro.

Ya que estos conceptos forman el núcleo del sistema, todo el diseño y el uso de patrones expuesto hasta ahora intenta cumplir de la mejor manera posible con el principio de Inversión de Dependencia, que dice que las clases más abstractas no deberían depender de las más concretas, sino que todas deberían depender de las abstracciones [11]. Esto se ha conseguido cumplir por el momento, ya que las únicas dependencias son hacia la interfaz Transformer, como muestra la Figura 8.

4.2.1.5 - Entrada, salida y orquestación

La abstracción de los Transformers, junto a los Pipelines y Steps, nos da la capacidad de especificar y llevar a cabo procesados de datos cumpliendo con los requisitos de los usuarios. Sin embargo, ningún dato se puede procesar si no es antes obtenido, ya sea leído del sistema de ficheros de la máquina o desde la nube. Tampoco tiene sentido hacer un procesamiento si los resultados no pueden ser persistidos para su explotación futura o enviados a otro sistema. Dado que existen muchas maneras de obtener y persistir datos, es necesario definir dos nuevas interfaces:

- Un **InputProvider** se encarga de obtener conjuntos de datos, cuyo origen variará en función de su implementación.
- Un **OutputProvider** tiene la responsabilidad de persistir o enviar conjuntos de datos, a un destino que dependerá de su implementación.

De este modo, el flujo normal de los datos será el siguiente:

1. Los datos son obtenidos por un InputProvider.
2. Los datos obtenidos son tratados por un Transformer.
3. El resultado del tratamiento es persistido o enviado por un OutputProvider.

Para mantener desacopladas las tres piezas básicas del proceso (InputProvider, Transformer y OutputProvider) se introduce una nueva pieza en el sistema: el Orquestador. Un **Orquestador** debe conocer a un InputProvider, un Transformer y un OutputProvider, y se encarga de gestionar la interacción entre ellos haciendo lo siguiente:

1. Solicita al InputProvider un conjunto de datos.
2. Transforma este conjunto de datos utilizando el Transformer.
3. Envía el resultado al OutputProvider.

Con estas nuevas piezas se resuelve el problema de la entrada y salida de datos al sistema, utilizando los siguientes patrones de diseño:

- **Mediator** (Mediador, explicado anteriormente). Ya que el orquestador gestiona la interacción entre InputProvider, Transformer y OutputProvider.
- **Dependency Injection** (Inyección de dependencias). Consiste en desacoplar un proceso de sus dependencias externas, como puede ser el conector con una base de datos concreta, mediante la inyección de objetos que se encargan de gestionarlas [12]. En este caso, al Orquestador se le inyecta un InputProvider y un OutputProvider que lo desacoplan de las dependencias con las fuentes de lectura y destino de los datos.

4.2.1.6 - Paquete genérico *Data Pipelines*

Todas las piezas mencionadas hasta el momento constituyen el núcleo más abstracto del sistema, a partir del cual derivan las concreciones que le dan funcionalidad, siguiendo el principio de inversión de dependencia.

En los requisitos no funcionales se especifica que los datos que la biblioteca debe procesar están en la forma de la estructura de datos DataFrame del framework PySpark (RNF-002). También se especifica que las operaciones básicas realizadas sobre estos DataFrames deben ser las provistas

por el propio PySpark, para asegurar que el procesado puede ser paralelizado en un cluster (RNF-003). A pesar de esto, los conceptos introducidos en esta sección no son dependientes de PySpark, ya que no asumen la naturaleza de los datos:

- Los Pipelines y Steps no realizan ninguna operación sobre los datos por sí mismos, sino que solo aplican otros Transformers a los datos.
- El Orchestrator solo se encarga de gestionar el flujo de datos entre InputProvider, Transformer y OutputProvider.
- El resto de las clases introducidas hasta el momento (Transformer, InputProvider, OutputProvider y DataMerger) son interfaces, y por lo tanto no contienen comportamiento.

Es por esto que todos los conceptos descritos hasta ahora forman un módulo genérico: sea cual sea el tipo de los datos (siempre que sea consistente entre todas las piezas del módulo), estas herramientas conceptuales pueden usarse para su tratamiento.

Esto permite que, en caso de que en el futuro se decida utilizar un motor de procesamiento distinto de PySpark, solo las piezas más concretas del sistema deberán cambiarse, lo que permitirá que los usuarios y otros sistemas conserven la misma interfaz de uso de este sistema, independientemente del motor de procesado que haya por debajo.

Estos conceptos se encapsulan en un módulo genérico que ha sido nombrado como **Data Pipelines**, y se describe en el diagrama UML de la Figura 9:

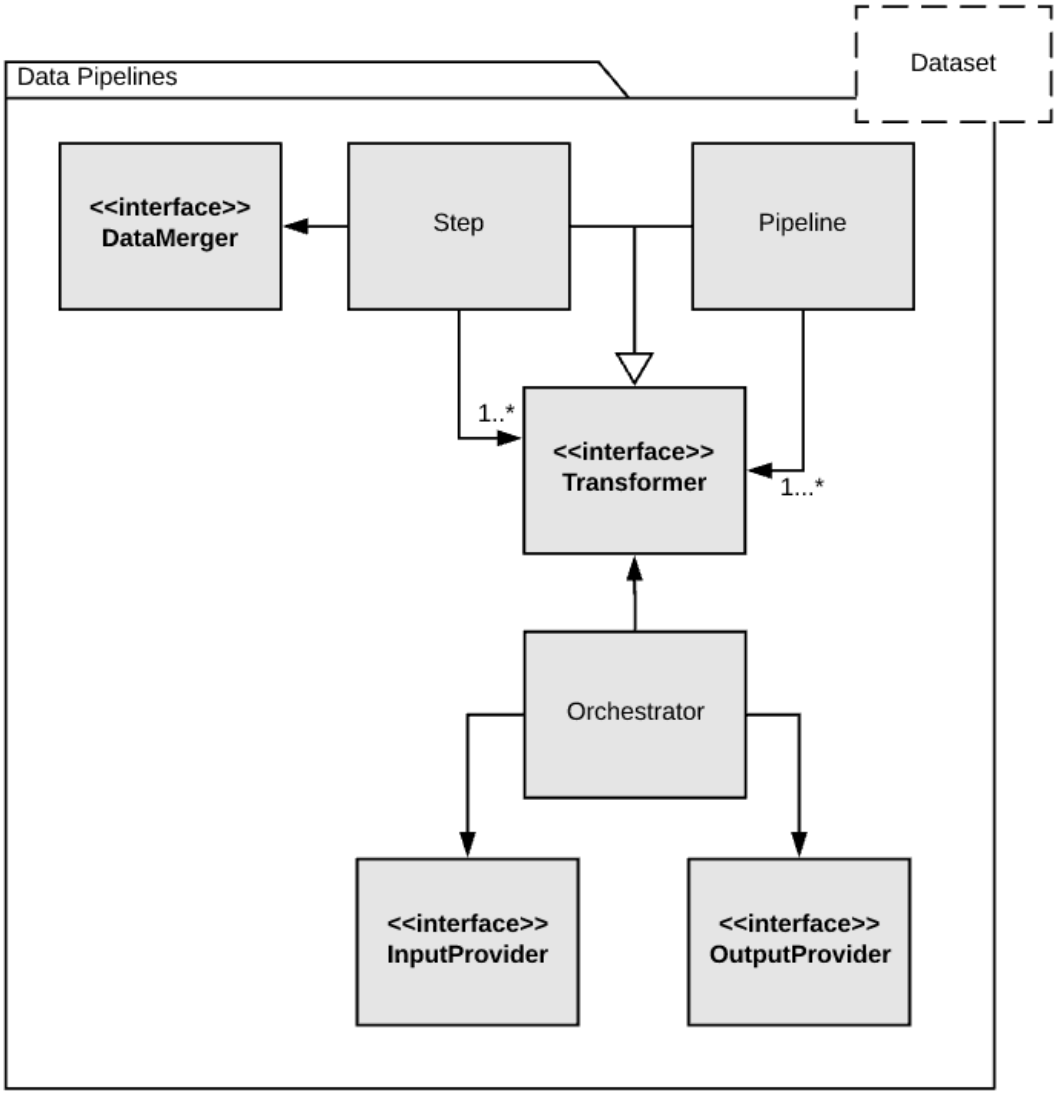


Figura 9: Diagrama de clases del paquete Data Pipelines.

4.2.1.7 - Paquete PySpark Operators

Para que esta herramienta sea funcional, es necesario agregar concreciones que contengan los algoritmos a utilizar.

Las historias de usuario desde la HU-010 hasta la HU-028 expresan la necesidad de utilizar un cierto algoritmo de procesado de datos, mientras que la historia HU-031 expresa la necesidad de uso de un algoritmo concreto para la agregación de resultados. Según expresa el requisito RF-011, estos algoritmos deben estar implementados en la biblioteca: los algoritmos de procesado serán implementados como una implementación de la interfaz Transformer, mientras que el algoritmo de agregación de resultados lo serán como una implementación de DataMerger.

Por otra parte, las historias de usuario HU-029 y HU-030 hablan de la necesidad de métodos concretos de lectura y escritura de datos, que según el requisito RF-012 deben estar implementados por la biblioteca. El método de lectura de datos se realizará como implementación de la interfaz InputProvider, y el de escritura de datos, como implementación de OutputProvider.

Estos nuevos Transformers, DataMergers, InputProviders y OutputProviders serán recogidos en un nuevo paquete, que ha sido nombrado como **PySpark Operators**. Cabe hacer hincapié en que este no es un paquete propio de PySpark, sino que es el paquete en el que se recogen las piezas del sistema que son dependientes de PySpark. El paquete se muestra junto al paquete Data Pipelines en la Figura 10.

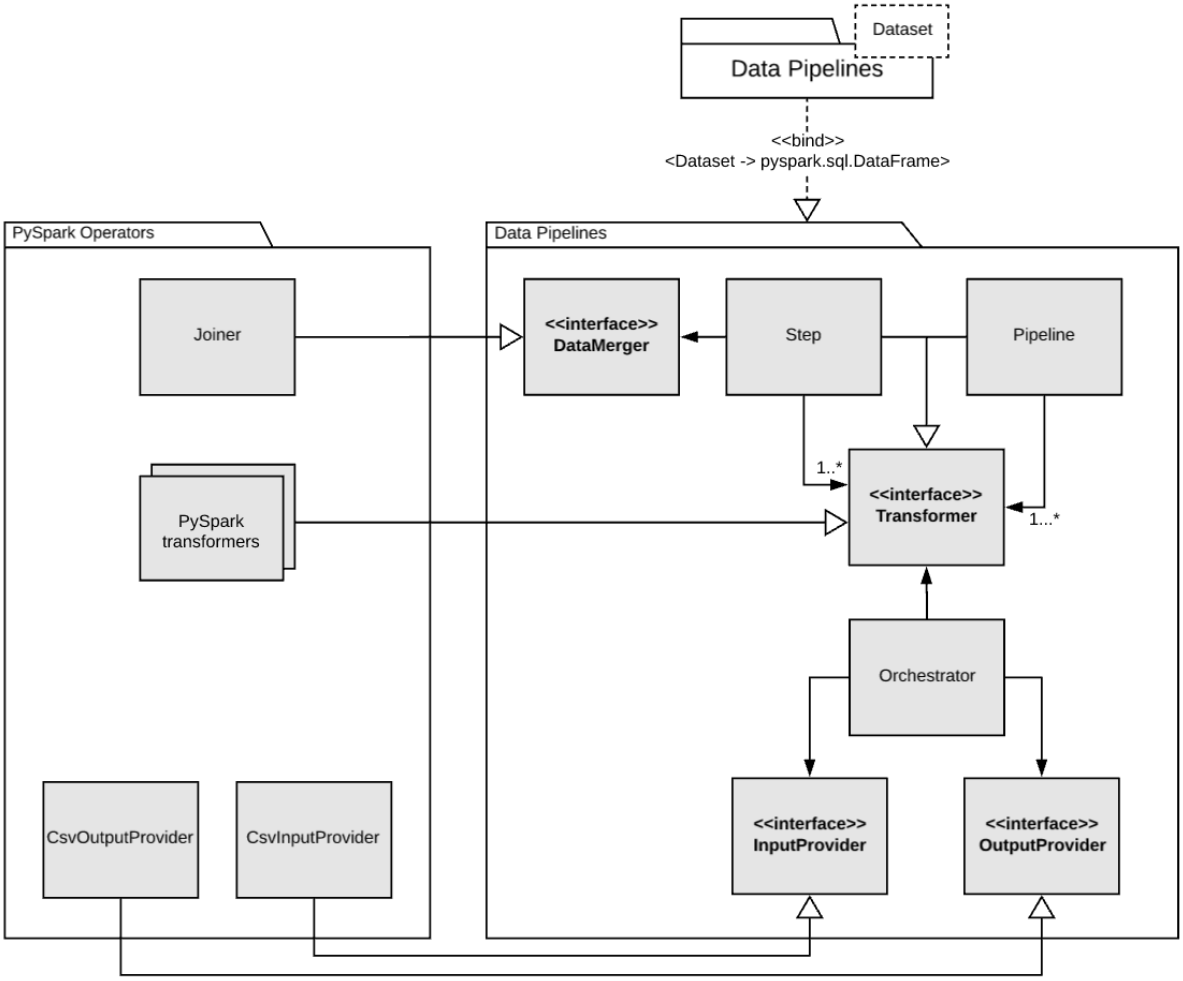


Figura 10: Diagrama de clases de los paquetes Data Pipelines y PySpark Operators.

Hay que notar que es necesario concretar el paquete genérico Data Pipelines, ya que los Transformers y resto de componentes añadidos utilizan y transforman conjuntos de datos del tipo DataFrame de PySpark. Por esto, nuestro paquete genérico se concreta para especificar que los

conjuntos de datos que trata son del tipo DataFrame. Para documentarlo, utilizo la palabra clave UML “bind”, junto con la información de que el tipo Dataset se concreta en el tipo DataFrame.

A la vista de la Figura 10 se puede ver que el sistema cumple perfectamente con el principio de inversión de dependencias, ya que las únicas dependencias presentes en él son hacia interfaces, clases puramente abstractas.

4.2.1.8 - Repasando las historias de usuario

Tras el proceso de diseño de este sistema, se consideró que el resultado obtenido y expuesto hasta ahora en esta sección cumplía con todos los requisitos y satisfacía las historias de usuario. Sin embargo, a lo largo del proceso iterativo de desarrollo se llegó a la conclusión que este diseño no cumple con su cometido.

Según las historias de usuario, la biblioteca debe implementar 19 algoritmos de procesado de datos, cada uno encapsulado en una clase que implemente la interfaz Transformer. Cada algoritmo requiere una inicialización diferente. Por ejemplo, un algoritmo que realiza una suma ponderada de columnas requiere la lista de las columnas a sumar y los pesos correspondientes, mientras que otro que sustituye valores en una columna necesita la especificación de qué valores sustituir por qué otros. A la hora de crear un Pipeline con sus correspondientes Steps, la heterogeneidad en la inicialización de los Transformers hacía el código confuso e ilegible.

Además, mientras que el concepto de un Pipeline formado por varios Steps es sencillo e intuitivo para cualquier analista de datos, el concepto de Transformer no lo es tanto, y requiere una comprensión más profunda de la arquitectura de la biblioteca.

Por estos motivos, se ha considerado que el diseño propuesto no satisface adecuadamente las historias de usuario HU-006 y HU-007, que expresan la necesidad por parte de los analistas de datos de una herramienta que les permita hacer los procesados de forma intuitiva, centrándose en su significado y no en su implementación. El sistema es adecuado en términos de ingeniería de software, ya que cumple principios de diseño como los que forman el conjunto SOLID. En cambio, no lo es en términos de usabilidad: las abstracciones que lo hacen modular y realmente potente también lo han hecho demasiado complejo para ser utilizado sin conocerlo en profundidad.

Para solucionar este problema, se consideró usar el patrón **Builder** (Constructor) [13], forzando a todos los Transformers a encapsular sus parámetros de inicialización para presentar una interfaz común que pudiera usarse por una clase constructora para crear los Transformers mediante una interfaz más amigable. Sin embargo, esta solución agravaba el problema, ya que el uso de una nueva clase para la construcción de los transformadores no hacía más legible el código, y complicaba la comprensión y uso del sistema aún más.

Por este motivo, se decide hacer uso del patrón de diseño **Facade** (fachada) [14], para reducir la complejidad de la inicialización de los Transformers. En una subclase del Step llamada PysparkStep decido añadir métodos (uno por Transformer) que inicializan y agregan al Step cada uno de los Transformers implementados, permitiendo que la existencia de estos Transformadores sea transparente para el usuario inexperto. De este modo se consigue un código mucho más legible, como se verá en ejemplos posteriores.

Por otro lado, esta decisión tiene sus desventajas:

- La clase PysparkStep está acoplada a cada uno de los Transformers implementados, por lo que se incumple el principio de Inversión de Dependencia de SOLID.
- Incumple el principio de Responsabilidad Única de SOLID, ya que el PysparkStep tiene la responsabilidad del Step, y además la de proveer una interfaz amigable de creación de transformadores.
- Para mantener la interfaz amigable, requiere que se añadan nuevos métodos al PysparkStep a medida que nuevos Transformers sean implementados.

A pesar de que estas desventajas son graves, dado que solo afectan a una clase de todo el sistema de la que no depende ninguna, el resto queda intacto y se puede usar con todas las ventajas de no incumplir los principios de diseño mencionados. Incumplir estos principios en una pieza pequeña permite satisfacer las historias de usuario, lo que da sentido al sistema. De no ofrecer una interfaz realmente amigable, todo el sistema sería inútil.

El diagrama de clases, tras tomar esta decisión, sería como muestra la figura 11.

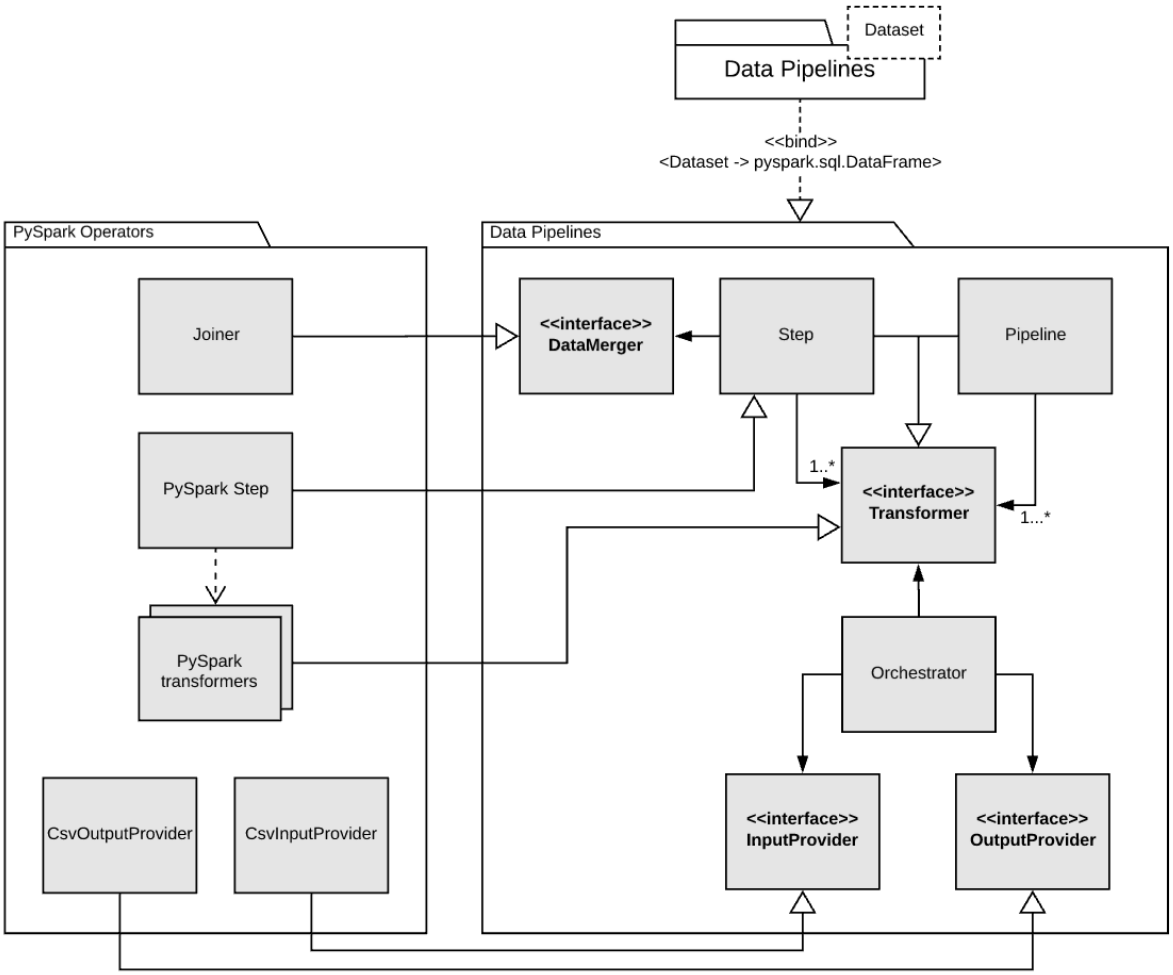


Figura 11: Diagrama de clases de los paquetes Data Pipelines y PySpark Operators tras añadir el PySpark Step.

4.2.2 - Diseño detallado

A continuación, se presentan diagramas de clases detallados del sistema descrito en el apartado anterior, además de una descripción para cada una de las clases del mismo.

Paquete genérico Data Pipelines

Diagrama de clases

El diagrama de clases del paquete Data Pipelines se muestra en la Figura 12.

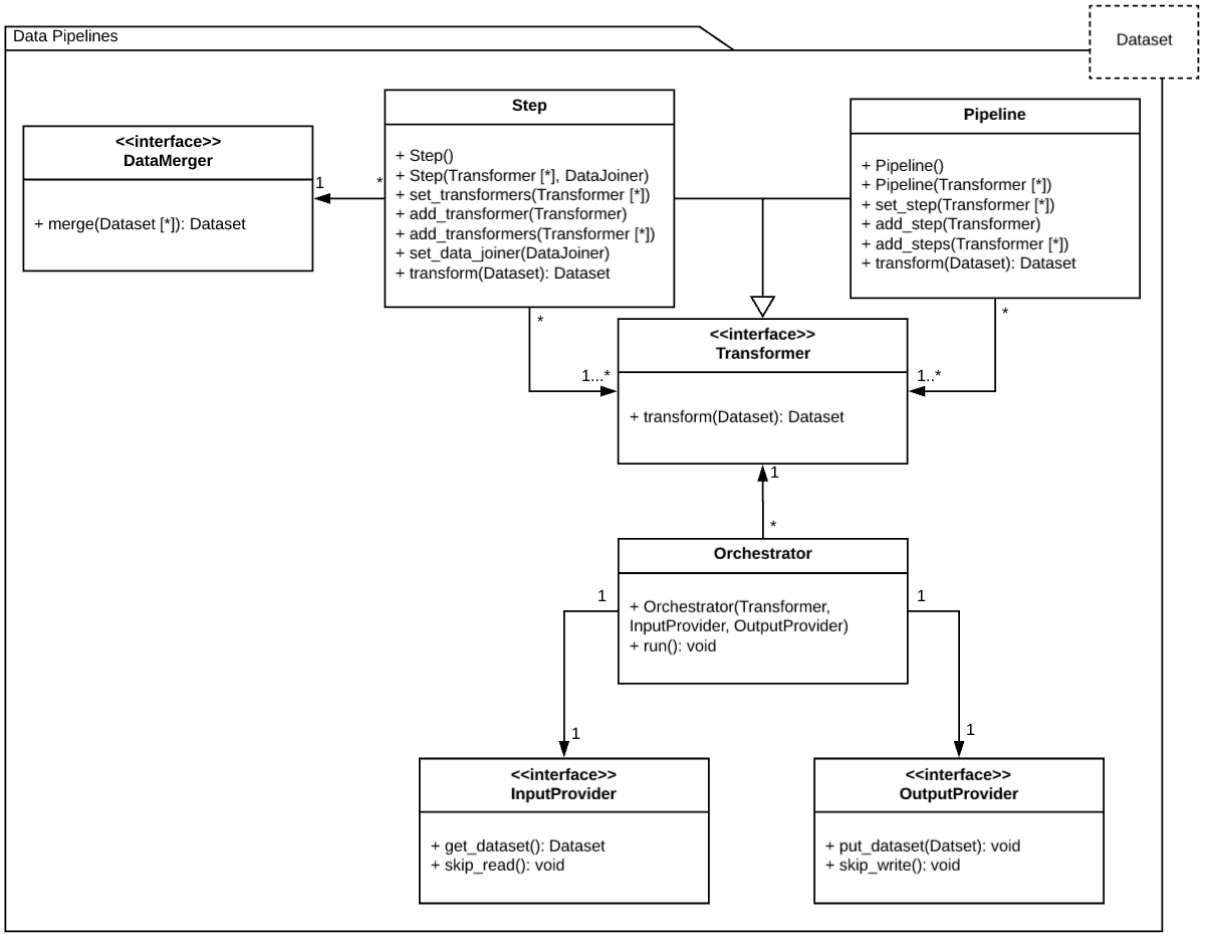


Figura 12: Diagrama de clases detallado del paquete Data Pipelines.

Descripción de las clases:

Dado que la función de cada clase de este paquete se explica con detalle en la sección anterior, sólo serán mencionados algunos detalles sobre cada una.

Transformer

Interfaz que implementarán las clases de objetos capaces de procesar un Dataset y devolver otro como resultado.

- Esta característica de los Transformers se realiza en el método transform(Dataset): Dataset.

Step

Transformer que contiene a otros Transformers y a un DataMerger. Su transformación consiste en aplicar cada Transformer al Dataset de entrada y agregar todos los resultados usando el DataMerger.

- Para facilitar diferentes sintaxis a la hora de inicializar el Step, se ofrecen dos constructores, uno vacío y otro que tiene como parámetros la lista de Transformers y el DataMerger.
- Con el mismo objetivo se dan distintas opciones para definir la lista de Transformers:
    - set_transformers(Transformer [*]) sustituye la lista de Transformers del step por la lista dada.
    - add_trasnformer(Transformer) añade el transformador dado a la lista del Step-
    - add_transformers(Transformer [*]) añade los transformadores dados a la lista del Step.

Pipeline

Transformer que contiene a otros Transformers, ordenados. Su transformación consiste en aplicar el primer transformador al Dataset de entrada, y usar el resultado de este como entrada del segundo. La salida del último transformador es el resultado que devuelve el Pipeline.

- Al igual que en el caso del Step, se ofrecen diferentes constructores y métodos para añadir Transformadores al Pipeline.
- Para hacer el uso del Pipeline más intuitivo para los usuarios que no estén familiarizados con el concepto de Transformador, los métodos que agregan transformadores al Pipeline han sido llamados add_step en lugar de add_transformer, para simbolizar la idea intuitiva de añadir pasos al procesado que realiza el Pipeline.

DataMerger

Interfaz de la que heredarán las clases de objetos capaces de realizar una operación que toma varios Datasets de entrada y tiene uno nuevo como salida. Serán principalmente utilizados por el Step para agregar resultados de diferentes transformaciones.

- Esta capacidad es realizada en el método merge(Dataset [*]): dataset

InputProvider

Interfaz de la que heredarán las clases de objetos capaces de proveer de Datasets al sistema.

- Esta capacidad es realizada en el método get_dataset(void): Dataset. El comportamiento esperado del InputProvider es que devuelva cada Dataset una única vez, pasando al siguiente con cada llamada a este método y devolviendo un valor nulo cuando no queden más Datasets por leer.
- Como herramienta para manejar este cambio de estado se da el método skip_read(void): void, que saltará un Dataset sin leerlo.

OutputProvider

Interfaz de la que heredarán las clases de objetos capaces de efectuar la salida de Datasets del sistema.

- Esta capacidad es realizada en el método put_dataset(Dataset): void.
- En caso de que el OutputProvider disponga de una lista de localizaciones donde enviar o escribir cada Dataset, como herramienta para manejar el cambio de estado correspondiente se da el método skip_write(void): void, que saltará una localización para la escritura del Dataset.

Orchestrator

Clase de los objetos encargados de gestionar la lectura, transformación y escritura de Datasets.

- El objeto se inicializa especificando un InputProvider como fuente de Datasets, un Transformer como método de procesado y un OutputProvider como destino de los resultados.
- El método run(void): void del Orchestrator pedirá Datasets al InputProvider hasta que este no pueda proveer más (devuelva un valor nulo), los transformará usando su Transformer, y los escribirá usando el OutputProvider.
- Ante un error en el procesado de un Dataset, el Orchestrator deberá proseguir con el procesado del siguiente, si hubiera alguno. Esto evita la interrupción de un procesado de muchos Datasets si alguno de ellos es defectuoso u ocurre otro error inesperado.

Paquete PySpark Operators

Provee al paquete Data Pipelines de la funcionalidad necesaria para el procesado de DataFrames de PySpark.

Diagrama de clases

En el diagrama, presente en la Figura 13, se omiten las relaciones entre este paquete y el paquete Data Pipelines ya que han sido mostradas en la Figura 11. Omitirlas nos permitirá ver más claramente los detalles. También se han mantenido agregados todos los PySpark Transformers, ya que se dará detalle sobre ellos más adelante.

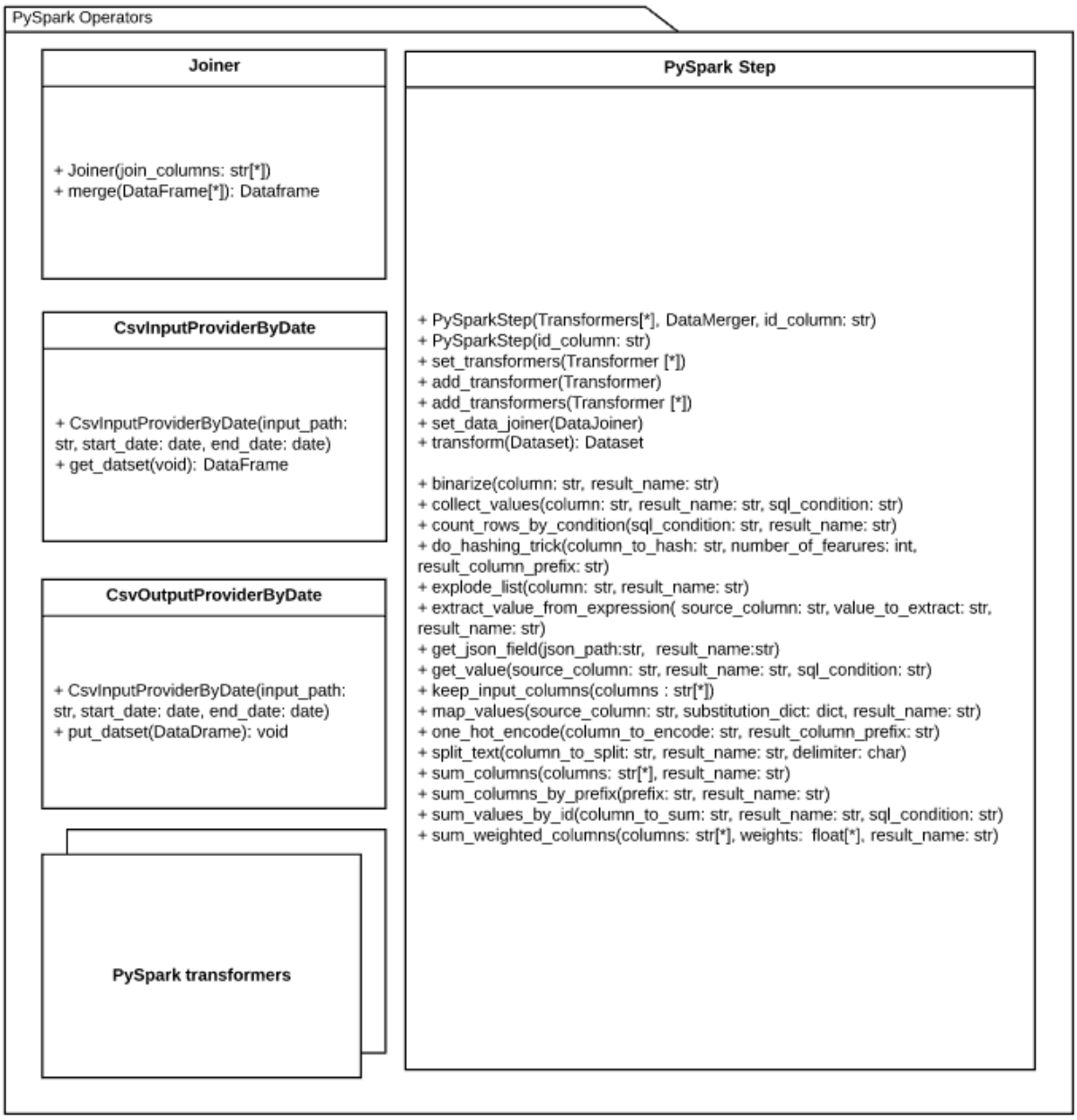


Figura 13: Diagrama de clases detallado del paquete PySpark Operators.

Descripción de las clases

Joiner

DataMerger que realiza left-outer equijoins sobre una columna común para agregar todos los DataFrames de entrada en uno de salida. Este es el único método de agregado de DataFrames que es requerido por los requisitos, según especifica la historia de usuario HU-031.

- En su constructor deben especificarse los nombre de columnas comunes en todos los DataFrames que deben ser utilizados para realizar las operaciones join.

CsvInputProviderBydate

InputProvider que lee DataFrames del sistema de ficheros, en formato CSV, y siguiendo una estructura de subdirectorios correspondiente a una fecha, con el siguiente formato: *año/añomes/añomesdía.*

- En el constructor se especifica la ruta raíz y el rango de fechas a leer.
- Cada llamada a get_dataset leerá y devolverá el DataFrame correspondiente a la fecha actual, y la fecha avanzará un día para la siguiente lectura.

CsvOutputProviderBydate

OutputProvider que escribe DataFrames al sistema de ficheros, en formato CSV, y siguiendo una estructura de subdirectorios correspondiente a una fecha, con el siguiente formato: *año/añomes/añomesdía.*

- En el constructor se especifica la ruta raíz y el rango de fechas donde escribir.
- Cada llamada a put_dataset escribirá el DataFrame en la ruta correspondiente a la fecha actual, y la fecha avanzará un día para la siguiente escritura.

PySparkStep

Esta clase es subclase del Step. Incluye métodos que permiten añadir comportamiento al procesado realizado por el step, como los métodos sum_columns o one_hot_encode. Cada método corresponde a la funcionalidad de un Transformador, y al invocarlo ese transformador será añadido a la lista de transformadores del PysparkStep, con los parámetros adecuados.

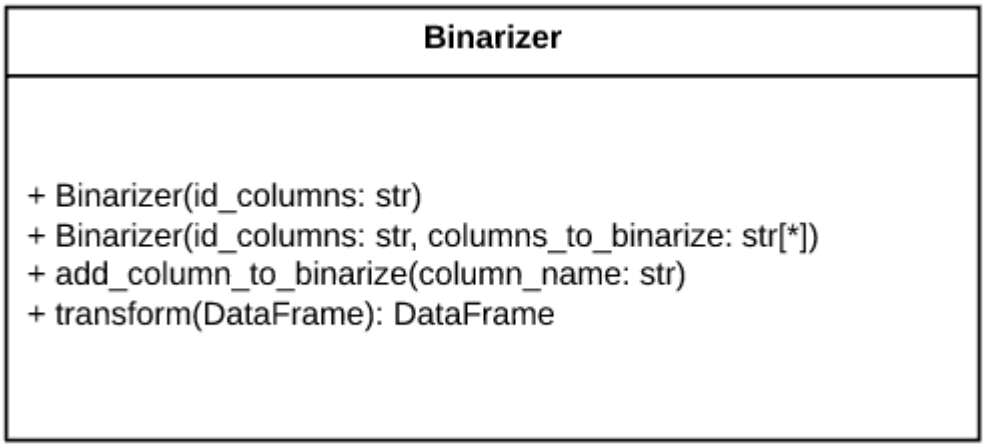
- No todas las llamadas a uno de estos métodos ayudantes añade un nuevo Transformador. Por ejemplo, la llamada al método sum_columns(columns, result_name) provoca que durante la transformación realizada por el step se genere una columna llamada *result_name* que contiene la suma de las columnas especificadas. La primera llamada a este método creará un nuevo Transformador llamado ColumnsSummator, que se encarga de realizar la suma. Llamadas posteriores a este método solo modificará el ColumnsSummator, ya que un solo transformador se encarga de realizar todas las sumas. Esto permite realizar optimizaciones que mejoran notablemente el rendimiento del sistema.

Pyspark Transformers

Aquí se incluyen los transformadores que contienen los algoritmos de procesamiento de datos requeridos según las historias de usuario desde HU-010 hasta HU-028. Estos transformadores deberán proveer resultados que sean agregables mediante operaciones join (de modo que puedan ser utilizados junto a un objeto de la clase Joiner). Por este motivo, la mayoría devolverán un DataFrame que solamente contendrá las columnas con los resultados del procesado, así como las que contienen los identificadores que se especifiquen.

A continuación, se presenta una breve especificación del comportamiento esperado por cada uno de estos transformadores.

Binarizer



Este transformador ‘binariza’ los valores de una o más columnas, dejándolos sus celdas a 0 si:

- Contiene una cadena de texto vacía
- Contiene un cero
- Contiene una lista vacía
- Contiene un valor nulo

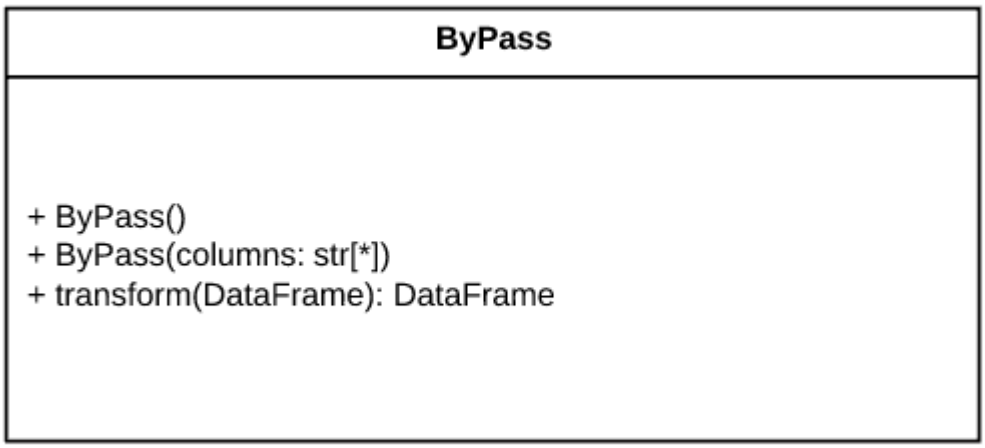
Ejemplo de entrada:

Identificador	**Columna a binarizar 1**	**Columna a binarizar 2**	**(... otras columnas)**
1	“”	12	(...)
2	“¡Hola mundo!”	Null	(...)
3	“¡Bienvenido!”	0	(...)

Salida esperada:

Identificador	**Columna resultado 1**	**Columna resultado 2**
1	0	1
2	1	0
3	1	0

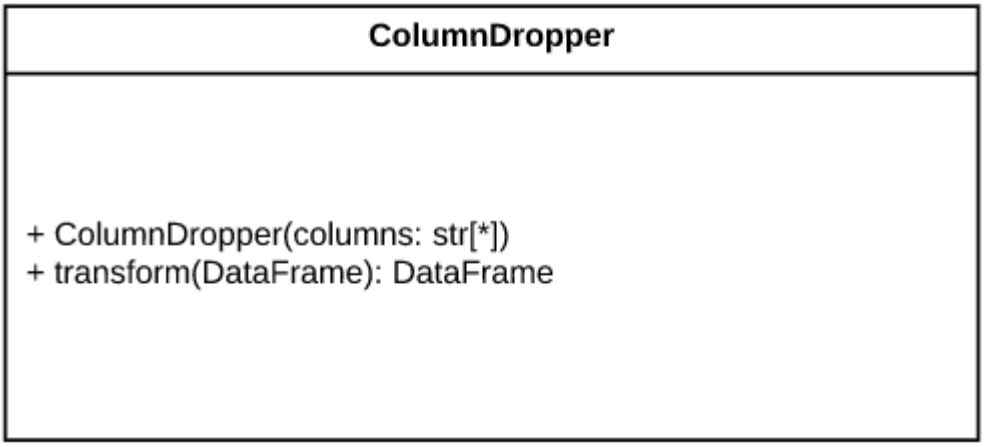
ByPass



Este transformador devuelve algunas o todas las columnas del DataFrame de entrada, sin modificarlas.

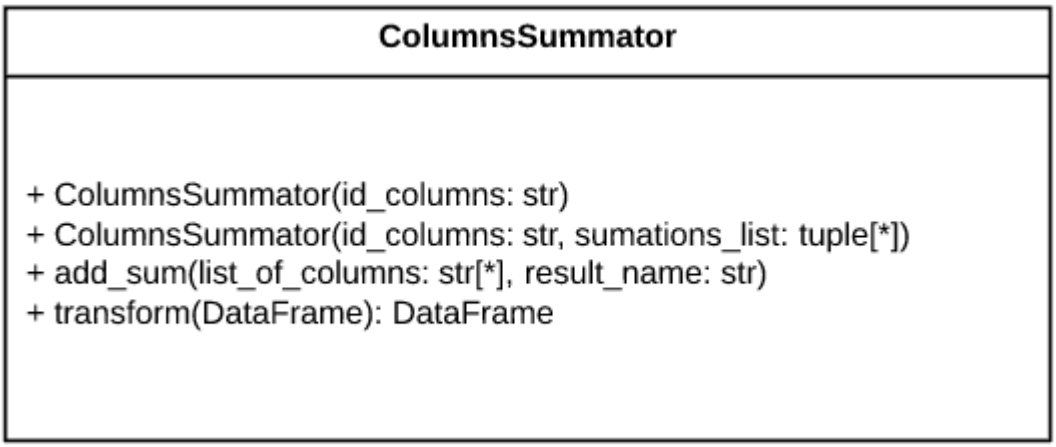
- Si se crea mediante el constructor vacío, el DataFrame de salida será el mismo que el de entrada, simboliza una transformación vacía.
- Si se crea especificando una lista de columnas, solo esas columnas serán devueltas en el DataFrame de salida.

ColumnDropper



Elimina columnas del DataFrame de entrada, manteniendo el resto de los datos inalterados.

ColumnsSummator



Devuelve una columna con la suma de dos o más columnas numéricas del DataFrame de entrada. Si se especifican varias sumas, devolverá varias columnas, cada una conteniendo el resultado de una de las sumas.

- Estas sumas pueden especificarse mediante llamadas sucesivas al método add_sum.
- También pueden especificarse creando el ColumnsSummator mediante el constructor que contiene el parámetro summations_list, dando los parámetros de cada suma como una tupla.

Este esquema para la inicialización se repetirá en varios de los próximos transformadores.

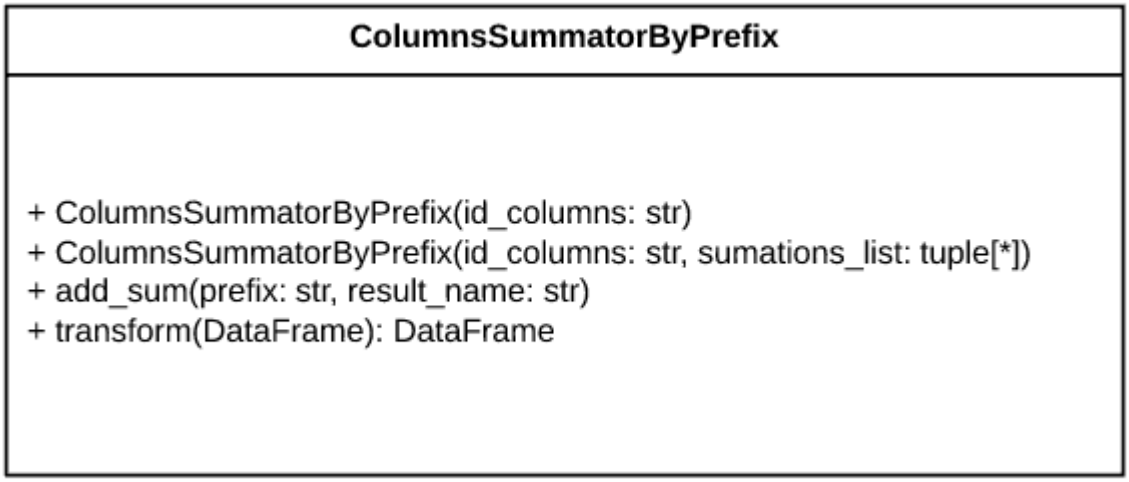
Ejemplo de entrada:

ID	**A**	**B**	**C**	**D**	**(... otras columnas)**
1	5	3	4	Null	(...)
2	7	2	2	1	(...)

Salida de un ColumnsSummator que debe realizar las sumas A+B y B+C+D:

ID	A+B	B+C+D
1	8	7
2	9	5

ColumnsSummatorByPrefix



Suma las columnas del DataFrame de entrada cuyos nombres comienzan por un prefijo determinado.

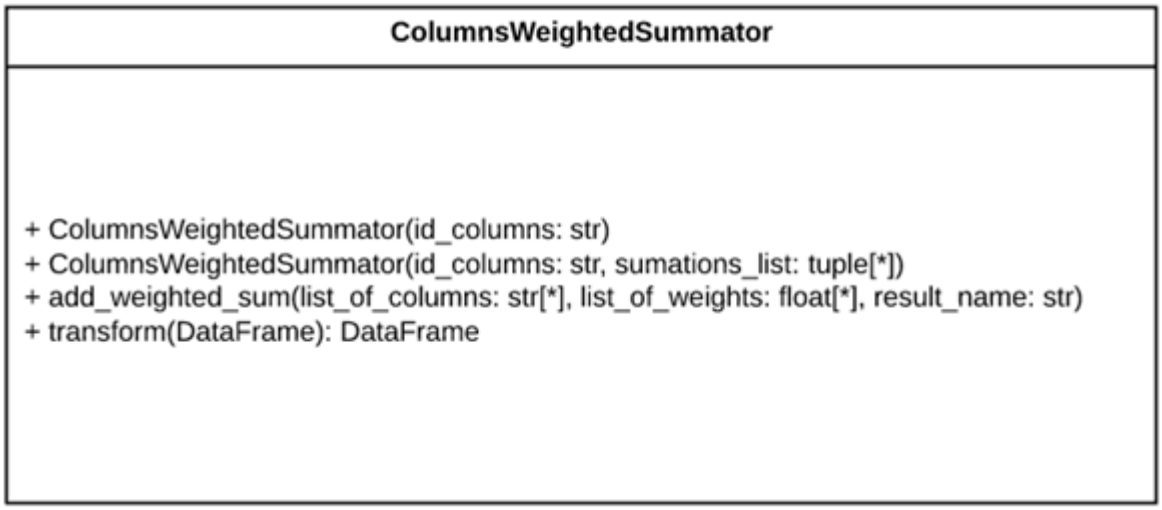
Ejemplo de entrada:

ID	AA	AB	ABC	DD	(... otras columnas)
1	4	5	7	8	(...)
2	3	4	6	7	(...)

Ejemplo de salida de un ColumnsSummatorByPrefix que debe sumar las columnas con prefijos “A”, “AB” y “D” :

ID	A	AB	D
1	16	12	8
2	13	10	7

ColumnsWeigtedSummator



Suma dos o más columnas numéricas, multiplicando cada una por un peso determinado.

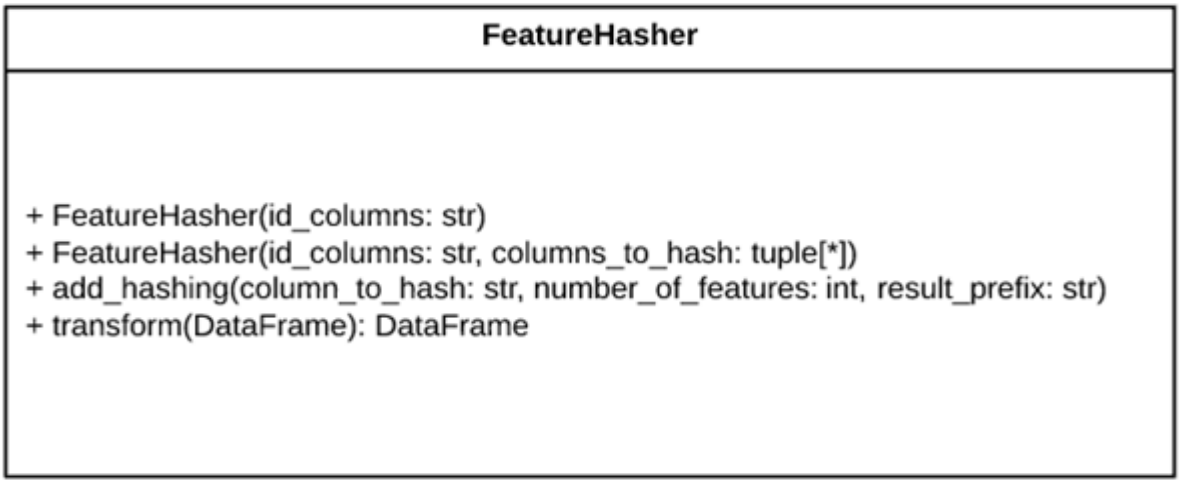
Ejemplo de entrada:

ID	**A**	**B**	**C**	**D**	**(... otras columnas)**
1	5	3	4	Null	(...)
2	7	2	2	1	(...)

Ejemplo de salida si las sumas a realizar son: 0.5*A + 2*B y 0.1*C + 0.1*D

ID	**0.5*A + 2*B**	**0.1*C + 0.1*D**
1	8.5	0.4
2	7.5	0.3

FeatureHasher



Realiza *feature hashing* sobre una columna.

- El feature hashing [15] se aplica sobre una columna con valores categóricos, y genera varias columnas, tantas como el valor del parámetro *number_of_features*. Para cada fila, una de esas columnas tendrá un valor de 1, y las demás, de 0. La columna que contiene el valor 1 se determina mediante el hasheado del valor categórico correspondiente.
- Es conveniente que el *number_of_features* sea una potencia de dos [16].

Ejemplo de entrada:

Identificador	**Valores a hashear**	**(... otras columnas)**
1	1	(...)
2	2	(...)
3	3	(...)
4	4	(...)
5	5	(...)

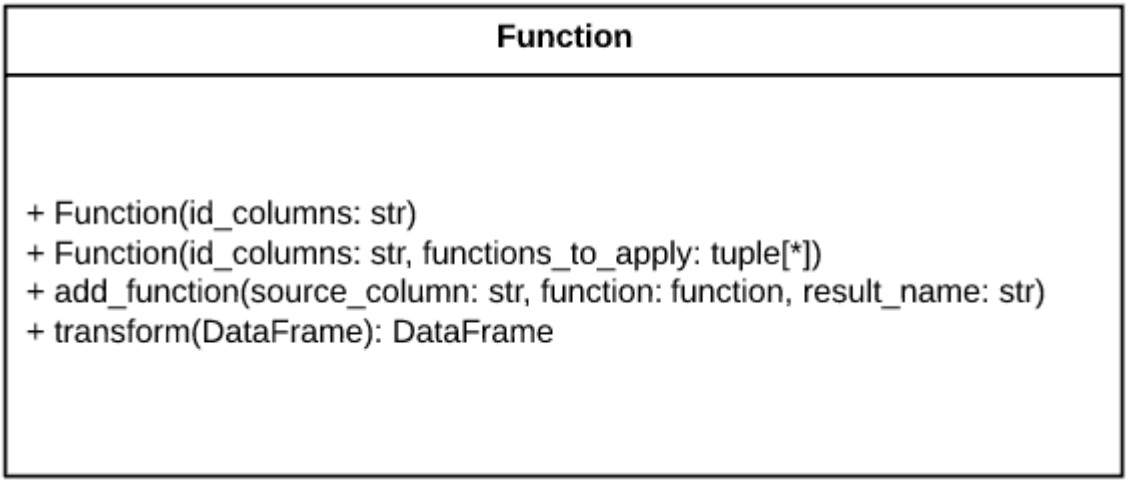
Ejemplo de salida si *number_of_features*=2 y la función hash es h(x) = x mod 2 :

Identificador	**feature_1**	**feature_2**
1	0	1
2	1	0
3	0	1
4	1	0
5	0	1

Este es un caso irreal, ya que el algoritmo utilizado realmente es *MurmurHash3_x86_32*, de Austin Appleby [16].

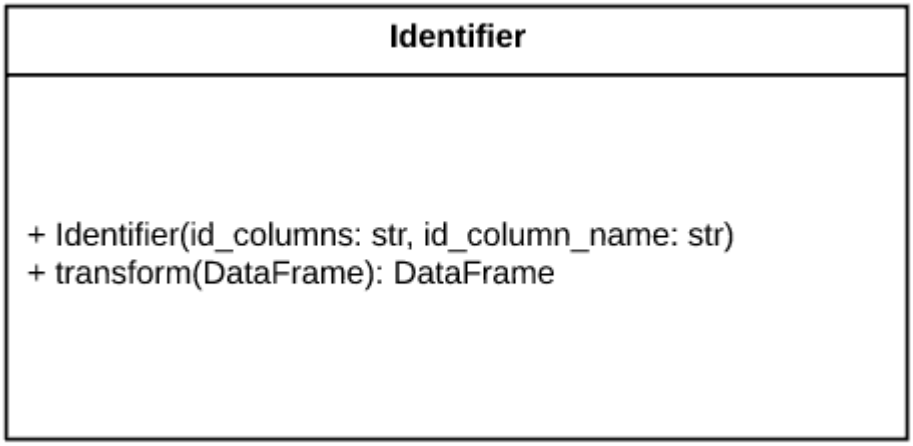
Este transformador es especialmente útil para adaptar columnas con muchos valores categóricos distintos a su uso por algoritmos de aprendizaje automático.

Function



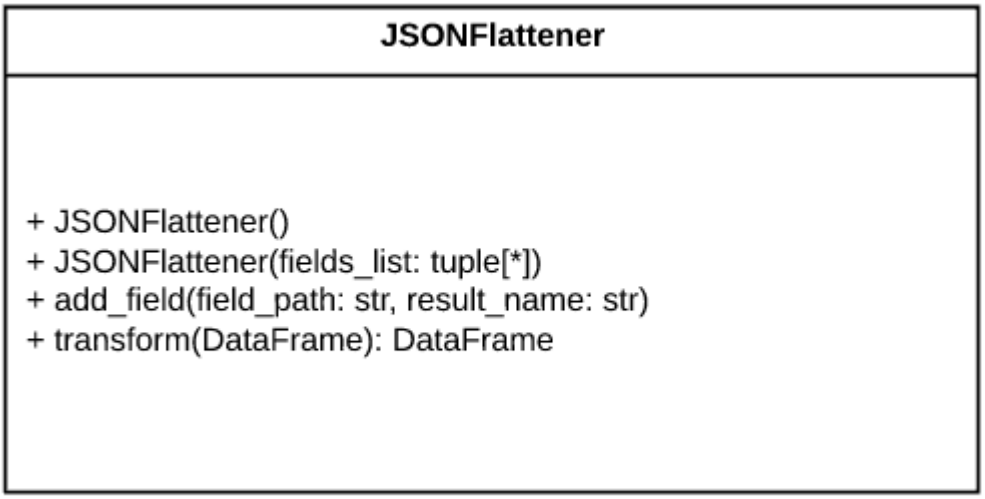
Este transformador aplica una función definida por el usuario a cada elemento de una columna. Su eficiencia depende de la de la función a aplicar.

Identifier



Devuelve el DataFrame de entrada, con una nueva columna conteniendo un identificador único para cada fila.

JSONFlattener



Los DataFrames de PySpark pueden contener datos en formato JSON, por lo que los valores de interés pueden estar contenidos en una estructura de objetos dentro de otros objetos. Este transformador genera una columna que contiene para cada fila el valor encontrado en una ruta JSON concreta.

Ejemplo de entrada: DataFrame que contiene los siguientes objetos JSON

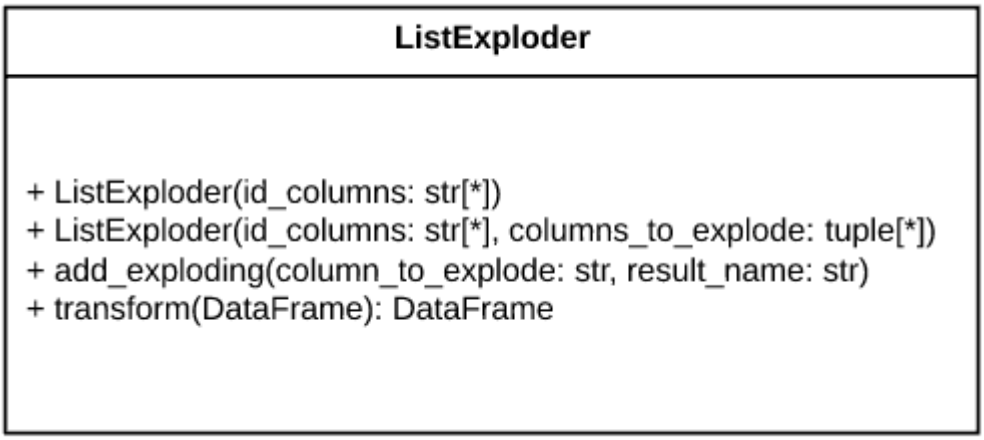
{ id: 1, params: {'name': 'Oliver', 'age': 21} },
{ id: 2, params:{'name': 'Luis', 'age': 78} }

Ejemplo de salida si las rutas a obtener son:

- /id
- /params/name

id	**params/name**
1	Oliver
2	Luis

ListExploder



Dada una columna de entrada que contiene listas, genera una nueva columna con una fila por cada elemento de cada lista.

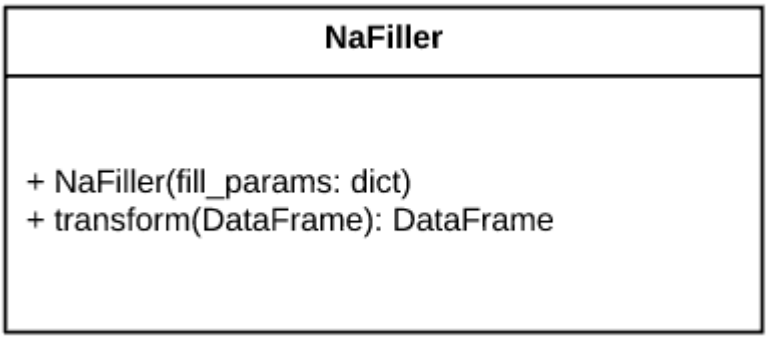
Ejemplo de entrada:

Identificador	**Lista a explotar**	**(... otras columnas)**
1	'a', 'b', 'c'	(...)
2	'a', 'b'	(...)

Salida correspondiente:

Identificador	Resultado
1	‘a’
1	‘b’
1	‘c’
2	‘a’
2	‘b’

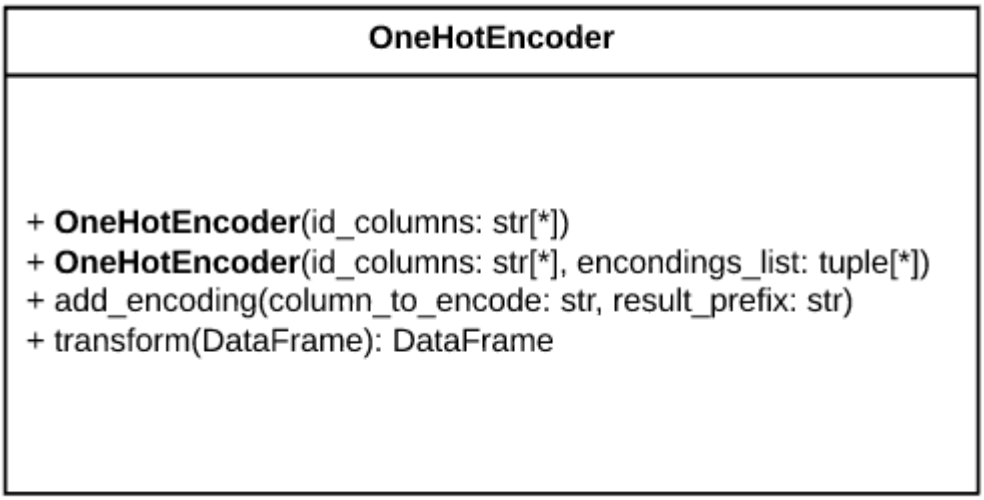
NaFiller



Sustituye los valores nulos en una columna por un valor dado.

- El parámetro fill_params contiene un diccionario en el que cada par (clave, valor) corresponde a un par (columna donde sustituir nulos, valor por el que sustituirlos).
- El NaFiller modifica las columnas correspondientes a los *fill_params*, siendo el resto del DataFrame de salida idéntico al de entrada.

OneHotEncoder



Realiza *OneHotEncoding* sobre una columna.

- El OneHotEncoding es similar al FeatureHashing, con la distinción de que usando OneHotEncoding cada valor categórico en la entrada genera una nueva columna en la salida [17].

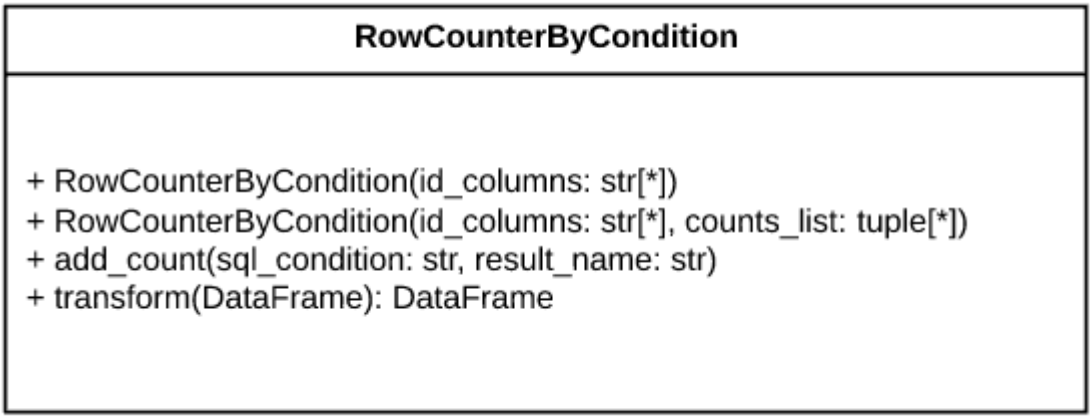
Ejemplo de entrada:

ID	Columna de entrada	(... otras columnas)
1	‘a’	(...)
2	‘a’	(...)
3	‘b’	(...)
4	‘c’	(...)
5	‘c’	(...)
6	‘a’	(...)

Salida esperada:

ID	encoding_a	encoding_b	encoding_c
1	1	0	0
2	1	0	0
3	0	1	0
4	0	0	1
5	0	0	1
6	1	0	0

RowCounterByCondition



Este Transformador cuenta, para cada ID, el número de columnas que cumple una cierta condición SQL.

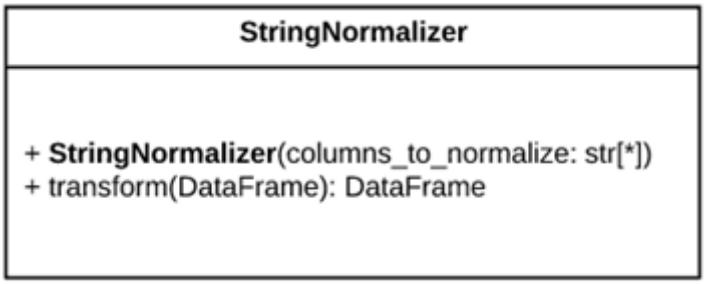
Ejemplo de entrada:

Identificador	Value	(... otras columnas)
1	100	(...)
1	100	(...)
1	200	(...)
2	100	(...)
2	900	(...)
3	12	(...)

Salida esperada para la condición: “Value == 100” :

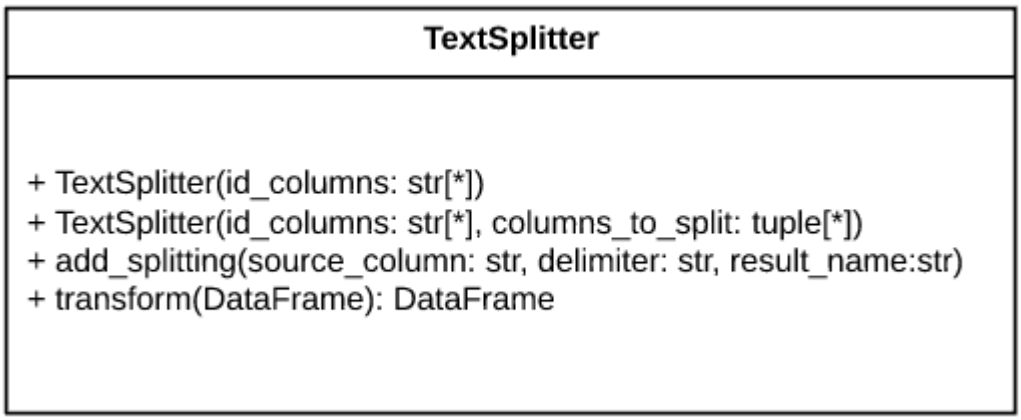
Identificador	Resultado
1	2
2	1
3	0

StringNormalizer



Este transformador elimina espacios al principio y al final de las cadenas de texto en las columnas de entrada, y pone las cadenas resultantes en minúsculas. El resto del DataFrame de salida es idéntico al de entrada.

TextSplitter



Divide cadenas de texto en listas de cadenas, usando un carácter especificado como delimitador.

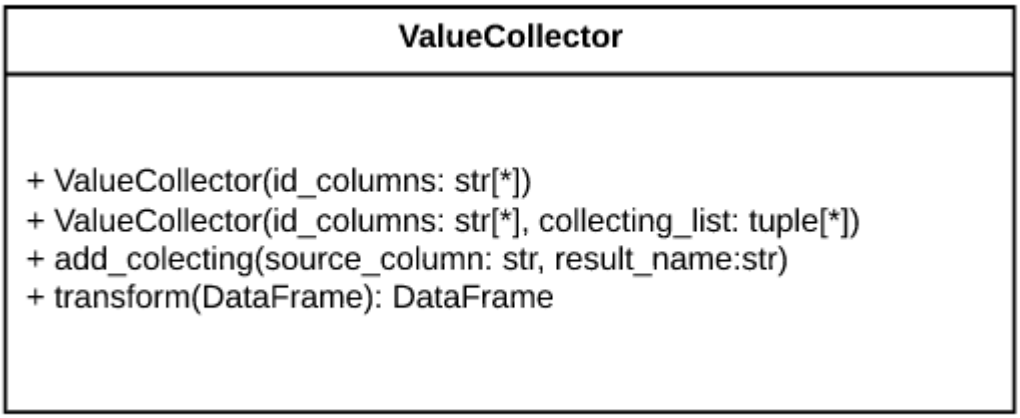
Ejemplo de entrada:

Identificador	**Columna de entrada**	**(... otras columnas)**
1	‘primero,segundo,tercero’	(...)
2	‘uno,dos,tres’	(...)

Salida correspondiente:

Identificador	**Resultado**
1	[‘primero’, ‘segundo’, ‘tercero’]
2	[‘uno’, ‘dos’, ‘tres’]

ValueCollector



Este transformador obtiene una lista de valores para cada id, recolectados de una columna dada.

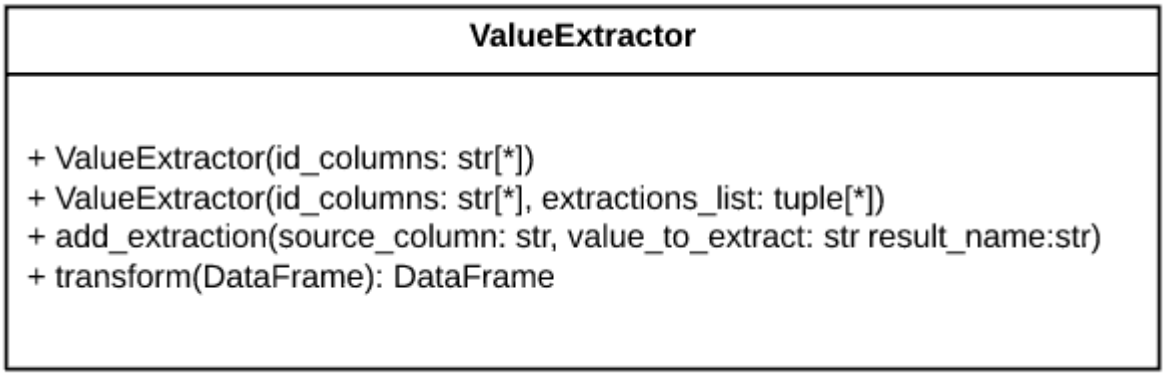
Ejemplo de entrada:

Identificador	**Columna de entrada**	**(... otras columnas)**
1	‘a’	(...)
1	‘b’	(...)
1	‘b’	(...)
2	‘a’	(...)

Salida esperada:

Identificador	**Resultado**
1	[‘a’, ‘b’, ‘b’]
2	[‘a’]

ValueExtractor



Extrae valores de una columna que contiene strings del tipo ‘clave1=valor1;clave2=valor2’

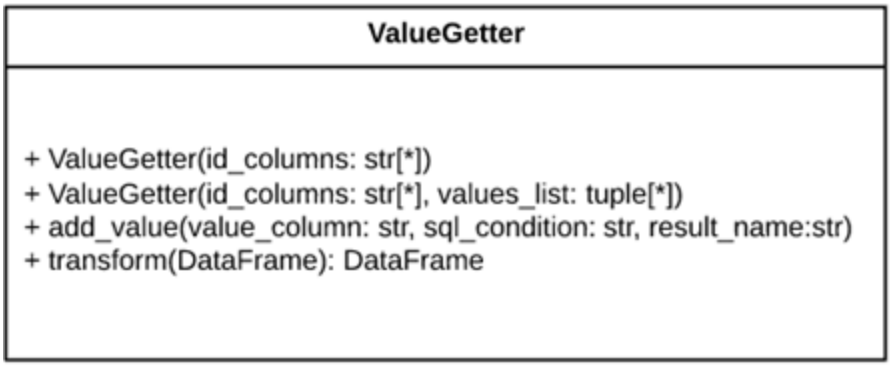
Ejemplo de entrada:

Identificador	**Columna de entrada**	**(... otras columnas)**
1	‘nombre=Paco;edad=10’	(...)
2	‘nombre=Pepe;edad=22’	(...)
3	‘nombre=Ana;edad=31’	(...)
4	‘nombre=María;edad=45’	(...)

Salida correspondiente si hay que extraer el valor de la clave ‘nombre’:

Identificador	**Nombre**
1	Paco
2	Pepe
3	Ana
4	María

ValueGetter



Genera una nueva columna con los valores de filas que cumplen una cierta condición SQL.

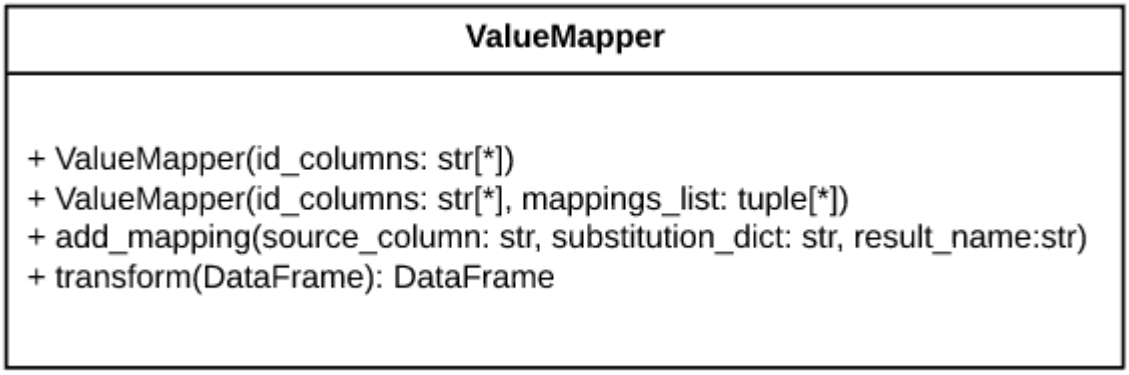
Ejemplo de entrada:

Identificador	**Dato**	**Valor**	**(... otras columnas)**
1	‘nombre’	‘Óliver’	(...)
1	‘edad	21	(...)
2	‘nombre’	‘Luis’	(...)
2	‘edad’	45	(...)

Salida correspondiente si tomamos los valores de la columna Valor según la condición “Dato == ‘edad’”:

Identificador	**Resultado**
1	21
2	45

ValueMapper



Este transformador sustituye los valores encontrados en una columna según un diccionario de sustituciones en el que en cada par “clave, valor”, la “clave” es el valor encontrado y el “valor”, el valor por el que sustituirlo.

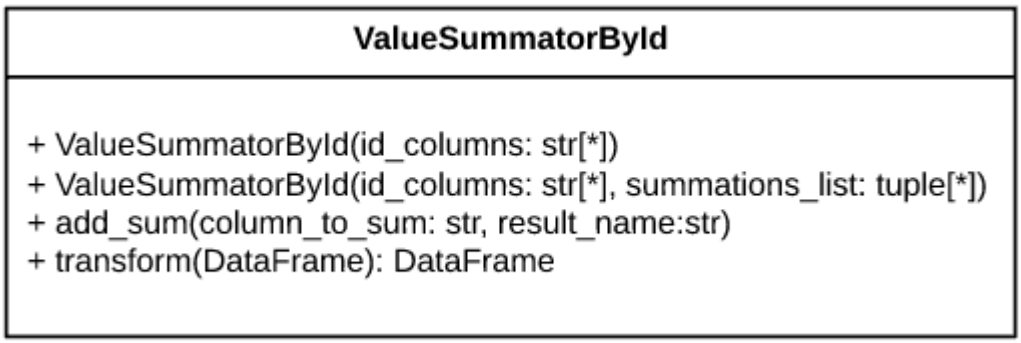
Entrada de ejemplo:

Identificador	**Columna de entrada**	**(... otras columnas)**
1	‘Madrid’	(...)
2	‘ES’	(...)
3	‘Espanya’	(...)
4	‘NY’	(...)
5	‘Estados Unidos’	(...)
6	‘USA’	(...)

Salida para el diccionario de sustituciones
{‘Madrid’: ‘España’,
‘ES’: ‘España’,
‘Espanya’: ‘España’,
‘NY’: ‘EEUU’,
‘Estados Unidos’: ‘EEUU’,
‘USA’: ‘EEUU’}:

Identificador	**Resultado**
1	‘España’
2	‘España’
3	‘España’
4	‘EEUU’
5	‘EEUU’
6	‘EEUU’

ValueSummatorById



Este transformador suma todos los valores encontrados en una columna numérica para cada identificador.

Ejemplo de entrada:

Identificador	**Columna a sumar**	**(... otras columnas)**
1	2	(...)
1	4	(...)
1	1	(...)
1	5	(...)
2	6	(...)
2	8	(...)

Salida correspondiente:

Identificador	**Suma**
1	12
2	14

4.2.3 - Diseño de la interacción

En esta sección se describe la interacción entre las clases de la biblioteca en un ejemplo de uso modelo, en el que el usuario escribe un script que realiza las siguientes tareas:

1. Crear un nuevo Pipeline.
2. Crear dos PysparkStep.
3. Poblar estos Step con transformaciones usando su fachada para la configuración de Transformers. Los Transformers de cada PysparkStep serán:
    a. PysparkStep 1:
        i. Dos cuentas de columnas que cumplen ciertas condiciones SQL.
        ii. Una obtención de valores.
    b. PysparkStep 2:
        i. Una suma de columnas.
4. Añadir los Steps al Pipeline.
5. Crear un nuevo CsvInputProviderByDate.
6. Crear un nuevo CsvOutputProviderByDate.
7. Crear un nuevo Orchestrator usando el Pipeline, Input y OutputProvider creados.
8. Ejecutar el método run del Orchestrator.

La descripción de este proceso se realizará mediante varios diagramas de interacción.

Diagrama general (Figura 14)

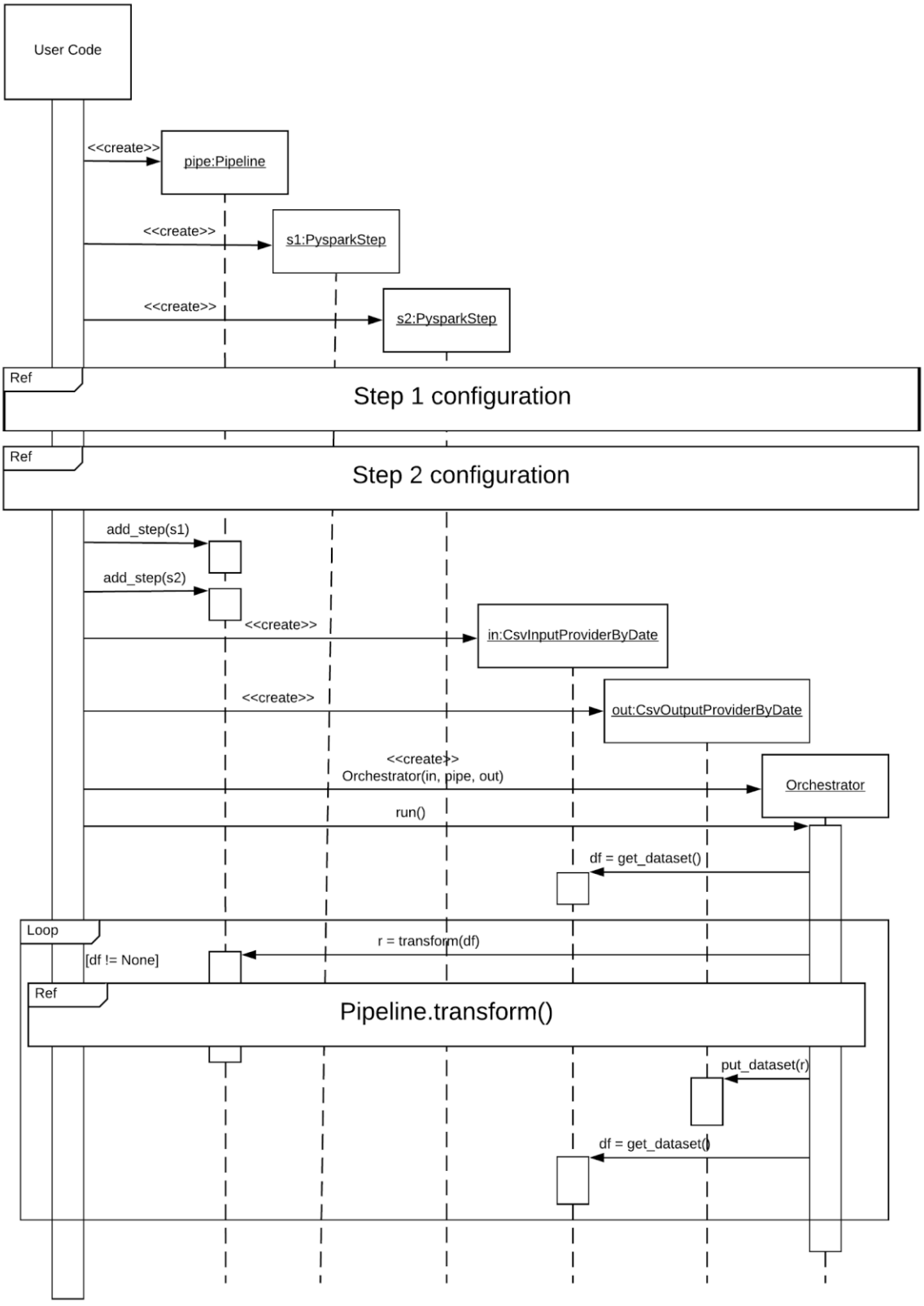


Figura 14: Diagrama de interacción del ejemplo de uso.

Configuración del Step 1 (Figura 15)

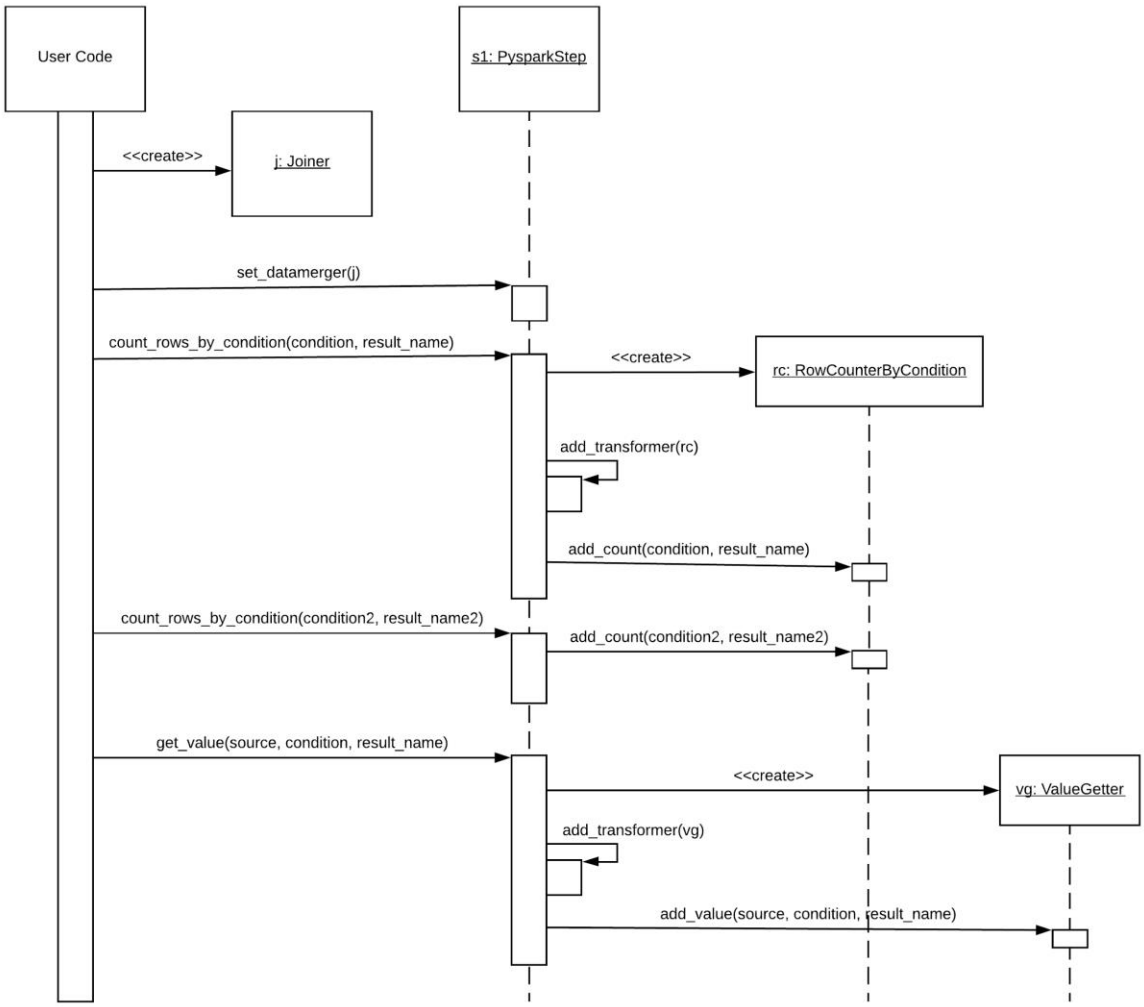


Figura 15: Diagrama de interacción de la configuración del Step 1.

Configuración del Step 2 (Figura 16)

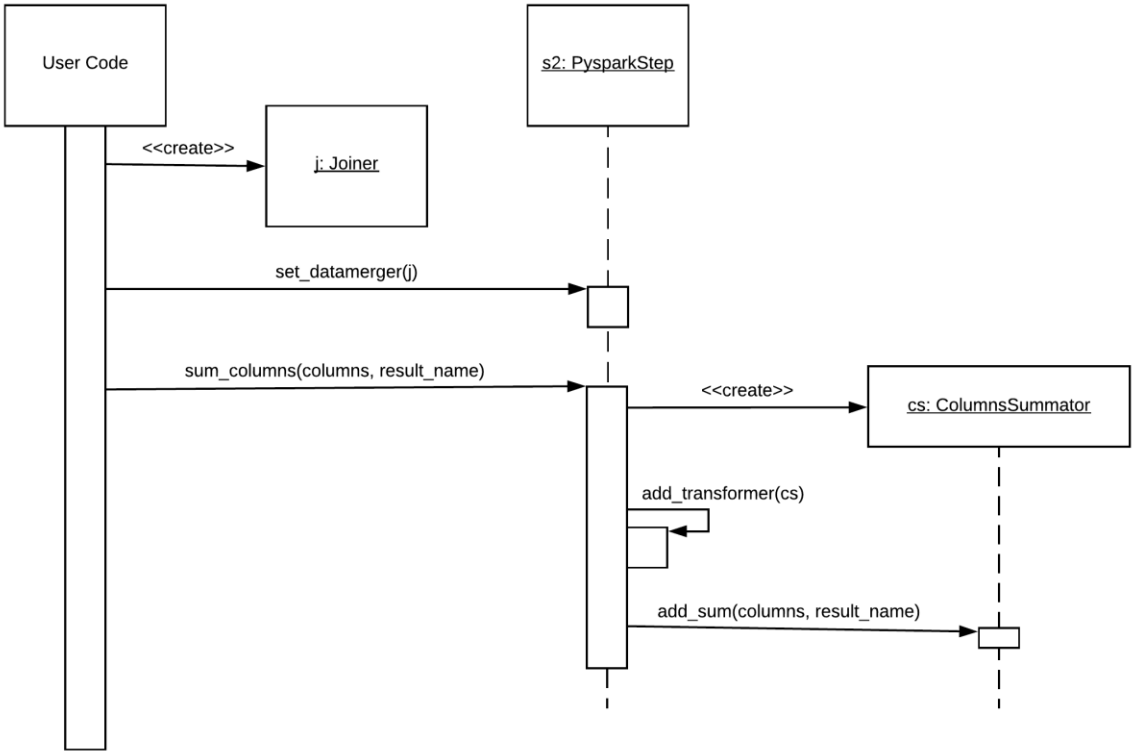


Figura 16: Diagrama de interacción de la configuración del Step 2.

Ejecución del método Orchestrator.run (Figura 17)

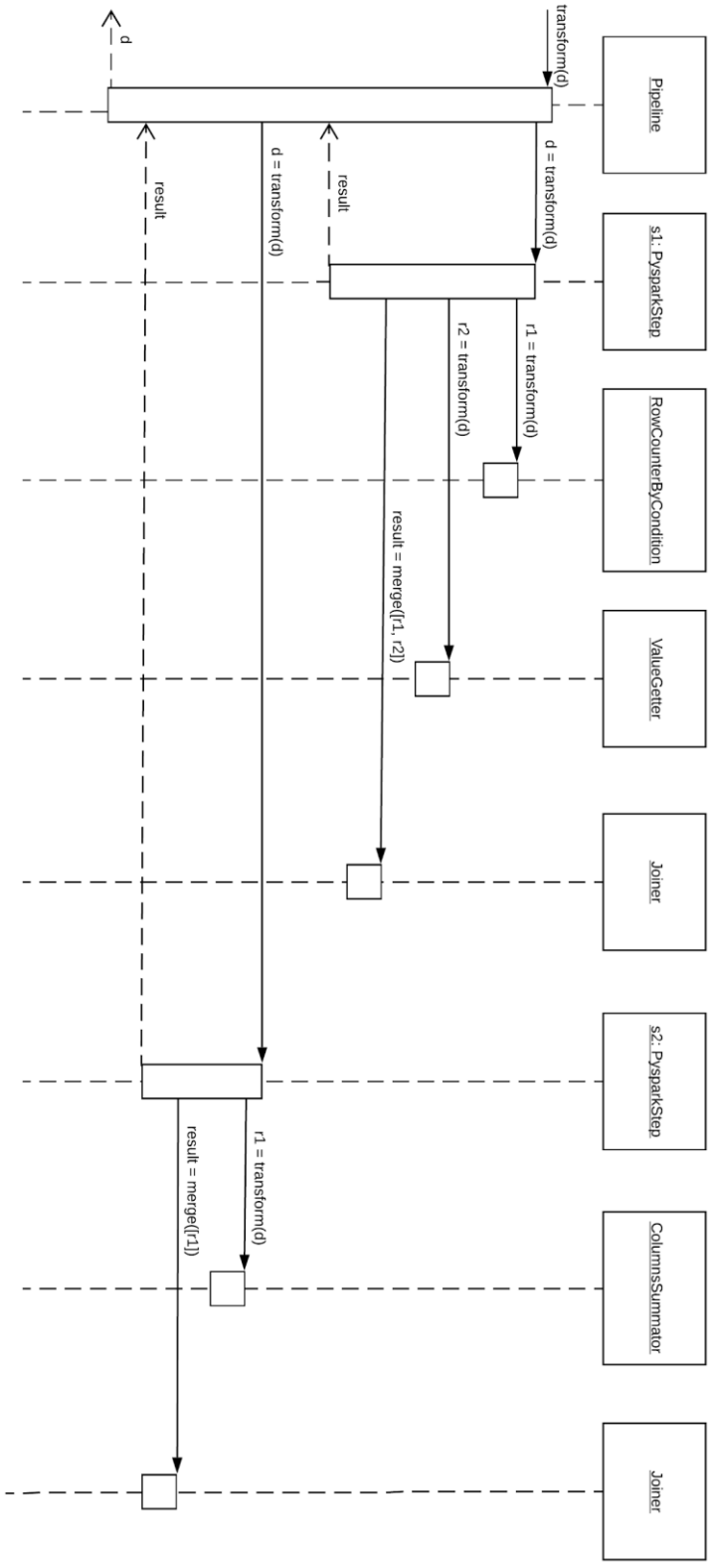


Figura 17: Diagrama de interacción del método Orchestrator.run

4.3 - Implementación

Debido a las características de Python (el lenguaje utilizado para la implementación de este sistema), el diseño desarrollado durante la sección anterior no está expresado con exactitud en la implementación. En esta sección se detalla cómo el diseño ha sido adaptado durante el proceso de implementación a las herramientas utilizadas, para aprovechar sus fortalezas y seguir sus estándares de uso.

No se expondrá durante esta sección una descripción detallada del sistema implementado, ya que exceptuando las diferencias que voy a resaltar es idéntico al diseño. Para encontrar una descripción de la interfaz de programación del sistema puede acudirse al anexo B: documentación de la biblioteca.

No se detalla tampoco en esta sección el comportamiento interno ni miembros privados de cada clase, como tampoco se ha hecho en la sección de diseño. Esto es así porque, siguiendo la filosofía de desarrollo de Python, he llevado a cabo la implementación con el objetivo de que el código sea legible y auto-documentado, evitando incluso los comentarios en la medida de lo posible.

Tipado dinámico y débil

En Python no hay tipado estático, y las referencias pueden apuntar a objetos de diferentes tipos durante su tiempo de vida. Es posible hacer comprobaciones de tipo en tiempo de ejecución, pero va en contra del estilo Python aceptado, que se basa en una buena documentación del código y en el *duck typing*. (El duck typing es un sistema de tipificación que asume el tipo de los objetos en función de sus métodos. Su nombre viene de la frase: "Cuando veo un ave que camina como un pato, nada como un pato y suena como un pato, a esa ave yo la llamo un pato." [18])

Por esto, todos los tipos de los argumentos especificados en el diseño son tipos esperados, pero no obligados: valores de cualquier tipo podrían usarse como argumentos de los métodos del sistema, siempre que tengan el comportamiento que se espera de ellos.

Métodos sobrecargados

En Python no es posible definir en una clase varios métodos con el mismo nombre pero distintos argumentos. En cambio, sí es posible definir métodos con argumentos opcionales, o argumentos que aceptan valores de diferentes tipos.

Por este motivo, las clases que en el diseño tenían varios métodos con el mismo nombre, en la implementación solo tienen uno, que aporta la funcionalidad de todos los definidos en el diseño.

Clases con varios constructores

Las clases con varios constructores en el diseño solo tienen uno en la implementación, que mediante argumentos opcionales aporta la funcionalidad de todos los presentes en el diseño.

Miembros privados

En Python, no es posible, usando métodos convencionales, hacer privada una propiedad o método de una clase. Se utiliza el convenio de comenzar con una barra baja (‘_’) los nombres de los miembros privados de una clase [19]. Se espera que en un uso normal de estas clases los miembros que comienzan con una barra baja no se utilicen, pero se da la opción de hacerlo a aquellos usuarios que sepan lo que hacen.

Getters y setters

En Python es posible definir métodos getters y setters que obtengan o modifiquen el valor de una propiedad de un objeto, consiguiendo de este modo encapsulación de datos. Sin embargo, siguiendo

la forma más *Phytonic* de desarrollo, no se han implementado getters ni setters, sino que las propiedades son públicas.

Podría pensarse que de este modo no se puede ejercer ningún control sobre cómo se asigna y obtiene el valor de estas propiedades. Sin embargo, esto no es así, ya que en Python es posible modificar cómo se accede a las propiedades de un objeto mediante una sintaxis especial, de forma transparente para el usuario [20]. De este modo, tenemos las ventajas de usar getters y setters, pero con un código más legible: tanto el código del sistema, que no estará plagado de getters y setters que no tienen ninguna función más allá de una asignación o un return; como el código usuario, ya que podrá acceder diréctamente a las propiedades sin llenarse de llamadas a métodos.

Modificaciones en algunos transformadores

Durante una de las aplicaciones del sistema, al final del proceso de desarrollo, se detectó la necesidad de filtrar los datos mediante una condición SQL antes de realizar el procesado de cada transformador. Esto ocurre ante un conjunto de datos que agrupa datos muy dispares, en el que cada transformador debe aplicarse solo a un subconjunto de todo el dataset.

Para solucionar esta necesidad se valoró la opción de crear un pequeño pipeline para cada transformador, que filtrase los datos antes de aplicar el algoritmo de procesado. Esta opción se desestimó, ya que requiere sustituir cada transformador por un pipeline, complicando mucho la definición del procesado.

En su defecto, se ha optado por añadir un argumento opcional a los transformadores que lo necesitasen, que aporta una condición SQL que el propio transformador utiliza para filtrar los datos de entrada antes de procesarlos. Al ser un argumento opcional, no modifica la interfaz de los transformadores, solo la amplía, y soluciona el problema de una manera sencilla.

Modificaciones la estructura de paquetes

Para mantener ordenado el código, los paquetes que se presentan en este capítulo se han subdividido. La nueva estructura es la siguiente:

- **pipelib**: el paquete raíz y nombre de la biblioteca en el código.
    - **pipeline_components**: contiene los componentes más básicos, es decir, el paquete Data Pipelines.
    - **pyspark_step**: contiene el PysparkStep.
    - **transformers**: contiene todos los Transformers del paquete PySpark Operators.
    - **data_mergers**: contiene el Joiner.
    - **io_providers**: contiene los InputProviders y OutputProviders del paquete PySpark Operators.
    - **utils**: contiene excepciones y funciones de utilidad que usa el resto de la biblioteca.

Capítulo 5 - Aplicación de la biblioteca

Durante este capítulo se especifica un procesado de datos y se detalla el proceso seguido para su implementación utilizando la biblioteca desarrollada a lo largo de este proyecto.

Debe notarse que la aplicación que va a presentarse ha sido realizada para la empresa asociada a este proyecto, usando recursos y datos reales y confidenciales. Para describir la aplicación manteniendo la confidencialidad, se muestra un conjunto de datos de ejemplo con un esquema similar al de los datos reales, que permiten exponer la funcionalidad de la biblioteca en la misma medida que lo harían los auténticos. Algunas columnas innecesarias de los datos reales han sido omitidas para hacer el ejemplo más sencillo de comprender.

Por el mismo motivo el procesado a realizar no será exactamente igual, pero utilizará los mismos elementos de procesado y la misma estructura que el original, por lo que no habrá diferencias entre el contenido que se expondría al describir el procesado real y este procesado de ejemplo.

5.1 - Introducción

El procesado de datos a desarrollar tiene como objetivo la extracción de unas características que sean aptas para su uso en algoritmos de aprendizaje automático, por lo que cae dentro del campo del *feature engineering* o ingeniería de características [21]. Se usarán conocimientos sobre el dominio de los datos y el valor que se quiere sacar de ellos para determinar qué características son las más interesantes.

Los datos a procesar están extraídos mediante la herramienta Tealeaf, que se introduce en el capítulo 2 (Conceptos previos). Los datos son referentes a la interacción de los usuarios con una web de comercio electrónico, y vienen distribuidos en tres tablas:

- **Tabla de Sesiones**. Contiene una fila por cada sesión que presenta datos sobre la misma. Una sesión constituye la experiencia de un usuario en el sitio web, desde que entra en ella hasta que la interacción termina. De cada sesión se obtiene la siguiente información:
  - Un identificador para cada sesión. Se encuentra en la columna con nombre SESSION_KEY.
  - Datos referentes a cada sesión, como su duración, fecha y hora de inicio, etc.
- **Tabla de Eventos**. Contiene una fila por cada evento ocurrido en el sitio web. Un evento es la ocurrencia de unas determinadas condiciones en el sitio web, como que un usuario haga click en un cierto ítem, haga scroll, pase cierto tiempo viendo una misma página, etc. Cada evento pertenece a una sesión, y todos los eventos se recogen en esta tabla. La información disponible sobre cada evento es:
  - El identificador de la ocurrencia del evento. (El nombre de la columna que contiene esta información es HIT_KEY)
  - El identificador de la sesión a la que pertenece el evento. (Su columna se llama SESSION_KEY)
  - El tipo al que pertenece el evento, es decir: a qué condiciones en el sitio web responde. (Su columna se llama EVENT_ID)
  - Un valor asociado a ese evento, cuyo significado depende del tipo del evento. Por ejemplo, un evento de click puede tener como valor el elemento en el que se ha hecho click, mientras que un evento que ocurre cuando el usuario compra puede tener como valor el importe del carrito. (Su columna es FACT_VALUE)
  - Otros datos asociados al evento.
- **Tabla de Grupos de Informes** (o **Report Groups**). Contiene datos asociados al evento, de un modo similar al de la columna FACT_VALUE de la Tabla de Eventos. Cuando se debe capturar más de un dato de cada evento, el FACT_VALUE no es suficiente, y se añade una o más filas a esta tabla. Cada fila está asociada a la ocurrencia de un evento, y contiene la siguiente información:
  - El identificador del grupo de informe. (Su columna se llama GROUP_ID)
  - El identificador de la ocurrencia de un evento a la que pertenece la fila. (Su columna se llama HIT_KEY)

    - Un valor que indica de qué tipo es el grupo de informes que se presenta en esta fila. Puede haber varios grupos de informes para un solo evento, conteniendo cada uno valores con diferentes significados. Saber el tipo del grupo nos permite dar significado a los valores que contiene. (Su columna se llama GROUP_TYPE)
    - Cuatro columnas llamadas FACT_DIM_1, FACT_DIM_2, FACT_DIM_3 y FACT_DIM_4. Cada una de estas columnas es un hueco para introducir un dato relevante sobre el evento correspondiente al grupo.

Es evidente que estos datos, del modo en el que están presentados, son difícilmente aprovechables por un algoritmo de aprendizaje automático. Para hacerlos más usables, el objetivo de la aplicación a desarrollar es transformarlos en un solo conjunto de datos que presente una sola fila por sesión. Cada fila contendrá características interesantes sobre la sesión, como su fecha de inicio, el número de veces que ocurren ciertos eventos, el tiempo que el usuario permanece en una cierta página, etc.

5.2 - Descripción de los datos

En la introducción se ha descrito la estructura general de los datos. A continuación se detallan los eventos que pueden encontrarse en el conjunto de datos que se va a utilizar, así como su significado y grupos de informe correspondientes.

5.2.1 - Eventos

Los eventos que se han capturado y aparecen en los datos están descritos a continuación:

- **click**: ocurre cuando el usuario hace click en algún elemento de la página.
- **scroll**: ocurre cuando el usuario hace scroll en una página.
- **reviews**: ocurre cuando un usuario se detiene en las reseñas de un producto.
- **device**: ocurre al inicio de cada sesión, y en su FACT_VALUE se indica si el dispositivo que está usando el usuario es PC o MOBILE.
- **country**: ocurre al inicio de cada sesión, y su FACT_VALUE indica el país desde el que se conecta el usuario.
- **visit**: ocurre cuando el usuario entra en una página de la web. El FACT_VALUE indica cual es la página visitada, que puede ser HOME y SEARCH, entre otras.
- **amount**: ocurre cuando el usuario añade o elimina un producto de su carrito de la compra, y su FACT_VALUE indica el importe del carrito tras haber añadido o eliminado ese producto.
- **payment**: ocurre cuando el usuario paga por los productos en su carrito.

5.2.2 - Grupos de informe

En estos datos de ejemplo solo aparece un grupo de informe:

- **timestamp**: contiene el timestamp de ocurrencia del evento al que está asociado el grupo. En los datos de ejemplo, solo algunos eventos tienen este grupo de informe.

5.2.3 - Sesiones

La Tabla de Sesiones será tal y como se describió en la introducción de este capítulo, conteniendo, para cada sesión, su identificador, tiempo de inicio, duración y cantidad de peticiones realizadas por el usuario durante la sesión.

5.2.4 - Datos de ejemplo

Los datos de ejemplo utilizados para esta sección contienen información sobre cinco sesiones ficticias, siendo la correspondiente Tabla de Sesiones:

SESSION_KEY	SESSION_TIMESTAMP	SESSION_DURATION	HIT_COUNT
S001	1530546200	123	15
S002	1530534134	698	139
S003	1530546220	10	1
S004	1530543423	346	52
S005	1530559872	276	78

En las tablas de Eventos y Grupos de Informes los datos sobre las distintas sesiones están mezclados. Sin embargo, para poder tener un ejemplo de muy pequeño tamaño que muestre la forma de los datos, así como el resultado de las operaciones, a continuación se muestran los datos de eventos y grupos de informes correspondientes a una sola sesión, la S005.

Tabla de Eventos

HIT_KEY	SESSION_KEY	EVENT_ID	FACT_VALUE
H073	S005	click	
H074	S005	click	
H075	S005	click	
H076	S005	click	
H077	S005	click	
H078	S005	click	
H079	S005	click	
H080	S005	scroll	
H081	S005	scroll	
H082	S005	scroll	
H083	S005	reviews	
H084	S005	reviews	
H085	S005	device	MOBILE
H086	S005	visit	SEARCH

H087	S005	country	EN
H088	S005	amount	1520
H089	S005	amount	2560
H090	S005	payment	
H091	S005	click	
H092	S005	visit	SEARCH
H093	S005	visit	HOME
H094	S005	click	
H095	S005	click	
H096	S005	scroll	
H097	S005	scroll	
H098	S005	click	
H099	S005	amount	21
H100	S005	payment	
H101	S005	click	
H102	S005	click	
H103	S005	click	
H104	S005	visit	HOME

Report Groups

GROUP ID	HIT KEY	SESSION KEY	EVENT ID	GROUP TYPE	FACT DIM 1	FACT DIM 2	FACT DIM 3	FACT DIM 4
G064	H073	S005	click	TIMESTAMP	1530559873			
G065	H074	S005	click	TIMESTAMP	1530559874			

G066	H075	S005	click	TIMESTAMP	1530559875			
G067	H076	S005	click	TIMESTAMP	1530559876			
G068	H077	S005	click	TIMESTAMP	1530559877			
G069	H078	S005	click	TIMESTAMP	1530559878			
G070	H079	S005	click	TIMESTAMP	1530559879			
G071	H080	S005	scroll	TIMESTAMP	1530559880			
G072	H081	S005	scroll	TIMESTAMP	1530559881			
G073	H082	S005	scroll	TIMESTAMP	1530559882			
G074	H083	S005	reviews	TIMESTAMP	1530559883			
G075	H084	S005	reviews	TIMESTAMP	1530559884			
G076	H086	S005	visit	TIMESTAMP	1530559885			
G077	H088	S005	amount	TIMESTAMP	1530559886			
G078	H089	S005	amount	TIMESTAMP	1530559887			
G079	H090	S005	payment	TIMESTAMP	1530559888			
G080	H091	S005	click	TIMESTAMP	1530559889			
G081	H092	S005	visit	TIMESTAMP	1530559890			
G082	H093	S005	visit	TIMESTAMP	1530559891			
G083	H094	S005	click	TIMESTAMP	1530559892			
G084	H095	S005	click	TIMESTAMP	1530559893			
G085	H096	S005	scroll	TIMESTAMP	1530559894			
G086	H097	S005	scroll	TIMESTAMP	1530559895			
G087	H098	S005	click	TIMESTAMP	1530559896			

G088	H099	S005	amount	TIMESTAMP	1530559897			
G089	H100	S005	payment	TIMESTAMP	1530559898			
G090	H101	S005	click	TIMESTAMP	1530559899			
G091	H102	S005	click	TIMESTAMP	1530559900			
G092	H103	S005	click	TIMESTAMP	1530559901			
G093	H104	S005	visit	TIMESTAMP	1530559902			

5.3 - Objetivo del procesado

El objetivo del procesado a realizar es extraer una serie de características de cada sesión, de forma que puedan ser utilizadas mediante algoritmos de aprendizaje automático.

Las características son las siguientes:

- **ClicksCount**: el número de clicks que se ha realizado durante la sesión.
- **Interest**: es una medida del interés que muestra el usuario en la web. Su valor es la suma ponderada del número de veces que hace scroll y el número de veces que se para a ver los comentarios. El scroll tiene una ponderación de 0'3, y los comentarios de 0'7.
- **PC**: Su valor será '1' si el usuario interactuó con la web mediante un PC, y '0' en cualquier otro caso.
- **MOBILE**: Su valor será '1' si el usuario interactuó con la web mediante un teléfono móvil, y '0' en cualquier otro caso.
- **HomeVisits**: el número de veces que el usuario visita la página de inicio.
- **SearchVisits**: el número de veces que el usuario visita la página de búsqueda.
- **Country**: el país desde el cual se ha conectado el usuario.
- **CartValue**: el importe del carrito de la compra al finalizar la sesión.
- **Conversion**: Su valor será '1' si el usuario compró, y '0' en cualquier otro caso.

5.4 - Diseño

Durante este capítulo se detalla el diseño del procesado, así como de los componentes que sea necesario desarrollar para realizarlo.

5.4.1 - Agregación de los datos de entrada

El primer problema que aparece es la distribución de los datos de entrada en tres tablas diferentes, cuando la salida se requiere en una sola tabla que contenga todas las características extraídas.

Para facilitar toda la labor de procesado, la primera operación que se realizará sobre los datos es su agregación en una sola tabla. Esta agregación se realizará mediante una unión, de modo que la nueva tabla contendrá todos los datos de las tablas de entrada. También se añadirá una nueva columna, ROW_TYPE, que indicará a qué tabla pertenecía originalmente cada fila.

Esta agregación será encapsulada en una nueva clase Python, cuyo nombre será TealeafConnector.

Esta agregación se realizará sobre todos los datos de entrada y su resultado será almacenado. A partir de ahora, al hablar de datos de entrada me referiré al resultado de esta agregación.

5.4.2 - Operaciones

En esta sección se enumeran las operaciones que será necesario realizar sobre los datos, y con qué componente de la biblioteca se realizarán. En caso de que no haya un componente adecuado, se detalla un componente a desarrollar. En muchos casos se ha decidido implementar un nuevo componente aún siendo posible realizar la operación usando componentes ya implementados. Esta decisión se ha tomado para mejorar la eficiencia y legibilidad del código, ya que se considera que estas operaciones asociadas al contexto de Tealeaf van a ser utilizadas por la empresa en muchas aplicaciones diferentes.

Cuenta de eventos

Para conseguir la característica ClicksCount, así como Interest, es necesario contar el número de ocurrencias de ciertos eventos para cada sesión. Esta operación podría realizarse mediante el transformador RowCounterByCondition de la biblioteca, pero se realizará un nuevo transformador, llamado EventCounter, que se adaptará al esquema conocido de los datos para hacer la operación más eficiente. Usar el transformador EventCounter también hará el código más legible, ya que se adapta al contexto de la aplicación.

Comprobación de la ocurrencia de eventos

Para extraer la característica Conversión se debe comprobar si un evento ocurre alguna vez o no, siendo la salida de esta operación un 1 si el evento existe y un 0 en cualquier otro caso. Esto podría realizarse también usando transformadores ya implementados, pero para mejorar la eficiencia se implementará un nuevo transformador: EventChecker.

Comprobación de la aparición de ciertos FACT_VALUE

Para obtener las características PC y MOBILE se debe comprobar la aparición de los FACT_VALUE correspondientes en el evento Device. De nuevo, esta operación podría hacerse con los transformadores que provee la biblioteca, pero se implementará uno nuevo usando la información sobre el esquema de los datos: FactValueChecker.

Cuenta de las apariciones de ciertos FACT_VALUE

Para extraer las características HomeVisits y SearchVisits es necesario contar las veces que ocurre el FACT_VALUE asociado en el evento visit. De nuevo, se implementará un nuevo transformador: FactValueCounter.

Obtener un FACT_VALUE

Para extraer la característica Country es necesario obtener el FACT_VALUE del evento country. Para ello se implementará el transformador FactValueFinder.

Obtener el último FACT_VALUE, según un timestamp

Para obtener la característica CartValue es necesario obtener el FACT_VALUE del evento amount. Pero, dado que el evento amount ocurre varias veces durante la sesión, es necesario tomar el FACT_VALUE del último evento en ocurrir. Dado que el evento amount tiene un Grupo de Informe que contiene el timestamp del momento en el que ocurre el evento, podemos llevar a cabo esta operación. Para realizarla se implementará un nuevo transformador: LastFactValueFinder.

Suma ponderada de valores

Para obtener la característica Interest es necesario sumar ponderadamente el número de ocurrencias de dos eventos. Para realizar esta suma se utilizará el transformador de la biblioteca llamado WeightedSummator.

5.4.3 - Estructura del procesado

El PysparkStep que provee una fachada de uso sencillo para la creación y configuración de los transformadores. En este caso se desarrollará un Step que proveerá una interfaz sencilla para el uso de los nuevos transformadores. Se llamará TealeafStep, y será una subclase del PysparkStep.

Las operaciones descritas en el apartado anterior se deben realizar sobre el conjunto de datos de entrada, excepto la suma ponderada de las cuentas de los eventos scroll y reviews, que debe realizarse una vez que estos eventos han sido contados. Por eso, organizaremos el procesado en un Pipeline formado por dos Steps, del siguiente modo:

- Step 1. Contendrá:
  - Un EventCounter que contará las ocurrencias de los eventos click, scroll y reviews.
  - Un EventChecker que comprueba si ocurre el evento payment.
  - Un FactValueChecker que comprobará si el evento device contiene los valores PC o MOBILE.
  - Un FactValueCounter que contará el número de apariciones de los valores HOME y SEARCH en el evento visit.
  - Un FactValueFinder que obtendrá el valor del evento country.
  - Un LastFactValueFinder que obtendrá el valor de la última ocurrencia del evento amount.
- Step 2. Contendrá:
  - Un WeightedSummator, que sumará las cuentas de los eventos scroll y reviews realizadas en el step anterior.
  - Un ByPass, que llevará a la salida de este step las columnas que nos interesan de la salida del step anterior, es decir, todas las columnas excepto la cuenta de los scroll y reviews.

La Figura 18 muestra un esquema de este procesado.

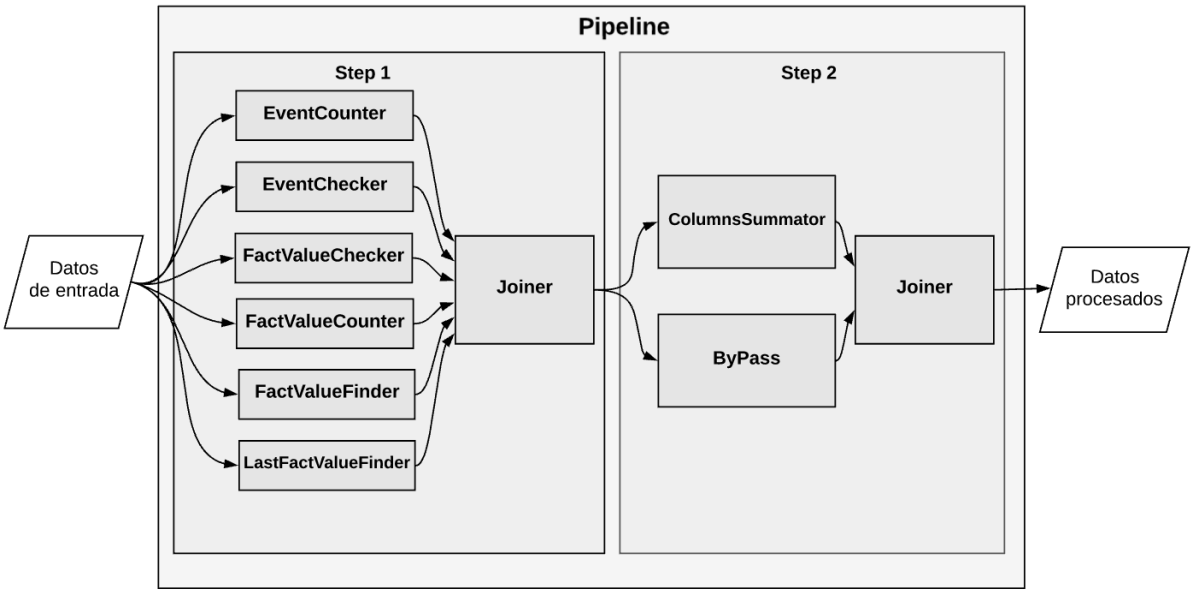


Figura 18: Esquema del procesado de la aplicación de la biblioteca.

5.4.4 - Implementación

Los transformadores se han implementado de una manera similar a los transformadores que provee la biblioteca. Han sido empaquetados en un nuevo módulo llamado TealeafTransformers, ya que son transformadores que dependen del esquema de los datos que provee Tealeaf.

Tras implementar estos transformadores, el procesado debe especificarse en un script, usando los componentes de la biblioteca. Después del trabajo realizado, el script es pequeño y legible, por lo que se incluye en este apartado como muestra de la consecución de los objetivos de este proyecto:

```
pipeline = Pipeline()
joiner = Joiner(join_columns=['SESSION_KEY'])
step1 = TealeafStep(data_merger=joiner, session_id_column='SESSION_KEY')
step2 = PysparkStep(data_merger=joiner, id_column='SESSION_KEY')

(step1
 .count_event(event_id='click', result_name='ClicksCount')
 .count_event(event_id='scroll', result_name='ScrollCount')
 .count_event(event_id='reviews', result_name='ReviewsCount')
 .check_event(event_id='payment', result_name='Conversion')
 .check_fact_value(event_id='device', value_to_check='PC', result_name='PC')
 .check_fact_value(event_id='device', value_to_check='MOBILE', result_name='MOBILE')
 .count_fact_value(event_id='visit', value_to_count='HOME', result_name='HomeVisits')
 .count_fact_value(event_id='visit', value_to_count='SEARCH', result_name='SearchVisits')
 .get_last_fact_value(event_id='amount', result_name='CartValue')
 .get_fact_value(event_id='country', result_name='Country')
 )

(step2
 .sum_weighted_columns(list_of_columns=['ScrollCount', 'ReviewsCount'],
       list_of_weights=[0.3, 0.7], result_name='Interest')
 .keep_input_columns(['SESSION_KEY', 'ClicksCount', 'PC', 'MOBILE', 'HomeVisits',
       'SearchVisits', 'CartValue', 'Country', 'Conversion'])
 )

pipeline.add_steps([step1, step2])

inputProvider = CsvInputProvider(
                  read_paths=['/source/data/path/data.csv'],
                  sql_ctx=sqlCtx
                )
outputProvider = CsvOutputProvider(
                 write_paths=['/result/data/path/result.csv'],
                  sql_ctx=sqlCtx
                )

orchestrator = Orchestrator(
         input_provider=inputProvider, transformer=pipeline, output_provider=outputProvider
 )

orchestrator.run()
```

El conjunto de datos resultante al aplicar esta transformación es el siguiente, tal y como se esperaba:

SESSION KEY	Interest	Clicks Count	PC	MOBILE	Home Visits	Search Visits	Cart Value	Country	Conversion
S001	8.5	3	1	0	1	2		ES	0

S002	2.8	6	1	0	3	0	230	ES	1
S003	0	1	1	0	1	0		EN	0
S004	1	9	0	1	1	4	42	USA	1
S005	2.9	14	0	1	2	2	21	EN	1

De este modo puede verse como todos los datos presentados sobre la sesión S005 han sido resumidos en una sola fila mucho más explicativa, con 9 características de las que se podría extraer valor mediante un algoritmo de aprendizaje automático o una simple descripción estadística.

5.5 - Cambio en los requisitos

Tras la realización del procesado del apartado anterior, se vio necesario modificarlo, por el siguiente motivo:

Al equipo de analistas de datos le pareció interesante la posibilidad de dividir una sesión en varias sub-sesiones, siguiendo diferentes criterios. Por ejemplo, a lo largo de algunas sesiones se realizan varias compras, es decir: se añaden objetos al carrito y se pagan, y posteriormente se añaden más objetos y se paga de nuevo.

Ante esta idea, se hace necesario extraer características no de las sesiones completas, sino de los tramos de las sesiones que ocurren entre cada compra.

La figura 19 muestra gráficamente cómo una sesión se divide en varias subsesiones a partir de un evento de corte.

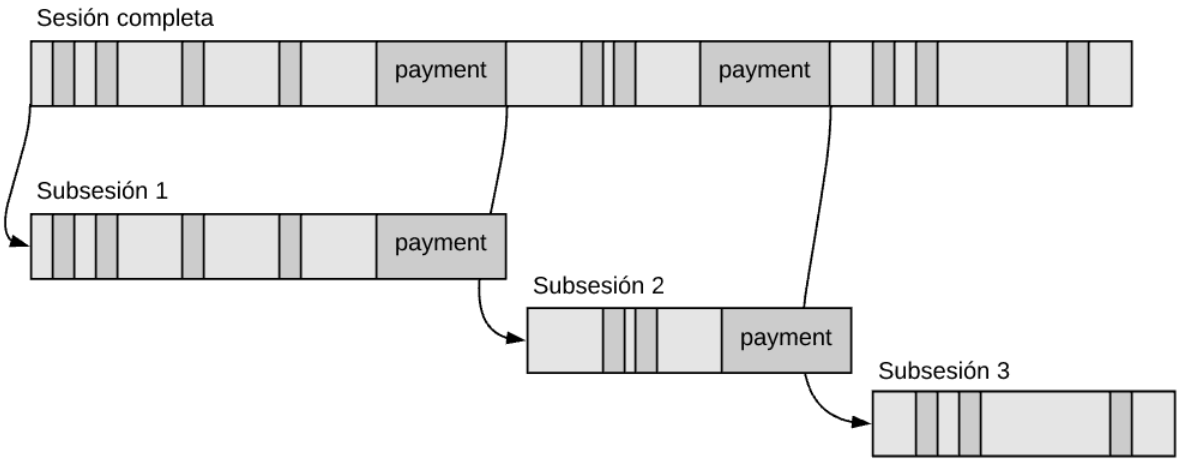


Figura 19: División de una sesión en subsesiones.

Para ello, se implementa un nuevo transformador para el módulo TealeafTransformers: SessionSlicer. Este transformador genera un nuevo sub-identificador para cada fila de los datos de entrada, dividiendo las sesiones según un evento dado, en este caso el evento payment.

Como ejemplo, los eventos de la sesión S005 de los datos de ejemplo, divididos según el evento *payment*, serían del siguiente modo:

HIT_KEY	SESSION_KEY	EVENT_ID	FACT_VALUE	SUBSESSION_ID
H073	S005	click		1
H074	S005	click		1
H075	S005	click		1
H076	S005	click		1
H077	S005	click		1
H078	S005	click		1
H079	S005	click		1
H080	S005	scroll		1
H081	S005	scroll		1
H082	S005	scroll		1
H083	S005	reviews		1
H084	S005	reviews		1
H085	S005	device	MOBILE	1
H085	S005	device	MOBILE	2
H085	S005	device	MOBILE	3
H086	S005	visit	SEARCH	1
H087	S005	country	EN	1
H087	S005	country	EN	2
H087	S005	country	EN	3
H088	S005	amount	1520	1

H089	S005	amount	2560	1
H090	S005	payment		1
H091	S005	click		2
H092	S005	visit	SEARCH	2
H093	S005	visit	HOME	2
H094	S005	click		2
H095	S005	click		2
H096	S005	scroll		2
H097	S005	scroll		2
H098	S005	click		2
H099	S005	amount	21	2
H100	S005	payment		2
H101	S005	click		3
H102	S005	click		3
H103	S005	click		3
H104	S005	visit	HOME	3

Para mantener la información sobre el orden que cada subsesión ocupa en la sesión, los subidentificadores siguen un orden ascendente.

Debe tenerse en cuenta que para asignar un id de subsesión no se usa el orden de los eventos en la tabla, sino que se usa el timestamp asociado a cada evento. A partir de los eventos elegidos para cortar la sesión, en este caso el evento payment, se obtienen las ventanas temporales correspondientes a cada subsesión, y con ellas se etiquetan el resto de eventos. Los eventos que no disponen de timestamp se consideran eventos globales a la sesión, como el país de origen y el dispositivo utilizado, y se replican para que aparezcan en todas las subsesiones de su sesión correspondiente.

Una vez que el SessionSlicer está implementado, el script de procesado se modifica añadiendo un nuevo Step al inicio del pipeline, que lo aplica a todos los datos de entrada. Además, los Steps posteriores se modifican para usar como ID de sesión tanto el SESSION_KEY como el nuevo sub-identificador.

A continuación se muestra el script resultante, que ya está listo para realizar el nuevo procesado.

```
step1 = TealeafStep(data_merger=joiner, session_id_column='SESSION_KEY')
step2 = TealeafStep(data_merger=sliced_joiner, session_id_column=['SESSION_KEY', 'SUB_ID'])
step3 = PysparkStep(data_merger=sliced_joiner, id_column=['SESSION_KEY', 'SUB_ID'])

step1.slice_sessions(slicer_event_id='payment', subsession_id_column_name='SUB_ID')

(step2
  .count_event(event_id='click', result_name='ClicksCount')
  .count_event(event_id='scroll', result_name='ScrollCount')
  .count_event(event_id='reviews', result_name='ReviewsCount')
  .check_event(event_id='payment', result_name='Conversion')
  .check_fact_value(event_id='device', value_to_check='PC', result_name='PC')
  .check_fact_value(event_id='device', value_to_check='MOBILE', result_name='MOBILE')
  .count_fact_value(event_id='visit', value_to_count='HOME', result_name='HomeVisits')
  .count_fact_value(event_id='visit', value_to_count='SEARCH', result_name='SearchVisits')
  .get_last_fact_value(event_id='amount', result_name='CartValue')
  .get_fact_value(event_id='country', result_name='Country')
)

(step3
  .sum_weighted_columns(list_of_columns=['ScrollCount', 'ReviewsCount'], list_of_weights=[0.3,
0.7],
      result_name='Interest')
  .keep_input_columns(['SESSION_KEY', 'ClicksCount', 'PC', 'MOBILE', 'HomeVisits', 'SearchVisits',
     'CartValue', 'Country', 'Conversion', 'SUB_ID'])
)

pipeline.add_steps([step1, step2, step3])

inputProvider = CsvInputProvider(
            read_paths=[‘/source/data/path/data.csv'],
            sql_ctx=sqlCtx
)

outputProvider = CsvOutputProvider(
            write_paths=[‘/result/data/path/result.csv'],
            sql_ctx=sqlCtx
)

orchestrator = Orchestrator(
            input_provider=inputProvider, transformer=pipeline, output_provider=outputProvider
)

orchestrator.run()
```

Los nuevos datos de salida usando este script son:

SESSION KEY	SUB ID	Interest	Clicks Count	PC	MOBILE	Home Visits	Search Visits	Cart Value	Country	Conversion
S001	1	8.5	3	1	0	1	2		ES	0
S002	1	2.8	6	1	0	3	0	230	ES	1
S003	1	0	1	1	0	1	0		EN	0
S004	1	1	4	0	1	1	4	10	USA	1
S004	2	0	5	0	1	0	0	42	USA	1
S005	1	2.3	7	0	1	0	1	2560	EN	1
S005	2	0.6	4	0	1	1	1	21	EN	1
S005	3	0	3	0	1	1	0		EN	0

Capítulo 6 - Pruebas

6.1 - Introducción

A lo largo de este capítulo se describirán las herramientas y el proceso seguido para la validación de la biblioteca desarrollada, así como de la aplicación descrita en el capítulo 5.

La validación de la biblioteca y su aplicación han seguido un esquema similar, y ha sido realizada mediante pruebas que pueden clasificarse según los siguientes tipos:

- **Pruebas unitarias**. Se utilizan validar el correcto funcionamiento de una única pieza del sistema, y por eso en la prueba se debe utilizar únicamente esa. Dado el hincapié que ha habido a lo largo del proceso de desarrollo de la biblioteca en el diseño modular, es sencillo realizar este tipo de pruebas para la mayoría de partes del sistema.
- **Pruebas de integración**. Se utilizan para validar que la interacción entre diferentes piezas del sistema es correcta.
- **Pruebas de aceptación**. Son las últimas pruebas en realizarse, durante las cuales el Product Owner prueba el sistema como usuario final y aporta retroalimentación sobre el mismo.
- **Revisiones de código**. Consisten en lecturas críticas del código, que intentan encontrar errores tanto en los algoritmos implementados como en el estilo de escritura del mismo.

A continuación se describe el modo en el que han sido llevadas a cabo las pruebas, y cuáles han sido estas.

6.2 - Plan de pruebas

Dada la naturaleza iterativa del proceso de desarrollo, se han realizado pruebas sobre cada nueva pieza del sistema implementada y sobre cada nueva versión funcional del sistema. Concretamente, las pruebas de aceptación han sido realizadas por el Product Owner frecuentemente, en cada revisión de sprint. Estas pruebas han tenido un doble propósito: por un lado, validar la versión del sistema realizada hasta el momento, y por otro, ayudar a refinar los requisitos y concretar las expectativas del Product Owner sobre el sistema.

Se realizaron revisiones de código sobre todo el sistema, varias veces en las partes más problemáticas. Debido al objetivo de conseguir código auto-documentado, estas revisiones son cruciales para asegurar la legibilidad del mismo. Lo ideal es que un código no sea revisado por su autor, pero en este caso excepcional en el que el equipo sólo dispone de un miembro, fue el mismo el autor y crítico del código. Sin embargo, la revisión de una pieza de código no se llevó a cabo hasta varios días después de su escritura. De este modo se intentó que el desarrollador hubiera olvidado en parte los motivos que le llevaron a escribir el código como lo hizo, y poder ser más crítico.

6.3 - Pruebas realizadas

En esta sección se exponen las pruebas unitarias y de integración realizadas sobre el sistema. Las pruebas de aceptación y las revisiones de código no están descritas, ya que no siguieron un proceso definido ni replicable que haga útil su documentación en el contexto de este proyecto.

Información adicional:
Debe notarse que el estado de todas las pruebas documentadas corresponde al de su aplicación sobre la última versión del sistema.

Las pruebas serán descritas siguiendo la siguiente plantilla:

Identificador	El identificador de la prueba, de la forma P-000
Tipo	Unitarias o de integración
Objeto	Parte o partes del sistema sobre las que se aplica esta prueba.
Descripción	Descripción de la prueba.
Salida esperada	Salida esperada de la prueba.
Estado	Estado actual de la prueba. Puede ser: No implementada, si no está implementada. Pendiente de aplicación, si no ha llegado a realizarse sobre la versión actual del sistema. No superada, si ha sido realizada pero el resultado no ha sido satisfactorio. Superada, si ha sido realizada y el resultado es satisfactorio.

Identificador	P-001
Tipo	Unitaria
Objeto	La clase Joiner, del paquete PySpark Operators.
Descripción	Se dará como entrada varios DataFrames, y se comprobará que la agregación que hace de ellos es correcta.
Salida esperada	Un DataFrame resultado del join de cada uno de los DataFrames de entrada sobre las columnas especificadas.
Estado	Superada
Notas	

Identificador	P-002
Tipo	Unitaria
Objeto	Pipeline, del paquete Data Pipelines.
Descripción	Se crea un Pipeline con varios transformadores triviales que realizan operaciones Python sobre enteros y se ejecuta sobre una cierta entrada.
Salida esperada	La salida obtenida al aplicar cada uno de los transformadores secuencialmente a la entrada.
Estado	Superadas
Notas	Aunque en esta prueba también se incluyen transformadores, he intentado tener esta prueba unitaria usando transformadores triviales.

Identificador	P-003
Tipo	Unitaria
Objeto	Step, del paquete Data Pipelines.
Descripción	Se crea un Pipeline con varios transformadores y un DataMerger triviales que realizan operaciones Python sobre enteros y se ejecuta sobre una cierta entrada.
Salida esperada	La salida obtenida al aplicar a la entrada cada uno de los transformadores, y agregar la salida de cada uno usando el DataMeger.
Estado	Superada
Notas	Aunque en esta prueba también se incluyen transformadores y un DataMerger, he intentado tener esta prueba unitaria usando transformadores triviales.

Identificador	P-004
Tipo	Unitaria
Objeto	Cada uno de los transformadores del paquete PySpark Operators y de la extensión de Tealeaf.
Descripción	Se comprobará que el procesado realizado por el transformador en cuestión es correcto.
Salida esperada	Un conjunto de datos que contenga el resultado correcto según la entrada y especificación del transformador.
Estado	Superada
Notas	Se ha decidido no detallar estas pruebas transformador por transformador, dada la gran similitud que habría entre las pruebas de cada uno.

Identificador	P-005
Tipo	Unitaria
Objeto	El CsvInputProviderByDate, del paquete PySpark Operators.
Descripción	Se comprobará que el InputProvider realiza correctamente la lectura de los ficheros en los directorios esperados.
Salida esperada	Los DataFrames resultado de la carga de los ficheros CSV en las rutas correspondientes a las fechas especificadas.
Estado	Superada
Notas	

Identificador	P-006
Tipo	Unitaria
Objeto	El CsvOutputProviderByDate del paquete PySpark Operators.
Descripción	Se comprobará que el OutputProvider realiza correctamente la escritura de los DataFrames de entrada en los directorios esperados.
Salida esperada	Ficheros CSV conteniendo los datos de los DataFrames de entrada, en las rutas correspondientes a las fechas especificadas.
Estado	Superada
Notas	

Identificador	P-007
Tipo	Unitaria
Objeto	La clase Orchestrator, del paquete Data Pipelines.
Descripción	Se comprobará que el Orchestrator regula correctamente la interacción entre InputProvider, Transformer y OutputProvider, probándolo usando versiones triviales de estos tres componentes.
Salida esperada	Los resultados de aplicar el Transformer a los DataFrames provistos por el InputProvider son recibidos por el OutputProvider.
Estado	Superada
Notas	A pesar de que esta prueba no afecta exclusivamente al Orchestrator, he intentado conseguir una prueba unitaria de este componente usando versiones triviales de los demás.

Identificador	P-008
Tipo	Integración
Objeto	El PysparkStep, del paquete PySpark Operators.
Descripción	Se probará el correcto funcionamiento de los métodos fachada para la configuración de los transformadores de la biblioteca provistos por el PysparkStep, mediante la configuración y ejecución de estos Steps con los distintos transformadores.
Salida esperada	Un DataFrame resultado de la aplicación de todos los transformadores configurados al DataFrame de entrada, y la posterior agregación de los resultados usando el DataMerger provisto.
Estado	Superada
Notas	

Identificador	P-009
Tipo	Integración
Objeto	El TealeafStep, de la extensión de Tealeaf.
Descripción	Se probará el correcto funcionamiento de los métodos fachada para la configuración de los transformadores de la biblioteca y de la extensión de Tealeaf provistos por el TealeafStep, mediante la configuración y ejecución de estos Steps con los distintos transformadores.
Salida esperada	Un DataFrame resultado de la aplicación de todos los transformadores configurados al DataFrame de entrada, y la posterior agregación de los resultados usando el DataMerger provisto.
Estado	Superada
Notas	

Identificador	P-010
Tipo	Unitaria
Objeto	El TealeafConnector, de la extensión de Tealeaf.
Descripción	Se comprobará que el TealeafConnector unifica correctamente DataFrames de sesiones, eventos y dimensiones de Tealeaf en un solo DataFrame.
Salida esperada	Un DataFrame resultado de la unificación de los DataFrames de sesiones, eventos y dimensiones.
Estado	Superada
Notas	

Identificador	P-011
Tipo	Integración
Objeto	La interacción entre Step, Pipeline, algunos transformadores, Orchestrator, CsvInputProviderByDate y CsvOutputProviderByDate
Descripción	Se probará el funcionamiento completo de la biblioteca mediante un caso de prueba que simula un caso real.
Estado	Superada
Notas	

Capítulo 7 - Conclusiones y trabajo futuro

7.1 - Conclusiones

Durante este trabajo de fin de grado se ha desarrollado una herramienta que otorga gran agilidad durante las primeras fases del análisis de datos, y con la capacidad para mejorar la forma de trabajar en este tipo de proyectos.

Precisamente, es en las primeras fases del análisis donde se dedican aproximadamente un 80% de los esfuerzos [1], por lo que la mejora en esta fase supone un impacto global para el proyecto.

La abstracción que provee la biblioteca desarrollada provee de agilidad y permite al analista de datos centrarse en su trabajo, el análisis, perdiendo menos tiempo y esfuerzo pensando en cómo obtener los datos que necesita. Al mismo tiempo, permite al ingeniero de datos implementar algoritmos de procesamiento como herramientas de uso sencillo que el analista puede utilizar allí donde necesite. De este modo se facilita la cooperación dentro del equipo y se evita que el mismo código tenga que escribirse dos veces.

La flexibilidad de esta abstracción se ha puesto a prueba cuando, durante el desarrollo de la biblioteca y de una aplicación de la misma, se han producido cambios en los requisitos. Estos casos se han solucionado desarrollando nuevos componentes, sin modificar el núcleo de la biblioteca. Esta flexibilidad y facilidad de extensión permitirá a la herramienta crecer para satisfacer necesidades futuras.

La fuente del valor que puede aportar la biblioteca radica en el hecho de que en ocasiones merece la pena cesar el trabajo, analizarlo y desarrollar una herramienta que lo facilita, ya que esta inversión en esfuerzo puede retornar en el ahorro de mucho más durante el trabajo futuro.

La herramienta desarrollada ha surgido a partir una necesidad de la empresa cliente para un proyecto real, y actualmente se planea la refactorización de procesos actuales para su utilización, donde se pondrá a prueba y aportará valor en un entorno de uso real.

7.2 - Líneas futuras de trabajo

Por el carácter modular y extensible de la biblioteca desarrollada, hay mucha funcionalidad que puede añadirse a la misma. Igualmente, algunas funciones importantes han quedado fuera del alcance de este proyecto. A continuación se enumeran algunas de las posibles líneas de trabajo futuras referentes a este proyecto:

- **Mejorar el control de errores** y emisión de mensajes (logs), para conseguir dos objetivos:
    - Por un lado mejorar, la experiencia del usuario, ya que ante errores por su parte en el uso de la biblioteca, el sistema debería lanzar un mensaje de error descriptivo que le indique cuál es el fallo y cómo debe solucionarlo, sin mostrar pormenores sobre la implementación de la biblioteca o Spark.
    - Por otro, facilitar el procesado automatizado por lotes, permitiendo que el sistema se recupere mejor ante un error y continúe con el procesado, notificando del fallo mediante mensajes descriptivos.
- **Ampliar el abanico de transformadores**, DataMergers, InputProviders y OutputProviders disponibles, en función de las necesidades de los usuarios.
- **Integrar el uso de esquemas de las tablas** para la lectura y escritura de datos. Actualmente, la biblioteca infiere el tipo de datos que aparecen en las tablas de datos que lee. En un entorno de producción, para mejorar la robustez y eficiencia del sistema, este debería conocer de antemano los tipos de datos que aparecen en las tablas. Por los mismos motivos, el sistema debería generar y persistir una descripción de los tipos de datos presentes en los resultados de los procesados que realiza, facilitando así su aprovechamiento en el futuro.

- **Generar una batería de pruebas** unitarias y de integración que puedan ejecutarse automáticamente cuando sea necesario, de forma que pueda validarse rápidamente todo el sistema ante cambios en el mismo o en la interfaz de programación de Spark.
- **Revisión del código por parte de un tercero**. Ya que la biblioteca ha sido diseñada, implementada y revisada por un solo desarrollador, es posible que su forma particular de trabajar y de entender el sistema y el código haya perjudicado a la calidad del mismo, aún con la ayuda del Product Owner para asegurar que el sistema se ciñera a las necesidades de los usuarios. Por esto, sería interesante la revisión del código por parte de un desarrollador que no haya estado implicado en el proyecto.
- **Optimización de los algoritmos de procesado** implementados usando para ello las herramientas que proporciona Spark, como la visualización del DAG de procesamiento.

Bibliografía

[1] Lohr, S. (2014). *For Big-Data Scientists, 'Janitor Work' Is Key Hurdle to Insights*. [online] Nytimes.com. Available at: https://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html?_r=0 [Accessed 13 Jul. 2018].

[2] Piatetsky, G. (2014). *CRISP-DM, still the top methodology for analytics, data mining, or data science projects*. [online] Kdnuggets.com. Available at: https://www.kdnuggets.com/2014/10/crisp-dm-top-methodology-analytics-data-mining-data-science-projects.html [Accessed 13 Jul. 2018].

[3] Jensen, K. (2012). *CRISP-DM Process Diagram*. [image] Available at: https://es.wikipedia.org/wiki/Cross_Industry_Standard_Process_for_Data_Mining#/media/File:CRISP-DM_Process_Diagram.png [Accessed 13 Jul. 2018].

[4] Peters, T. (2004). *PEP 20 -- The Zen of Python*. [online] Python.org. Available at: https://www.python.org/dev/peps/pep-0020/ [Accessed 13 Jul. 2018].

[5] Mick, T. (2002). *PEP 282 -- A Logging System*. [online] Python.org. Available at: https://www.python.org/dev/peps/pep-0282/ [Accessed 13 Jul. 2018].

[6] Martin, R. (n.d.). *The Liskov Substitution Principle*. [ebook] Available at: https://drive.google.com/file/d/0BwhCYaYDn8EgNzAzZjA5ZmItNjU3NS00MzQ5LTkwYjMtMDJhNDU5ZTM0MTlh/view [Accessed 13 Jul. 2018].

[7] Jakob Frank, E. (2018). *Best Practice Software Engineering - Interface*. [online] Best-practice-software-engineering.ifs.tuwien.ac.at. Available at: http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/interface.html [Accessed 13 Jul. 2018].

[8] Sourcemaking.com. (2018). *Strategy Design Pattern*. [online] Available at: https://sourcemaking.com/design_patterns/strategy [Accessed 13 Jul. 2018].

[9] Sourcemaking.com. (2018). *Mediator Design Pattern*. [online] Available at: https://sourcemaking.com/design_patterns/mediator [Accessed 13 Jul. 2018].

[10] Sourcemaking.com. (2018). *Composite Design Pattern*. [online] Available at: https://sourcemaking.com/design_patterns/composite [Accessed 13 Jul. 2018].

[11] Martin, R. (2003). *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, pp.127–131.

[12] Msdn.microsoft.com. (2018). *The Dependency Injection Design Pattern*. [online] Available at: https://msdn.microsoft.com/en-us/library/hh323705(v=vs.100).aspx [Accessed 13 Jul. 2018].

[13] Sourcemaking.com. (2018). *Builder Design Pattern*. [online] Available at: https://sourcemaking.com/design_patterns/builder [Accessed 13 Jul. 2018].

[14] Sourcemaking.com. (2018). *Facade Design Pattern*. [online] Available at: https://sourcemaking.com/design_patterns/*Facade* [Accessed 13 Jul. 2018].

[15] En.wikipedia.org. (2018). *Feature hashing*. [online] Available at: https://en.wikipedia.org/wiki/Feature_hashing [Accessed 13 Jul. 2018].

[16] Spark.apache.org. (2018). *pyspark.ml package — PySpark 2.2.0 documentation*. [online] Available at: http://spark.apache.org/docs/2.2.0/api/python/pyspark.ml.html#pyspark.ml.feature.HashingTF [Accessed 13 Jul. 2018].

[17] Harris, D. and Harris, S. (2012). *Digital design and computer architecture*. 2nd ed. San Francisco, Calif.: Morgan Kaufmann, p.129.

[18] boo.codehaus.org. (2008). *BOO - Duck Typing*. [online] Available at: https://web.archive.org/web/20081006075246/http://boo.codehaus.org/Duck+Typing [Accessed 13 Jul. 2018].

[19] van Rossum, G. (2001). *PEP 8 -- Style Guide for Python Code*. [online] Python.org. Available at: https://www.python.org/dev/peps/pep-0008/? [Accessed 13 Jul. 2018].

[20] Docs.python.org. (2018). *Property Built-in Function — Python 3.7.0 documentation*. [online] Available at: https://docs.python.org/3/library/functions.html#property [Accessed 13 Jul. 2018].

[21] En.wikipedia.org. (2018). *Feature engineering*. [online] Available at: https://en.wikipedia.org/wiki/Feature_engineering [Accessed 13 Jul. 2018].

ANEXOS

ANEXO A:
DOCUMENTACIÓN
DE LA BIBLIOTECA

Pipelines Library Documentation

Release 1

Óliver L. Sanz

July 04, 2018

CONTENTS

CHAPTER

ONE

PIPELINE_COMPONENTS MODULE

class pipeline_components.**DataMerger**
Bases: object

Abstract class for a DataMerger.

DataMergers take multiple datasets, run some processing on them, and then return a single dataset as result. Can be thought of as a way of merging multiple sets of data into a single one.

merge(*datasets*)
Merges multiple datasets into a single one, according to the implementation of the DataMerger.

Parameters **datasets** (*list of datasets*) – A list of the datasets to be merged.

Returns The merged dataset

class pipeline_components.**InputProvider**
Bases: object

Abtract class for an InputProvider.

An InputProvider supplies datasets that it reads, receives or generates following its own logic.

get_dataset()
Obtains a dataset and returns it. Each subsequent call to get_dataset should return a different dataset.

Returns The obtained dataset.

Raises pyspark_pipelines.utils.DataIOException – If the dataset could not be obtained. The state of the InputProvider must remain the same as before the call to this method.

skip_read()
Skips the next dataset to be obtained.

Returns None

class pipeline_components.**Orchestrator**(*transformer*, *input_provider*, *output_provider*)
Bases: object

Manages the reading of datasets, its processing and the output of results.

A Orchestrator queries a InputProvider for datasets, transforms them using a Transformer, and then outputs the result using a OutputProvider. This sequence occurs in a loop until there is no more input to read.

Parameters

- **input_provider** – The provider of the input datasets.
- **transformer** – The Transformer to be used to transform the input datasets.
- **output_provider** – The OutputProvider that will output the results of the transformations.

run()
Queries the InputProvider for datasets, transforms them using the Transformer, and then outputs the result using the OutputProvider. This sequence occurs in a loop until there is no more input to read.

Returns None

Notes

This method swallows all exceptions raised during the dataset transformations, and logs them using the python's logging module. This is done in order to prevent a malformed dataset or other similar circumstance to interrupt the processing, but may prevent a bug to stop the execution. Be sure to check the logs during or at the end of the execution of the run method.

class pipeline_components.**OutputProvider**
Bases: object

Abtract class for an OutputProvider.

An OutputProvider writes, sends, or do whatever it is implemented to do with the datasets provided to it.

put_dataset(*dataset*)
Receives a dataset as input and writes, sends, or outputs that dataframe in any way. This call might alter the state of the object, e.g. if it has a limited amount of slots to store its output.

Parameters **dataset** – The dataset to be outputted

Returns True if the output was successful, false if the OutputProvider has exhausted its potential to output datasets.

skip_write()
May alter the state of the OutputProvider, e.g. to skip a slot to write its output.

Returns None

class pipeline_components.**Pipeline**(*transformers=None*)
Bases: *pipeline_components.Transformer*

Chains multiple transformation together.

Pipelines execute multiple transformations in a sequential manner, using the output of a transformer as the input for the next one.

Parameters **transformers** – The list of transformers that will be used by the pipeline, in the order that the transformations have to be applied. Default is an empty list.

add_step(*step*)
Adds a step (or Transformer) to the end of the pipeline.

Parameters **step** – A Transformer to be added at the end of the transformer list.

Returns This pipeline object, so that method chaining can be used.

add_steps(*step_list*)
Adds a list of steps (or Transformers) to the end of the pipeline.

Parameters **step_list** – A list of Transformer to be added at the end of the transformer list.

Returns This pipeline object, so that method chaining can be used.

transform(*dataset*)
Transforms de dataset using the Transformers that have been configured to the Pipeline. This occurs in a sequential manner, using the output of a transformer as the input for the next one.

Parameters **dataset** – The dataset to be transformed

Returns The transformed dataset

class pipeline_components.**Step**(*transformers=None*, *data_merger=None*)
Bases: *pipeline_components.Transformer*

Performs parallel transformations on a dataset and merges the results.

This transformer transforms a single dataset using each one of the transformers provided to it, and then merges the results of each into a single dataset, through the use of a DataMerger.

Parameters

- **transformers** – The list of Transformers to be used by the step. Default is an empty list.
- **data_merger** – The DataMerger to be used to merge the results of each Transformer into a single dataset.

add_transformer(*transformer*)
Adds a transformer to the Step

Parameters **transformer** – The Transformer to be added to the Step

Returns This Step, so that method chaining can be used

transform(*dataset*)
Transforms the input dataset using its Transformers and DataMerger.

Parameters **dataset** – The dataset to be transformed

Returns The transformed dataset

class pipeline_components.**Transformer**
Bases: object

Abtract class for a transformer.

Transformers act as the base unit for data processing, being which transforms a dataset into another.

transform(*dataset*)
Trasforms a dataset according to the implementation of the transformer.

Parameters **dataset** – The data to be transformed

Returns the transformed data

CHAPTER

TWO

DATA_MERGERS MODULE

class data_mergers.**Joiner**(*join_columns*)

Bases: pipelib.pipeline_components.DataMerger

DataMerger that performs an equi left-outer join to merge multiple dataframes into a single one.

Parameters **join_columns** (*str or list of str*) – The columns to be used to perform the equi-join. Must be present in each of the DataFrames to be merged. If there is only one column, it may be specified as a string. Otherwise, a list of strings must me used.

Note: If columns other than the join columns share the same name in multiple DataFrames, you may face ambiguous reference problems. Also, if the same set of values for the join_columns occur more than once in a DataFrame, the result DataFrame will end up with duplicated rows.

---

merge(*datasets*)

Perform joins over the input DataFrames to merge them into a single one.

Parameters **datasets** (*pyspark.DataFrame*) – DataFrames to be merged

Returns The DataFrame result of the join

CHAPTER

THREE

IO_PROVIDERS MODULE

class io_providers.**CsvInputProvider**(*read_paths*, *sql_ctx*)
Bases: pipelib.pipeline_components.InputProvider

Reads PySpark DataFrames from CSV files found in a providen list of paths.

Parameters

- **read_paths** – A list of strings containing the paths of the CSV files are going to be read.
- **sql_ctx** (*pyspark.SQLContext*) – The PySpark SQLContext to be used.

get_dataset()
Reads a PySpark DataFrame from the next path in the list.

Returns The DataFrame, or None if there is no more files to read.

Raises pyspark_pipelines.utils.DataIOException – When the file can't be readen. When this occurs, the state of the object remains unchanged. You may want to skip the current date using the skip_read method.

skip_read()
Skips the current read path.

Returns None

class io_providers.**CsvInputProviderByDate**(*read_path*, *start_date*, *end_date*, *sql_ctx*)
Bases: pipelib.pipeline_components.InputProvider

Reads PySpark DataFrames from csv files according to a date sequence.

Seeks csv files according to the subpath structure 'year/yearmonth/yearmonthday', given a range of dates, and loads them as PySpark DataFrames.

Parameters

- **read_path** – The path from where the structured subpaths can be found.
- **start_date** (*datetime.date*) – The first date to load.
- **end_date** (*datetime.date*) – The last date to load.
- **sql_ctx** (*pyspark.SQLContext*) – The PySpark SQLContext to be used.

get_dataset()
Reads the DataFrame corresponding to the next date in the date range, then advances the current date, so subsequent calls to this method will return different DataFrames.

Returns The DataFrame corresponding to the current date, or None if the end of the date range has been reached.

Raises pyspark_pipelines.utils.DataIOException – When the file can't be readen. When this occurs, the state of the object remains unchanged. You may want to skip the current date using the skip_read method.

reset()
Returns the object to its initial state, so that it will start again reading from the start date.

Returns None

skip_read()
Skips to the next date to read.

Returns None

class io_providers.**CsvOutputProvider**(*write_paths*, *sql_ctx*)
Bases: pipelib.pipeline_components.OutputProvider

Writes PySpark DataFrames as CSV files to a providen list of paths.

Parameters **write_paths** – A list of strings containing the paths where the CSV files are going to be written.

put_dataset(*dataset*)
Writes a Dataframe to a CSV file

Parameters **dataset** – The PySpark DataFrame to be written

Returns True if the output was successful, False if the list of paths was exhausted.

skip_write()
Skips the current write path

Returns None

class io_providers.**CsvOutputProviderByDate**(*write_path*, *start_date*, *end_date*, *sql_ctx*)
Bases: pipelib.pipeline_components.OutputProvider

Writes PySpark DataFrames to csv files according to a date sequence.

Creates (or replace) csv files according to the subpath structure 'year/yearmonth/yearmonthday', given a range of dates, writing the content of PySpark DataFrames.

Parameters

- **write_path** – The path from where the structured subpaths should be created.
- **start_date** (*datetime.date*) – The first date to write.
- **end_date** (*datetime.date*) – The last date to write.
- **sql_ctx** (*pyspark.SQLContext*) – The PySpark SQLContext to be used.

put_dataset(*dataset*)
Writes the DataFrame in the subpath corresponding to the current date, then advances it, so that subsequent calls to this method will write the dataframes in different paths, each of them corresponding to a different date.

Parameters **dataset** (*pyspark.DataFrame*) – The DataFrame to be written

Returns True if the DataFrame was written, False if the end of the date range was reached and therefore the DataFrame was not written.

reset()
Returns the object to its initial state, so that it will start again writing from the start date.

Returns None

skip_write()
Skips to the next date to write.

Returns None

CHAPTER

FOUR

TRANSFORMERS MODULE

class transformers.**Binarizer**(*id_columns*, *column_list=None*)
Bases: pipelib.pipeline_components.Transformer

Binarizes values, leaving them at 0 if they are empty, otherwise setting them to 1.

Parameters

- **id_columns** – The columns containing the id values for each row.
- **column_list** – A list of named tuples containing all the info about the columns to be binarized. Default is an empty list.

add_column_to_binarize(*column*, *result_name*)
Adds a column to be binarized by the transform method.

Parameters

- **column** – The name of the column to be binarized.
- **result_name** – The name of the result column.

Returns This Binarizer, so that method chaining can be used.

classmethod get_param_type()
Returns the type of the named tuples containing the info about the columns to binarize. Only relevant if you are not going to use the helper method.

transform(*dataframe*)
Binarize the input DataFrame.

Parameters **dataframe** – The DataFrame containing the columns to be binarized.

Returns A new DataFrame containing the id columns as well as the binarized columns from the input DataFrame.

class transformers.**ByPass**(*columns=None*)
Bases: pipelib.pipeline_components.Transformer

Gets some columns from the input dataset, without any changes.

Parameters **columns** (*List of str*) – The names of the columns to bypass. If none is specified, all columns will be bypassed.

transform(*dataset*)
Bypass some columns of a dataset.

Parameters **dataset** – The DataFrame to bypass.

Returns A DataFrame containing the bypassed columns of the input dataframe. If no columns were specified, the returned DataFrame will be the input DataFrame.

class transformers.**ColumnDropper**(*cols_to_drop*)
Bases: pipelib.pipeline_components.Transformer

Drops columns from a DataFrame

Parameters **cols_to_drop** – A list of the names of the columns to be dropped.

transform(*dataframe*)
Drops the given columns from a input DataFrame.

Parameters **dataframe** – The input DataFrame.

Returns The DataFrame with the columns dropped.

class transformers.**ColumnsSummator**(*id_columns*, *summations_list=None*)
Bases: pipelib.pipeline_components.Transformer

Sums two or more numeric columns.

Parameters

- **id_columns** – The columns containing the id for each row.
- **summations_list** – A list of named tuples containing the params of each summation to be done. Default is an empty list.

add_sum(*list_of_columns*, *result_name*)
Adds a sum to be performed by the transform method.

Parameters

- **list_of_columns** (*list of str*) – The name sof the columns to be summed.
- **result_name** – The name of the column that will hold the results of the sum.

Returns This ColumnsSummator, so that method chaining can be used.

classmethod **get_param_type**()
Returns the type for the named tuple modelling summations. Only relevant if you are going to add summations without the use of the helper methods.

transform(*dataset*)
Performs all the sums over a dataset.

Parameters **dataset** – PySpark Dataframe containing the columns to be summed.

Returns A PySpark DataFrame containing the id columns and the result columns.

class transformers.**ColumnsSummatorByPrefix**(*id_columns*, *summation_list=None*)
Bases: pipelib.pipeline_components.Transformer

Sums all columns in a DataFrame which names share a specified prefix.

Parameters

- **id_columns** – The columns containing the id for each row.
- **summation_list** – A list of named tuples containing the params of each sum to be done. Default is an empty list.

add_summation(*prefix*, *result_name*)
Adds a summation to be performed by the transform method.

Parameters

- **prefix** – The prefix shared among the names of the columns to be summed.
- **result_name** – Tha name of the result column.

Returns This ColumnsSummatorByPrefix, so that method chaining can be used.

classmethod **get_param_type**()
Returns the type for the named tuple modelling sums. Only relevant if you are going to add sums without the use of the helper methods.

transform(*dataframe*)
Performs the summations over a DataFrame.

Parameters **dataframe** – The DataFrame containing the columns to sum.

Returns A new DataFrame containing the id columns of the input DataFrame, as well as the results of each summation.

class transformers.**ColumnsWeightedSummator**(*id_columns*, *summations_list=None*)
Bases: pipelib.pipeline_components.Transformer

Sums two or more numeric columns applying a weight to each one.

Parameters

- **id_columns** – The columns containing the id for each row.
- **summations_list** – A list of named tuples containing the params of each summation to be done. Default is an empty list.

add_weighted_sum(*list_of_columns*, *list_of_weights*, *result_name*)
Adds a weighted sum to be performed by the transform method.

Parameters

- **list_of_columns** (*list of str*) – The names of the columns to be summed.
- **list_of_weights** (*list of float*) – The weights corresponding to each of the columns to be summed.
- **result_name** – The name of the column that will hold the results of the sum.

Returns This ColumnsWeightedSummator, so that method chaining can be used.

classmethod **get_param_type**()
Returns the type for the named tuple modelling summations. Only relevant if you are going to add summations without the use of the helper methods.

transform(*dataset*)
Performs all the weighted sums over a dataset.

Parameters **dataset** – PySpark Dataframe containing the columns to be summed.

Returns A PySpark DataFrame containing the id columns and the result columns.

class transformers.**FeatureHasher**(*id_columns*, *columns_to_hash_list=None*)
Bases: pipelib.pipeline_components.Transformer

Performs feature hashing (also known as the hashing trick).

Parameters

- **id_columns** – The columns containing the id for each row.
- **columns_to_hash_list** – A list of named tuples containing the params of each hashing to be done. Default is an empty list.

add_hashing(*column_to_hash*, *number_of_features*, *result_columns_prefix*)
Adds a hashing to be performed by the transform method.

Parameters

- **column_to_hash** – The column to hash.
- **number_of_features** – The number of columns to generate. Should be a power of two.
- **result_columns_prefix** – A common prefix for the names of the columns to be generated by the feature hashing.

Returns This FeatureHasher, so that method chaining can be used.

classmethod **get_param_type**()
Returns the type for the named tuple with the info about one hashing. Only relevant if you are going to add them without the use of the helper methods.

transform(*dataframe*)
Performs the feature hashing over a DataFrame.

Parameters **dataframe** – The DataFrame containing the feature columns to hash.

Returns A new DataFrame containing the id columns of the input DataFrame, as well as the results of each feature hashing.

class transformers.**Function**(*id_columns*, *functions_list=None*)
Bases: pipelib.pipeline_components.Transformer

Applies a function to all the values in a column

classmethod **get_param_type**()
Returns the type of the named tuples containing the info about a function to apply. Only relevant if you are not going to use the helper method.

class transformers.**Identifier**(*id_columns_name*)
Bases: pipelib.pipeline_components.Transformer

Adds a unique id column to a DataFrame.

Parameters **id_columns_name** – The name of the new id column

transform(*dataframe*)
Generates a new unique id column for an input DataFrame

Parameters **dataframe** – The input DataFrame

Returns A DataFrame like the input DataFrame, with a the new unique id column.

class transformers.**JSONFlattener**(*field_list=None*)
Bases: pipelib.pipeline_components.Transformer

Gets values from a JSON DataFrame.

Parameters **field_list** – A list of named tuples with the info of each JSON field to be getted. Default is an empty list.

add_field(*field_path*, *result_name*)
Adds a JSON field to be getted.

Parameters

- **field_path** (*str*) – The path of the field within the JSON DataFrame.
- **result_name** – The name of the result column.

Returns This JSONFlattener, so that method chaining can be used.

classmethod **get_param_type**()
Returns the type for the named tuple containing the info for each JSON field. Only relevant if you are going to add fields to get without the use of the helper methods.

transform(*dataframe*)
Gets the JSON fields from a DataFrame

Parameters **dataframe** – The JSON Dataframe from where to get the fields.

Returns A DataFrame containing just one column for each field gotten.

class transformers.**ListExploder**(*id_columns*, *columns_to_explode_list=None*)
Bases: pipelib.pipeline_components.Transformer

Explodes columns containing lists.

Generates a new row for each value in each list. The corresponding value for each row will appear in a new column.

Parameters

- **id_columns** – The columns containing the id for each row.
- **columns_to_explode_list** – A list of named tuples containing the params of each exploding to be done. Default is an empty list.

add_exploding(*column_to_explode*, *result_name*)
Adds a exploding to be performed by the transform method.

Parameters

- **column_to_explode** – The row containing the lists to be exploded.
- **result_name** – The name of the column that will contain the exploded values.

Returns This ListExploder, so that method chaining can be used.

classmethod **get_param_type**()
Returns the type for the named tuple modelling explodings. Only relevant if you are going to add them without the use of the helper methods.

transform(*dataframe*)
Performs the explodings over a DataFrame.

Note: After the explosion of a list, it's likely that there is repeated id values across multiple rows.

Parameters **dataframe** – The DataFrame containing the lists to explode.

Returns A new DataFrame containing the id columns of the input DataFrame, as well as the results of each exploding.

class transformers.**NaFiller**(*fill_params*)
Bases: pipelib.pipeline_components.Transformer

Substitutes null values with a given value.

fill_params: { 'nombre de columna': valor por el que substituir los nulos

}

Parameters **fill_params** – A dictionary containing a key-value pair for each column which null values has to be filled. Each key is a column name, and its value the corresponding fill value.

transform(*dataframe*)
Fills the nulls of a DataFrame

Parameters **dataframe** – The DataFrame to fill.

Returns The filled DataFrame.

class transformers.**OneHotEncoder**(*id_columns*, *encoding_list=None*)
Bases: pipelib.pipeline_components.Transformer

Performs One Hot Encoding.

Parameters

- **id_columns** – The columns containing the id values for each row of the input DataFrames.
- **encoding_list** – A list of named tuples containing the info for each encoding to be done. Default is an empty list.

add_enconding(*column_to_encode*, *sql_condition*, *result_prefix*)
Adds a encoding to be performed by the transform method.

Parameters

- **column_to_encode** – The name of the column to be encoded.
- **sql_condition** – An sql condition used to filter rows before the enconding.
- **result_prefix** – The prefix in the name of the columns generated by the encoding.

Returns This OneHotEncoder, so that method chaining can be used.

classmethod get_param_type()
Returns the type for the named containing the info about a one hot encoding. Only relevant if you are going to add encodings without the use of the helper methods.

transform(*dataframe*)
Performs all the encodings over a DataFrame.

Parameters **dataframe** – The dataframe containing the rows to be encoded.

Returns A new DataFrame containing the id columns as well as the columns resulting of the encoding.

class transformers.**RowCounterByCondition**(*id_columns*, *counts_list=None*)
Bases: pipelib.pipeline_components.Transformer

Counts, for each id, the rows that fulfills a given SQL condition.

Parameters

- **id_columns** – The columns containing the id values for each row of the input DataFrames.
- **counts_list** – A list of named tuples containing the info for each count to be done. Default is an empty list.

add_count(*sql_condition*, *result_name*)
Adds a count to be performed by the transform method.

Parameters

- **sql_condition** – The SQL condition that has to be fulfilled by the rows to count. Has to be expressed as a pyspark column condition.
- **result_name** – The name of the result column.

Returns This RowCounterByCondition, so that method chaining can be used.

classmethod get_param_type()
Returns the type for the named tuple modelling counts. Only relevant if you are going to add counts without the use of the helper methods.

transform(*dataframe*)
Performs all the count over a DataFrame.

Parameters **dataframe** – The dataframe containing the rows to be counted.

Returns A new DataFrame containing the id columns as well as the result columns.

class transformers.**StringNormalizer**(*id_columns*, *columns_to_normalize=None*)
Bases: pipelib.pipeline_components.Transformer

Change to lowercase and strips a string from whitespaces.

Parameters

- **id_columns** – The columns containing the id values for each row.
- **columns_to_normalize** – The names of the columns to be normalized.

add_normalization(*column*)
Adds a column to be string normalized by the transform method.

Parameters **column** – The name of the column to be normalized

Returns This StringNormalizer, so that method chaining can be used.

transform(*dataframe*)
Normalizes the given text fields of a DataFrame

Parameters **dataframe** – The Dataframe from where to normalize the text fields.

Returns A DataFrame containing the same columns of the input DataFrame, but with the given columns with its text normalized.

class transformers.**TextSplitter**(*id_columns*, *columns_to_split_list=None*)
Bases: pipelib.pipeline_components.Transformer

Splits text into a list of words.

Parameters

- **id_columns** – The columns containing the id values for each row of the input DataFrames.
- **columns_to_split_list** – A list of named tuples containing the info for each split to be done. Default is an empty list.

add_splitting(*source_column*, *delimiter*, *sql_condition*, *result_name*)

Parameters

- **source_column** – The column containing the text to be splitted.
- **delimiter** – The delimiter to be used for the text split.
- **sql_condition** – A optional sql condition to filter the input DataFrame before the split.
- **result_name** – The name of the result column.

Returns This TextSplitter, so that method chaining can be used.

classmethod **get_param_type**()
Returns the type for the named tuple with the info for one text splitting. Only relevant if you are going to add splits without the use of the helper methods.

transform(*dataframe*)
Performs the splits over a DataFrame.

Parameters **dataframe** – The DataFrame containing the columns to split.

Returns A new DataFrame containing the id columns of the input DataFrame, as well as the results of each split.

class transformers.**ValueCollector**(*id_columns*, *collecting_list=None*)
Bases: pipelib.pipeline_components.Transformer

Collects a list of values for each id, in rows that fulfill a given SQL condition.

Parameters

- **id_columns** – The columns containing the id for each row.
- **collecting_list** – A list of named tuples containing the params of each collecting to be done. Default is an empty list.

add_collecting(*value_column*, *sql_condition*, *result_name*)
Adds a collecting to be performed by the transform method.

Parameters

- **value_column** – The name of the column containing the values to be collected.
- **sql_condition** – The sql condition to be fulfilled by the rows containing the values.
- **result_name** – The name of the result column.

Returns This ValueCollector, so that method chaining can be used.

classmethod get_param_type()
Returns the type for the named tuple with all the info about a collecting. Only relevant if you are going to add them without the use of the helper methods.

transform(*dataframe*)
Collects the values from a DataFrame.

Parameters **dataframe** – The DataFrame containing the values to collect.

Returns A new DataFrame containing the id columns of the input DataFrame, with just one row for each different id value. The DataFrame has a new column for each collecting done, with lists of the collected values.

class transformers.**ValueExtractor**(*id_columns*, *extractions_list=None*)
Bases: pipelib.pipeline_components.Transformer

Extract values from a column with strings of the kind: 'key1=value1;key2=value2'

Parameters

- **id_columns** – The columns containing the id for each row.
- **extractions_list** – A list of named tuples containing the params of each extraction to be done. Default is an empty list.

add_extraction(*source_column*, *value_to_extract*, *result_name*)
Adds a extraction to be performed by the transform method.

Parameters

- **source_column** – The column from where to extract the value.
- **value_to_extract** – The key corresponding to the value to extract.
- **result_name** – The name of the result column.

Returns This ValueExtractor, so that method chaining can be used.

classmethod **get_param_type**()
Returns the type for the named tuple modelling extractions. Only relevant if you are going to add extractions without the use of the helper methods.

transform(*dataframe*)
Performs the extractions.

Parameters **dataframe** – The dataframe containing the values to extract.

Returns A DataFrame containing the id columns and a extra column for each extraction performed.

class transformers.**ValueGetter**(*id_columns*, *value_list=None*)
Bases: pipelib.pipeline_components.Transformer

Gets values from source columns in rows that fulfills a given SQL condition.

Note: Gets one value for each id.

Parameters

- **id_columns** – The columns containing the id for each row.
- **value_list** – A list of named tuples containing the params of each value to be gotten. Default is an empty list.

add_value(*value_column*, *sql_condition*, *result_name*)
Adds a value to be getted by the transform method.

Parameters

- **value_column** – The column containing the values to be getted.
- **sql_condition** – The SQL condition to be fulfilled by the rows containing the values to be getted.
- **result_name** – The name of the result column.

Returns This ValueGetter, so that method chaining can be used.

classmethod **get_param_type**()
Returns the type for the named tuple containing the info for each value getting. Only relevant if you are going to add values without the use of the helper methods.

transform(*dataframe*)
Gets the proper values from a DataFrame.

Parameters **dataframe** – The DataFrame containing the values to get.

Returns A new DataFrame containing the id columns of the input DataFrame, as well as the gotten values.

class transformers.**ValueMapper**(*id_columns*, *mappings_list=None*)
Bases: pipelib.pipeline_components.Transformer

Substitutes values from a DataFrame.

Parameters

- **id_columns** – The columns containing the id values for each row.
- **mappings_list** – A list of named tuples containing all the info about the substitutions to be made. Default is an empty list.

add_mapping(*source_column*, *substitution_dict*, *default_value*, *result_name*, *use_default_value=True*)
Add a mapping to be performed by the transform method.

Parameters

- **source_column** – The column which values are to be mapped.
- **substitution_dict** – A dictionary that specifies the substitutions to perform. Each key-value pair represents the original and the substitute value, respectively.
- **default_value** – The value which will be used to substitute a value that can't be found in the substitution dict.
- **result_name** – The name of the result column.
- **use_default_value** (*bool*) – Whether the default value should be used. If it's set to false, values that can't be found in the substitution dict will remain unchanged.

Returns This ValueMapper, so that method chaining can be used.

classmethod get_param_type()
Returns the type of the named tuples containing the info about the substitutions. Only relevant if you are not going to use the helper method.

transform(*dataset*)
Performs the substitutions over an input DataFrame.

Parameters **dataset** – The DataFrame which values are to be substituted.

Returns A new DataFrame containing the id columns as well as the substituted columns.

class transformers.**ValueSummatorById**(*id_columns*, *summation_list=None*)
Bases: pipelib.pipeline_components.Transformer

Sums all the values in a column belonging to each id.

Generates a new column with one row for each different id value, containing the sum of the falues found in a numeric column for that value.

Parameters

- **id_columns** – The columns containing the id values for each row.
- **summation_list** – A list of named tuples containing all the info about the sums to be done. Default is an empty list.

add_sum(*value_column*, *sql_condition*, *result_name*)
Adds a sum to be performed by the transform method.

Parameters

- **value_column** – The column containing the numeric values to be summed.
- **sql_condition** – A sql condition used to filter rows before the sum.
- **result_name** – The name of the column containing the sum of values for each id.

Returns This ValueSummatorById, so that method chaining can be used.

classmethod get_param_type()
Returns the type of the named tuples containing the info about a sum. Only relevant if you are not going to use the helper method.

transform(*dataframe*)
Performs the sums over the input DataFrame.

Parameters **dataframe** – The DataFrame containing the columns to be summed.

Returns A new DataFrame containing the id columns as well as the result columns.

CHAPTER

FIVE

PYSPARK_STEP MODULE

class pyspark_step.**PysparkStep**(*data_merger*, *transformers=None*, ***, *id_column*)

Bases: pipelib.pipeline_components.Step

Performs multiple transformations over a single PySpark Dataframe, and then merges the results into a single one.

This Step provides a easy to use interface to configure transformations over a single PySpark Dataframe, runs them, and then merges the results using a DataMerger.

Parameters

- **transformers** – The list of Transformers to be used by the step. Default is an empty list. If you are going to use the easy to use interface to configure the transformations, you may want to leave it as its default value.
- **data_merger** – The DataMerger to be used to merge the results of each transformation into a single result DataFrame.
- **id_column** – The name of the column that contains the id values in the input DataFrame. Some transformations need this value to work.

binarize(*column*, *result_name*, *result_type=IntegerType*)

Configures the step to binarize a column of the input DataFrame.

The binarized column will contain a 0 where the source column were empty, had a 0, an empty string, or any value that is evaluated as False by Python. In any other case the value of the binarized column will be 1.

Parameters

- **column** – The name of the column to be binarized.
- **result_name** – The name of the result column.

Returns This Step, so that method chaining can be used.

collect_values(*source_column*, *result_name*, *sql_condition=None*, *result_type=StringType*)

Configures the step to generate a column containing a collection of certain values for each id.

The result column will have one row for each id value, containing a list with all the values found on a certain column, in rows that satisfy a given sql condition.

Parameters

- **source_column** – The column from which to take the values.
- **result_name** – The name of the result column.
- **sql_condition** – The sql condition to be satisfied by the rows containing the desired values.

Returns This Step, so that method chaining can be used.

count_rows_by_condition(*sql_condition*, *result_name*, *result_type=IntegerType*)
Configures de step to count the rows that satisfies a given sql condition for each id value in the id columns.

Parameters

- **sql_condition** – The sql condition to be satisfied by the rows to count.
- **result_name** – The name of the result column.

Returns This Step, so that method chaining can be used.

do_hashing_trick(*column_to_hash*, *number_of_features*, *result_columns_prefix*, *result_type=StringType*)
Does feature hashing trick to a column.

This is something like the One Hot Enconding, but results in a predefined number of columns.

Parameters

- **column_to_hash** – The column to hash.
- **number_of_features** – The number of columns to generate. Should be a power of two.
- **result_columns_prefix** – A common prefix for the names of the columns to be generated by the feature hashing.

Returns This Step, so that method chaining can be used.

explode_list(*column_to_explode*, *result_name*, *result_type=StringType*)
Configures the step to explode a column containing lists.

Generates a new row for each value in each list. The corresponding value for each row will appear in a new column.

Parameters

- **column_to_explode** – The row containing the lists to be exploded.
- **result_name** – The name of the column that will contain the exploded values.

Returns This Step, so that method chaining can be used.

extract_value_from_expression(*source_column*, *value_to_extract*, *result_name*)
Configures the step to extract values from a column containing strings of a given structure.

This transformation generates a column containing values extracted from a source column containing strings with a key=pair structure, separated by ';' characters. i.e.: 'key1=value1;key2=value2' and so on. More specifically, each cell of the result column will the value of the corresponding cell in the input DataFrame, for a single given key.

Parameters

- **source_column** – The column where the values can be found.
- **value_to_extract** – The key corresponding to de value to extract.
- **result_name** – The name of the result column.

Returns This Step, so that method chaining can be used.

get_json_field(*field_path*, *result_name*, *result_type=StringType*)
Configures the step to get a value from a path in a JSON DataFrame.

Note: This method should only be used when the input DataFrame contains JSON fields.

Parameters

- **field_path** – The path of the value to be get inside the JSON DataFrame.
- **result_name** – Name of the result column containing the gotten value

Returns This Step, so that method chaining can be used.

get_value(*source_column*, *result_name*, *sql_condition=None*, *result_type=StringType*)

Configures the step to generate a column with values of certain rows of the input DataFrame.

The newly generated column will contain the values of a specific column in the input dataframe, one for each row that satisfies a given sql condition.

Parameters

- **source_column** – The column from which to take the values.
- **result_name** – The name of the result column.
- **sql_condition** – The sql condition to be satisfied by the rows containing the desired values.
- **result_type**

Returns This Step, so that method chaining can be used.

keep_input_columns(*columns=None*)

Configures the step to keep some of the columns of the input DataFrame in the Output.

This is done by a transformation which output is a DataFrame consisting of the specified columns, or if none is specified, all columns of the input DataFrame.

Note: If you use this option, you may want to choose a DataMerger that can candle it appropriately.

Parameters **columns** – Columns to be kept from the input dataframe. If none is specified, all columns will be kept.

Returns This Step, so that method chaining can be used.

map_values(*source_column*, *substitution_dict*, *default_value*, *result_name*, *use_default_value=True*, *result_type=StringType*)

Configures the step to do a value mapping of a source column.

Configures de step to substitute the values of a column using a substitution dict, and then output the results.

Parameters

- **source_column** – The column which values are to be substituted.
- **substitution_dict** – A dictionary that specifies the substitutions to perform. Each key-value pair represents the original and the substitute value, respectively.
- **default_value** – The value which will be used to substitute a value that can't be found in the substitution dict.
- **result_name** – The name of the result column.
- **use_default_value** (*bool*) – Whether the default value should be used. If it's set to false, values that can't be found in the substitution dict will remain unchanged.

Returns This Step, so that method chaining can be used.

one_hot_encode(*column_to_encode*, *result_columns_prefix*, *sql_condition=None*, *result_type=StringType*)

Configures de step to perform one hot encoding to a source column.

The name of the result columns will be a given prefix followed by the encoded value.

Parameters

- **column_to_encode** – The column to be one-hot-encoded
- **result_columns_prefix** – A common prefix to the newly generated columns.
- **sql_condition** – A optional sql condition to filter the input DataFrame before the encoding.

Returns This Step, so that method chaining can be used.

split_text(*column_to_split*, *result_name*, *delimiter*, *sql_condition=None*, *result_type=StringType*)

Configures de step to split the text found in a column.

Each cell of the new column will contain a list of strings, in result of the split of the source columns string using a given character as delimiter.

Parameters

- **column_to_split** – The column containing the text to be splitted.
- **result_name** – The name of the result column.
- **delimiter** – The delimiter to be used for the text split.
- **sql_condition** – A optional sql condition to filter the input DataFrame before the split.

Returns This Step, so that method chaining can be used.

sum_columns(*list_of_columns*, *result_name*, *result_type=StringType*)

Configures the sum of two or more numeric columns of the input DataFrame.

Parameters

- **list_of_columns** (*List of str*) – List of columns to be summed
- **result_name** – Name of the column that will hold the result of the sum.

Returns This Step, so that method chaining can be used.

sum_columns_by_prefix(*prefix*, *result_name*, *result_type=StringType*)

Configures the step to sum all columns in the input DataFrame which names share a specified prefix.

Parameters

- **prefix** – The prefix of the name of the columns to be summed.
- **result_name** – The name of the result column.

Returns This Step, so that method chaining can be used.

sum_values_by_id(*column_to_sum*, *result_name*, *sql_condition=None*, *result_type=StringType*)

Configures the step to sum the values of a column belonging to each id.

Generates a new column with one row for each different id value, containing the sum of the falues found in a numeric column for that value.

Parameters

- **column_to_sum** – The column containing the numeric values to be summed.
- **result_name** – The name of the column containing the sum of values for each id.
- **sql_condition** – An optional sql condition used to filter rows before the sum.

Returns This Step, so that method chaining can be used.

sum_weighted_columns(*list_of_columns*, *list_of_weights*, *result_name*, *result_type=StringType*)
Configures the weighted sum of two or more numeric columns of the input DataFrame.

Parameters

- **list_of_columns** (*List of str*) – List of columns to be summed
- **list_of_weights** (*List of float*) – The weight factor of each corresponding column in list_of_columns
- **result_name** – Name of the column that will hold the result of the sum.

Returns This Step, so that method chaining can be used.

CHAPTER

SIX

UTILS MODULE

exception utils.**DataIOException**
Bases: Exception

A exception raised by an Input or Output Provider when an error occurs.

utils.**get_date_sub_path**(*date: datetime.date*)
Generates a subpath from a date, according to the structure: year/yearmonth/yearmonthday

Parameters **date** (*datetime.date*) – The date used for the subpath generation.

Returns The subpath corresponding to the input date.

utils.**load_dataframe**(*read_path*, *sqlCtx*, *delimiter=';'*, *mode='DROPMALFORMED'*, *schema=None*)
Loads a PySpark DataFrame from a csv file.

Parameters

- **read_path** – The path where the csv data can be found.
- **sqlCtx** – The PySpark SQLContext to use for the read.
- **delimiter** – The delimiter of the csv file. Default is ';'.
- **mode** – The mode of the read. Default is 'DROPMALFORMED'. More modes can be found on PySpark documentation.
- **schema** (*pyspark.sql.types.StructType*) – The schema of the data. If none is provided, it will be inferred.

Returns The loaded DataFrame.

utils.**write_dataframe**(*dataframe*, *path*)
Writes a PySpark dataframe to a csv file.

Parameters

- **dataframe** – The dataframe to be written.
- **path** – The path were the dataframe have to be written.

Returns None

ANEXO B:
DOCUMENTACIÓN
DE LA EXTENSIÓN DE TEALEAF

Tealeaf Extension to Pipelines Library Documentation

Release 1

Óliver L. Sanz

July 04, 2018

CONTENTS

CHAPTER

ONE

CONNECTOR MODULE

class connector.**TealeafConnector**

Bases: object

Merges the three tealeaf datasets (sessions, events and report groups) into a single one.

Puts all data from sessions, events and report groups datasets into a single one, suited to be used by the pipelines library (pipelib).

merge(*sessions_df*, *events_df*, *dimensions_df*)

Merges the Tealeaf datasets into a single one.

Parameters

- **sessions_df** (*pyspark.DataFrame*) – Teleaf sessions data, which must contain the following columns: SESSION_KEY, SESSION_DURATION, HIT_COUNT and SESSION_TIMESTAMP
- **events_df** (*pyspark.DataFrame*) – Tealeaf events data, which must contain the columns: SESSION_KEY, HIT_KEY, EVENT_ID and FACT_VALUE
- **dimensions_df** (*pyspark.DataFrame*) – Tealeaf report groups data, which must contain the columns: SESSION_KEY, EVENT_ID, HIT_KEY, GROUP_ID, FACT_DIM_1, FACT_DIM_2, FACT_DIM_3, FACT_DIM_4 and GROUP TYPE

Returns The merged DataFrame, containing all the columns from the source ones, plus a column ROW_TYPE, showing which row came from which data set.

CHAPTER

TWO

TRANSFORMERS MODULE

class transformers.**EventChecker**(*session_id_column*, *event_id_column*, *row_type_column*, *param_list=None*)

Bases: pipelib.pipeline_components.Transformer

Checks if a certain event occurs for each session.

Parameters

- **session_id_column** – The column containing the session ids.
- **event_id_column** – The column containing the event ids.
- **row_type_column** – The column containing the row type.
- **param_list** – A list of named tuples containing the necessary params for each event check. Default value is an empty list.

add_event(*event_id*, *result_name*)

Add a event to be checked.

Parameters

- **event_id** (*str*) – The id of the event to be checked.
- **result_name** – The name of the result column containing the check.

Returns This EventChecker, so that method chaining can be used.

classmethod get_param_type()

Returns a named tuple type with all the params necessary to perform an event check. Ignore this if you are going to use the helper methods.

transform(*dataframe*)

Performs the event check.

Parameters **dataframe** – A DataFrame containing the events to be checked.

Returns A new DataFrame containing the session id, as well as the check of each of the events for each session.

class transformers.**EventCounter**(*session_id_column*, *event_id_column*, *row_type_column*, *param_list=None*)

Bases: pipelib.pipeline_components.Transformer

Counts event occurrences for each session.

Parameters

- **session_id_column** – The column containing the session ids.
- **event_id_column** – The column containing the event ids.

- **row_type_column** – The column containing the row type.
- **param_list** – A list of named tuples containing the necessary params for each event count. Default value is an empty list.

add_event(*event_id*, *result_name*)
Add a event to be counted

Parameters

- **event_id** (*str*) – The id of the event to be counted.
- **result_name** – The name of the result column containing the count.

Returns This EventCounter, so that method chaining can be used.

classmethod get_param_type()
Returns a named tuple type with all the params necessary to perform an event count. Ignore this if you are going to use the helper methods.

transform(*dataframe*)
Performs the event count.

Parameters **dataframe** – A DataFrame containing the events to be counted.

Returns A new DataFrame containing the session id, as well as the count of each of the events for each session.

class transformers.**FactValueChecker**(*session_id_column*, *event_id_column*, *row_type_column*, *param_list=None*)
Bases: pipelib.pipeline_components.Transformer

Checks if a fact value appears in a given event type for each session.

Parameters

- **session_id_column** – The column containing the session ids.
- **event_id_column** – The column containing the event ids.
- **row_type_column** – The column containing the row type.
- **param_list** – A list of named tuples containing the necessary params for each fact value check. Default value is an empty list.

add_value_to_check(*event_id*, *fact_value*, *result_name*)
Sets a fact value of an event to be checked for each session.

Parameters

- **event_id** – The id of the event which fact values are to be checked.
- **fact_value** – The fact value to check.
- **result_name** – Te name of the result column.

Returns This FactValueChecker, so that method chaining can be used.

classmethod get_param_type()
Returns a named tuple type with all the params necessary to check if a fact value appears for a session. Ignore this if you are going to use the helper methods.

transform(*dataframe*)
Performs the fact value checks.

Parameters **dataframe** – A DataFrame containing the fact values to be checked.

Returns A new DataFrame containing the session id, as well as the check of each of the fact values for each session

class transformers.**FactValueCounter**(*session_id_column*, *event_id_column*, *row_type_column*, *param_list=None*)

Bases: pipelib.pipeline_components.Transformer

For each id, counts the number of appearances of a given fact value in a given event.

Parameters

- **session_id_column** – The column containing the session ids.
- **event_id_column** – The column containing the event ids.
- **row_type_column** – The column containing the row type.
- **param_list** – A list of named tuples containing the necessary params for each fact value count. Default value is an empty list.

add_value_to_count(*event_id*, *fact_value*, *result_name*)

Add a value to be counted by the transform method.

Parameters

- **event_id** (*str*) – The id of the event containing the fact value to be counted.
- **fact_value** – The fact value to be counted.
- **result_name** – The name of the result column containing the count.

Returns This FactValueCounter, so that method chaining can be used.

classmethod get_param_type()

Returns a named tuple type with all the params necessary to perform a fact value count. Ignore this if you are going to use the helper methods.

transform(*dataframe*)

Performs the fact value counts.

Parameters **dataframe** – A DataFrame containing the fact values to be counted.

Returns A new DataFrame containing the session id, as well as the count of each of the fact values for each session

class transformers.**FactValueFinder**(*session_id_column*, *event_id_column*, *param_list=None*)

Bases: pipelib.pipeline_components.Transformer

Gets the fact value of a given event, for each session.

Note: If that event occurs more that once for a session, any of the fact values will be taken, and just one of them.

Parameters

- **session_id_column** – The column containing the session ids.
- **event_id_column** – The column containing the event ids.
- **param_list** – A list of named tuples containing the necessary params for getting each fact value. Default value is an empty list.

add_event(*event_id*, *result_name*)

Adds a event, so that its fact value will be obtained by the transform method.

Parameters

- **event_id** – The id of the event.
- **result_name** – The name of the result column.

Returns This FactValueFinder, so that method chaining can be used.

classmethod **get_param_type**()
Returns a named tuple type with all the params necessary to get fact values from an event. Ignore this if you are going to use the helper methods.

transform(*dataframe*)
Gets the fact values for each of the given events.

Parameters **dataframe** – A DataFrame containing the fact values to be obtained.

Returns A new DataFrame containing the session id, as well as the the fact values for each of the given events.

class transformers.**LastFactValueFinder**(*session_id_column*, *event_id_column*, *hit_key_column*, *fact_value_column*, *timestamp_column*, *param_list=None*)
Bases: pipelib.pipeline_components.Transformer

Gets the fact value of the last occurrence of a given event for each session.

Note: The event must be timestamped through a report group of the timestamp type.

Parameters

- **session_id_column** – The column containing the session ids.
- **event_id_column** – The column containing the event ids.
- **hit_key_column** – The column containing the hit keys.
- **fact_value_column** – The column containing the fact values.
- **timestamp_column** – The column containing the timestamps.
- **param_list** – A list of named tuples containing the necessary params for each last fact value search. Default value is an empty list.

add_event(*event_id*, *result_name*)
Add a event to be searched for its last fact values.

Parameters

- **event_id** (*str*) – The id of the event.
- **result_name** – The name of the result column containing the last fact values.

Returns This LastFactValueFinder, so that method chaining can be used.

classmethod **get_param_type**()
Returns a named tuple type with all the params necessary to get the last fact values of an event. Ignore this if you are going to use the helper methods.

transform(*dataframe*)
Performs the search of the last fact values.

Parameters **dataframe** – A DataFrame containing the timestamped events and its fact values.

Returns A new DataFrame containing the session id, as well as the obtained last fact values.

class `transformers.`**`SessionSlicer`**(*session_id_column*, *event_id_column*, *hit_key_column*, *report_group_type_column*, *timestamp_column*, *slicer_event_id=None*, *result_name=None*)

Bases: `pipelib.pipeline_components.Transformer`

Breaks sessions into smaller pieces.

Labels each row with a 'subsession' id, splitting each session into multiple subsessions. This is done using a slicing event as point of subsession change. e.g.: if the slicing event occurs at time 5 and time 10, all other events in that session occurring between times 0 and 5 will belong to subsession 1; al events between times 6 and 10 will belong to subsession 2; and all events that occurs past time 10 will belong to subsession 3.

Notes

The timestamps must be of type pyspark.sql.types.IntegerType

Parameters

- **session_id_column** – The column containing the session ids.
- **event_id_column** – The column containing the event ids.
- **hit_key_column** – The column containing the hit keys.
- **report_group_type_column** – The column containing the report group type.
- **timestamp_column** – The column containing the timestamp of type IntegerType.
- **slicer_event_id** – The id of the event used to do the slicing. Default value is None.
- **result_name** – The name of the subsession column. Default value is None.

Note: If you don't specify the values of slicer_event_id or result_name, you will have to do so before calling the transform method, either manually or using the set_slicing method.

`set_slicing`(*slicer_event_id*, *result_name*)

Sets the slicer event id and result column name.

`transform`(*dataframe*)

Slices the sesions of a DataFrame

Parameters **dataframe** – The dataframe which sessions are to be sliced.

Returns The input DataFrame, with a new column specifying the subsession id for each row. For non timestamped rows, the subsesion id will be null.

CHAPTER

THREE

TEALEAF_STEP MODULE

class tealeaf_step.**TealeafStep**(*data_merger*, *transformers=None*, *session_id_column='SESSION_KEY'*, *event_id_column='EVENT_ID'*, *row_type_column='ROW_TYPE'*, *hit_key_column='HIT_KEY'*, *fact_value_column='FACT_VALUE'*, *timestamp_column='EVENT_TIMESTAMP'*, *report_group_type_column='GROUP_TYPE'*, *report_group_id_column='RG_ID'*, *dim_0_column='FACT_DIM_0'*, *dim_1_column='FACT_DIM_1'*, *dim_2_column='FACT_DIM_2'*, *dim_3_column='FACT_DIM_3'*)

Bases: pipelib.pyspark_step.PysparkStep

Performs multiple transformations over a single PySpark Dataframe, and then merges the results into a single one.

This Step provides a easy to use interface to configure transformations over a single PySpark Dataframe, runs them, and then merges the results using a DataMerger.

Parameters

- **transformers** – The list of Transformers to be used by the step. Default is an empty list. If you are going to use the easy to use interface to configure the transformations, you may want to leave it as its default value.
- **data_merger** – The DataMerger to be used to merge the results of each transformation into a single result DataFrame.
- **session_id_column** – The name of the column that contains the session id values in the input DataFrame.
- **event_id_column** – The name of the column containing the event id values.
- **row_type_column** – The name of the column containing the row type values.
- **hit_key_column** – The name of the column containing the hit key values.
- **fact_value_column** – The name of the column containing the fact values of events.
- **timestamp_column** – The name of the column containing the timestamp of the timestamped hits.
- **report_group_type_column** – The name of the column containing the type of the report groups.

check_event(*event_id*, *result_name*, *result_type=StringType*)

Checks if a given event occurs, for each session.

Parameters

- **event_id** (*str*) – The id of the event to be checked.

- **result_name** – The name of the result column.

Returns This step, so that method chaining can be used.

check_fact_value(*event_id*, *value_to_check*, *result_name*, *result_type=StringType*)
Checks if a fact value appears in a given event type for each session.

Parameters

- **event_id** – The id of the event which fact values are to be checked.
- **value_to_check** – The fact value to check.
- **result_name** – Te name of the result column.

Returns This step, so that method chaining can be used.

count_dimension_value(*event_id*, *report_group_id*, *dimension_index*, *value_to_count*, *result_name*, *result_type=StringType*)
Counts the appearances of a value in a dimension of a report group.

Parameters

- **event_id** – The id of the event holding the report group.
- **report_group_id** – The id of the report group.
- **dimension_index** – The index, from 0 to 3, of the dimension where the value will be counted.
- **value_to_count** – The value to be counted.
- **result_name** – The name of the result column.

Returns This step, so that method chaining can be used.

count_event(*event_id*, *result_name*, *result_type=StringType*)
Counts how many events of a given type occurred to each session.

Parameters

- **event_id** (*str*) – The id of the event to be counted.
- **result_name** – The name of the column containing the count.

Returns This step, so that method chaining can be used.

count_fact_value(*event_id*, *value_to_count*, *result_name*, *result_type=StringType*)
For each id, counts the number of appearances of a given fact value in a given event.

Parameters

- **event_id** – The id of the event.
- **value_to_count** – The fact value to be counted.
- **result_name** – The name of the result column.

Returns This step, so that method chaining can be used.

get_dimension_value(*event_id*, *report_group_id*, *dimension_index*, *result_name*, *result_type=StringType*)
Gets a dimension value from a report group.

Parameters

- **event_id** – The id of the event holding the required report group.
- **report_group_id** – The id of the report group.

- **dimension_index** – The index, from 0 to 3, of the dimension to get.
- **result_name** – The name of the result column.

Returns This step, so that method chaining can be used.

get_fact_value(*event_id*, *result_name*, *result_type=StringType*)
Gets the fact value of a given event, for each session.

If that event occurs more that once for a session, any of the fact values will be taken, and just one of them.

Parameters

- **event_id** – The id of the event.
- **result_name** – The name of the result column.

Returns This step, so that method chaining can be used.

get_last_fact_value(*event_id*, *result_name*, *result_type=StringType*)
Gets the fact value of the last occurrence of a given event for each session.

Note: The event must be timestamped.

Parameters

- **event_id** (*str*) – The id of the timestamped event to be used.
- **result_name** – The name of the result column.

Returns This step, so that method chaining can be used.

slice_sessions(*slicer_event_id*, *subsession_id_column_name*)
Breaks sessions into smaller pieces.

Labels each row with a 'subsession' id, splitting each session into multiple subsessions. This is done using a slicing event as point of subsession change. e.g.: if the slicing event occurs at time 5 and time 10, all other events in that session occurring between times 0 and 5 will belong to subsession 1; al events between times 6 and 10 will belong to subsession 2; and all events that occurs past time 10 will belong to subsession 3.

Parameters

- **slicer_event_id** – The id of the slicer event.
- **subsession_id_column_name** – The name of the column containing the subsession id.

Returns This step, so that method chaining can be used.

ANEXO C:
ÍNDICE DE CONTENIDOS
DEL MEDIO EXTERNO

Índice de contenidos del medio de almacenamiento

Estos son los contenidos incluídos en el medio de almacenamiento incluído con esta memoria.

- **TFG Óliver L Sanz - Memoria.pdf**: la memoria en formato PDF.
- **example_data/raw/**: Este directorio contiene los datos de ejemplo usados en los procesados de datos. Se corresponden con los mostrados en el capítulo 5 de la memoria.
- **source/**: Este directorio contiene el código implementado durante el proyecto.
    - **library/pipelib/**: Contiene el código de la biblioteca Pipelines Library, desarrollada en el capítulo 4 de la memoria.
    - **library/tealeaf/**: Contiene el código de la extensión de Tealeaf para la biblioteca, desarrollada en el capítulo 5 de la memoria.
    - **scripts/**: Contiene los scripts desarrollados durante el capítulo 5 de la memoria.
        - **s1_merge_tealeaf_data.py**: Script Python que utiliza la biblioteca para unificar los datos del directorio example_data/raw/ en una sola tabla, que deposita en example_data/merged/. Estos nuevos datos serán usados por los demás scripts.
        - **s2_audiences.py**: Script Python que utiliza la biblioteca para realizar la extracción de características de los datos detallada en el capítulo 5. El resultado será depositado en example_data/audiences/.
        - **s3_sliced_audiences.py**: Script Python que utiliza la biblioteca para realizar la extracción de características de los datos detallada al final del capítulo 5. El resultado será depositado en example_data/sliced_audiences/.