Big Data GIS (GeoAnalytics) y Análisis Espacial

Análisis espacial para Big Data

PARTE - 3

esri España
THE SCIENCE OF WHERE

Contenido

Copyright

Versión del curso 1.0. Fecha de lanzamiento de la versión de junio de 2026.

Esri resources

Utilice los recursos indicados a continuación para adquirir más conocimientos sobre ArcGIS y analizar el conjunto de aplicaciones disponibles para el usuario para el tratamiento de datos espaciales.

Recursos e-Learning

Los cursos dirigidos por instructores de Esri y los recursos de e-Learning lo ayudan a desarrollar y aplicar las habilidades de ArcGIS, los flujos de trabajo recomendados y las mejores prácticas. https://www.esri.com/training/

Planificación para organizaciones

Desde el departamento de formación de Esri España se desarrollarán itinerarios formativos que cumplan con las necesidades específicas de las empresas. Para ello póngase en contacto con formacion@esri.es

Certificación de Esri

Las diferentes certificaciones técnicas de Esri permiten avalar los conocimientos de los usuarios sobre nuestra tecnología. Dichas pruebas engloban a grandes rasgos desktop, developer y *Enter*prise. Para obtener más información acceda al enlace indicado a continuación https://www.esri.com/training/certification/

Redes sociales y publicaciones

Twitter: @EsriTraining y @Esri

Esri en LinkedIn: linkedin.com/company/esri

Blog de formación de Esri: esri.com/trainingblog

Publicaciones de Esri: acceda a las ediciones online de ArcNews, ArcUser y ArcWatch en esri.com/esri-news/publications

Boletín informativo de Esri: Suscríbete a go.esri.com/preferences

Otros boletines informativos de Esri: boletines informativos específicos go.esri.com/preferences

Esri Press

Documentos específicos sobre la ciencia y la tecnología de SIG tanto en sectores públicos como en privados. esripress.esri.com

Bibliografía GIS

Índice completo de revistas, conferencias, libros y documentación relacionada con GIS, referencias y materiales de texto completo. gis.library.esri.com

Documentación y tutoriales de ArcGIS

Información detallada, tutoriales y documentación para los productos de ArcGIS.

ArcGIS Online: arcgis.com

ArcGIS Desktop: desktop.arcgis.com

ArcGIS *Enter*prise: *Enter*prise.arcgis.com

GeoNet

Comunidad online de usuarios y expertos en GIS. esri.com/geonet

Eventos de Esri

Conferencias de Esri y reuniones donde los usuarios comparten experiencias, conocimientos etc. Se trata de una forma excelente intercambiar información. esri.com/events

Videos de Esri

Repositorio de vídeos de ponentes, técnicos de Esri y expertos de productos youtube.com/user/esri

ArcGIS for Personal Use

Mejore sus habilidades de GIS en casa usando ArcGIS para mejorar sus proyectos personales. El programa ArcGIS for Personal Use incluye una licencia de 12 meses para ArcGIS Desktop, extensiones y una cuenta de usuario nominada de ArcGIS Online con 100 créditos de servicio. esri.com/personaluse

Diccionario GIS

Glosario de términos GIS . https://support.esri.com/en/other-resources/gis-dictionary

Introducción al curso

ArcGIS GeoAnalytics Engine es una biblioteca de análisis espacial especializada en el análisis de grandes cantidades de datos. A lo largo de este curso conoceremos los detalles de la biblioteca, repasaremos las funciones y herramientas disponibles, así como su integración en el resto del sistema ArcGIS.

Iconos utilizados en el documento

 Tiempos estimados. Indican aproximadamente cuantos minutos se tarda en completar el ejercicio.

 Las notas indican información adicional, excepciones o circunstancias especiales sobre temas específicos del curso.

 Prácticas recomendadas que mejorar la eficiencia y ahorran tiempo

 Recursos de Esri que permiten ampliar conocimientos sobre temas específicos

 Recursos adicionales

 Advertencias que indican posibles problemas o acciones que deben evitarse

ArcGIS GeoAnalytics Engine

Qué es

ArcGIS GeoAnalytics Engine es una biblioteca de análisis espacial que nos permite procesar y analizar grandes volúmenes de datos espaciales de forma eficiente y escalable. Esta capacidad analítica se logra gracias a su integración con el framework de **Apache Spark**, lo que posibilita la ejecución de operaciones GIS avanzadas directamente sobre entornos de big data. Además, ArcGIS GeoAnalytics Engine permite trabajar mediante **Python, Scala y SQL**, facilitando su incorporación en flujos de trabajo de análisis de datos y ciencia de datos.

Características principales

Integración nativa con Apache Spark

ArcGIS GeoAnalytics Engine se integra de forma directa con Apache Spark, lo que permite el procesamiento de datos espaciales de manera distribuida y a gran escala. Además, al ejecutarse sobre Apache Spark directamente trabaja directamente con Spark DataFrames, y los análisis espaciales se distribuyen automáticamente entre los nodos del clúster, permitiendo así la escalabilidad.

Facilidad de uso

Se exponen APIs claras y fáciles de usar que extienden las capacidades de Spark permitiendo construir pipelines con soporte espacial sin necesidad de conocimientos avanzados en GIS tradicional. Esto permite una fácil adopción por parte de analistas GIS y científicos e ingenieros de datos.

Funciones espaciales, SQL y de seguimiento

La biblioteca de ArcGIS GeoAnalytics Engine proporciona un amplio conjunto de funciones para crear y manipular geometrías, evaluar relaciones espaciales, analizar datos espacio-temporales y gestionar rutas. Estas funciones nos permiten ejecutar análisis espaciales, espacio-temporales y estadísticos mediante pocas líneas de código simplificando así procesos complejos y aumentando la productividad.

Indexación espacial automática

El motor gestiona automáticamente la creación de índices espaciales, optimizando de forma transparente operaciones costosas como los joins espaciales, intersecciones y consultas de proximidad, sin requerir configuración adicional por parte del usuario.

Lectura y escritura de múltiples fuentes de datos

ArcGIS GeoAnalytics Engine nos permite cargar y guardar datos espaciales desde y hacia diferentes fuentes y formatos de datos como shapefiles, feature services o vector tiles, lo que facilita la interoperabilidad con infraestructuras GIS existentes.

Arquitectura cloud-native

ArcGIS GeoAnalytics Engine es una biblioteca diseñada para entornos cloud líderes del sector como son Databricks, Amazon EMR, Azure Synapse y Google Cloud Dataproc.

Herramientas y funciones

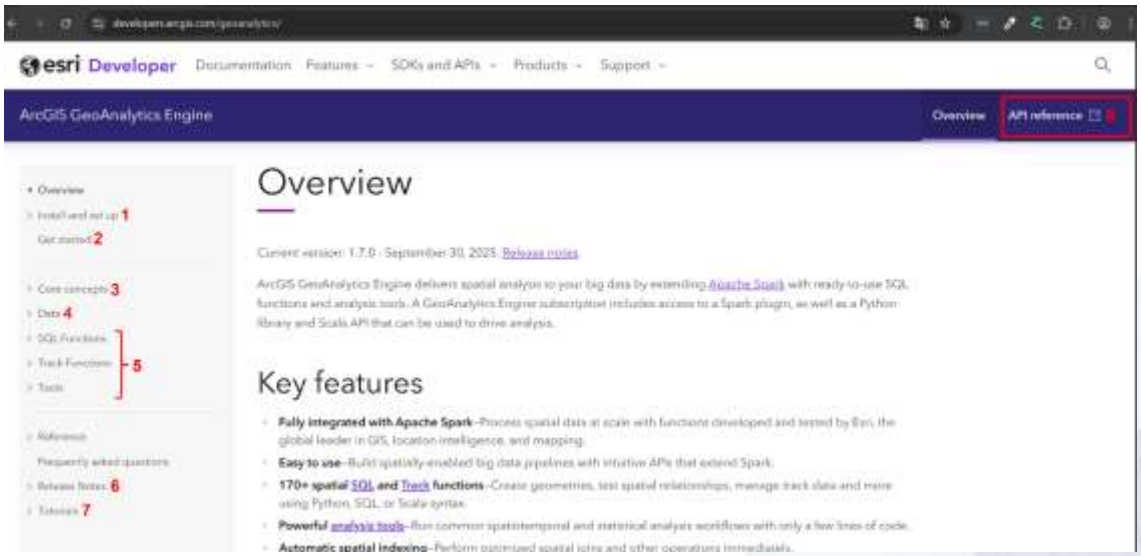
Cuando hablamos de capacidades analíticas en ArcGIS GeoAnalytics Engine solemos hablar de funciones y herramientas que tienen algunas diferencias clave que se deben tener en cuenta.

Hay más de 150 **funciones SQL** incluidas en la biblioteca que extienden la API de Spark SQL permitiendo así la consulta espacial sobre columnas de un DataFrame. Estas funciones permiten crear geometrías, operar sobre ellas, evaluar relaciones espaciales, resumir geometrías… Este tipo de funciones suele utilizar una o dos columnas para hacer la operación, frente a las **herramientas** incluidas que utilizan toda la tabla para calcular el resultado. Además, las herramientas utilizan todas las filas y columnas si la herramienta lo necesita, mientras que las funciones SQL se ejecutan en cada fila del DataFrame.

Documentación

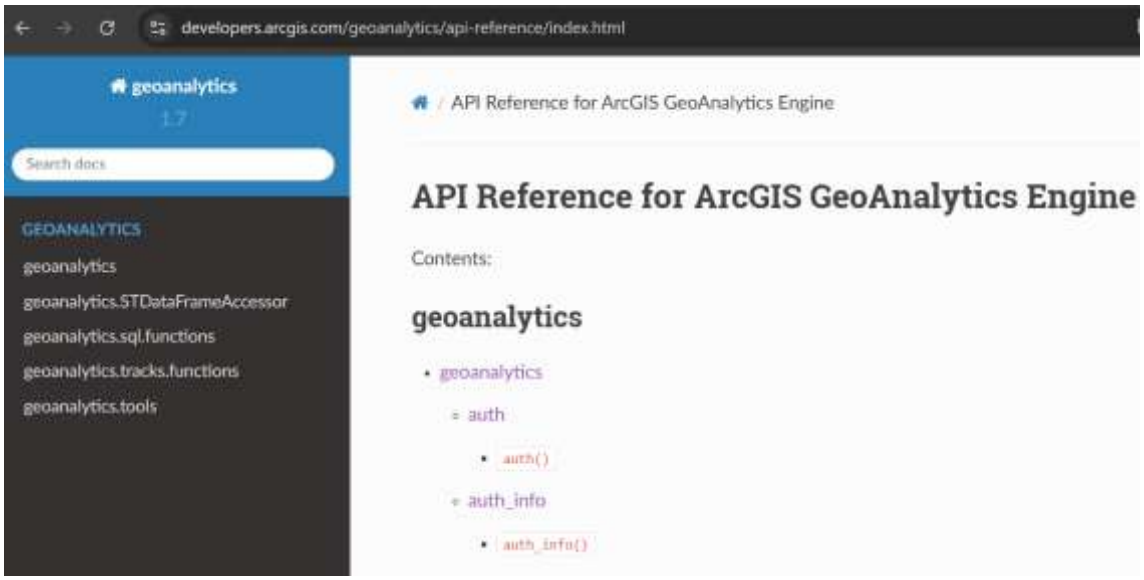
La documentación de la biblioteca es un elemento clave para poder estar actualizado y conocer los cambios de las nuevas versiones de la biblioteca.

Desde la página de GeoAnalytics Engine tendremos acceso a todos los recursos de GeoAnalytics Engine: https://developers.arcgis.com/geoanalytics



1. Desde el apartado de *Install and setup* podremos ver tutoriales para ver cómo integrar la biblioteca en los diferentes entornos. Así como cómo configurar los localizadores y redes.
2. En *Get Started* un pequeño tutorial que repasa un flujo general en GeoAnalytics Engine.
3. En *Core concepts* podemos repasar conceptos clave esenciales a la hora de trabajar con datos geoespaciales como son los sistemas de coordenadas, la geocodificación, relaciones espaciales, etc. Además, tenemos acceso también al glosario de términos generales que definen conceptos clave y extendidos relacionados con la tecnología geoespacial.
4. El apartado de *Data* tiene acceso directo a ejemplos y explicaciones de cómo integrar cada tipo de dato compatible.
5. Tanto en los apartados de funciones SQL, de trackeo y herramientas espaciales podemos ver pequeños ejemplos y explicaciones de cada análisis.
6. Las novedades y grandes cambios de todas las versiones de la biblioteca están en el apartado de *Release Notes.*
7. El apartado de *Tutorials* contiene varios tutoriales muy completos y paso a paso explicando funcionalidades básicas como la carga de datos así como temas más complejos como la visualización y análisis de datos.

8. Toda esta documentación es complementaria a la documentación pura de la biblioteca a la que podemos acceder desde la pestaña de *API reference*. Ahí encontraremos todos los métodos y parámetros definidos.



Geometría

La geometría es el dato esencial para poder trabajar con datos geoespaciales, concretamente, con ArcGIS GeoAnalytics Engine representamos cada fila como un elemento geográfico, con sus variables e información geográfica. Los diferentes tipos de geometría que se soportan implementan la especificación OpenGIS Simple Features Implementation Specification for SQL 1.2.1. y son:

- *point*
- *multipoint*
- *linestring:* para líneas y multilíneas
- *polygon:* para polígonos y polígonos multiparte.
- *geometry:* es un tipo genérico que se utiliza cuando una columna contiene más de un tipo de geometría o si se desconecte el tipo.

Dentro de los atributos de las geometrías con las que estemos trabajando debemos tener en cuenta la referencia espacial. Por defecto, si no la especificamos, la referencia espacial será 0 indicando que es desconocida y, por tanto, no podremos localizar los elementos en el mapa ni hacer análisis espaciales. Para definir la referencia espacial podremos usar funciones como ST_SRID o ST_SRText. Si estamos trabajando con shapefiles o feature services, la referencia espacial suele venir definida y se añadirá a la geometría automáticamente. Esto podríamos comprobarlo con la función get_spatial_reference().

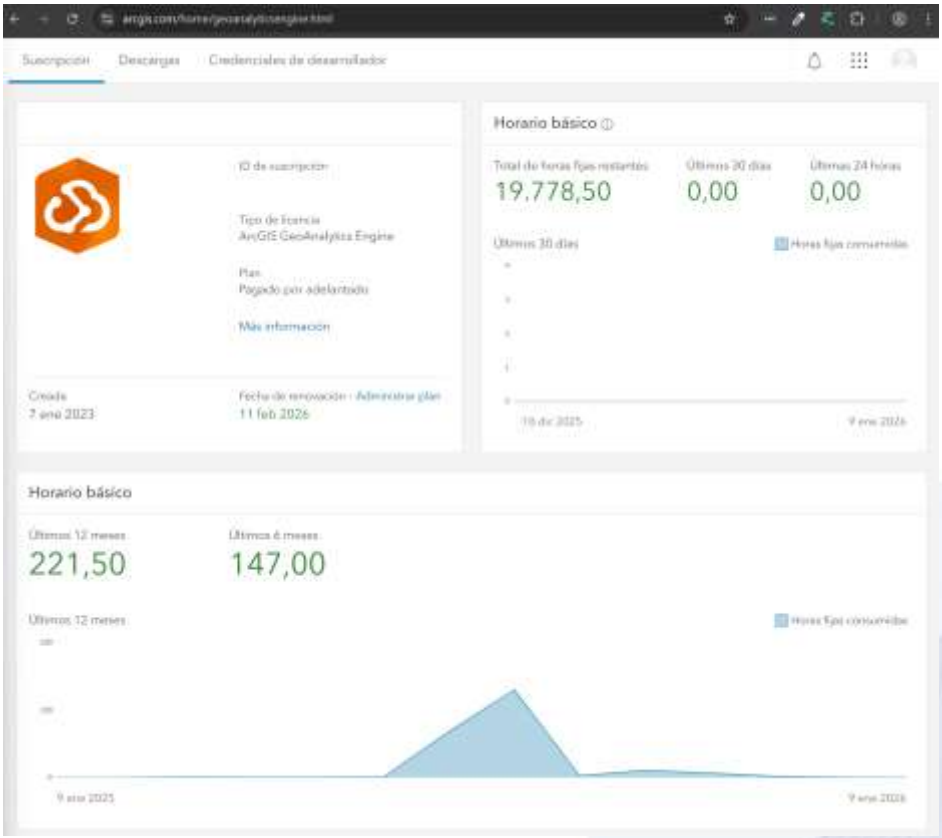
Licenciamiento y autenticación

Para poder utilizar las capacidades de GeoAnalytics Engine necesitamos tener una licencia activa. Hay dos tipos de licenciamiento para GeoAnalytics Engine:

- De forma **conectada** e inicio de sesión con usuario y contraseña o con una API key.
- En modo **desconectado** a través de un archivo de licenciamiento.

Licenciamiento en modo conectado

Este licenciamiento es una suscripción prepagada a ArcGIS GeoAnalytics Engine que incluye una cantidad determinadas de horas para procesamiento big data. Por ello es imprescindible que el clúster de Spark pueda conectarse a https://www.arcgis.com para generar informes de uso.



Al autorizar con una licencia conectada, cualquier trabajo de Spark que incluya la funcionalidad de GeoAnalytics Engine se medirán los milisegundos de cómputo y se deducirá de las unidades totales disponibles del plan. Por ejemplo, suponiendo que se ejecute un trabajo de Spark con funcionalidad de GeoAnalytics Engine en un clúster con 60 núcleos durante 1 minuto, el uso total notificado será de 60 núcleos por 60.000 milisegundos, es decir, 1 hora que será restado al total de horas restantes.

El uso de la biblioteca se puede consultar desde la página de informes y también con la función geoanalytics.usage() que nos devolverá un DataFrame donde cada fila representará los últimos usos de GeoAnalytics Engine.

Licenciamiento en modo desconectado

Este tipo de licenciamiento nos permite usar GeoAnalytics Engine en entornos Spark totalmente desconectados de la red. Para licenciarlo deberemos utilizar un archivo *.ecp* proporcionado por Esri para autorizar la biblioteca. Al contrario que con el modo desconectado, con una licencia desconectada no se registra ni se informa del uso del clúster Spark y no se requiere conexión a internet.

Autenticación

Para utilizar GeoAnalytics Engine debemos tener una licencia válida y para iniciar sesión con ella podemos utilizar usuario y contraseña, una API key o un archivo de licencia de Esri (.epc). Si no hemos iniciado sesión correctamente y, por tanto, el módulo no está autorizado, nos aparecerá el siguiente error:

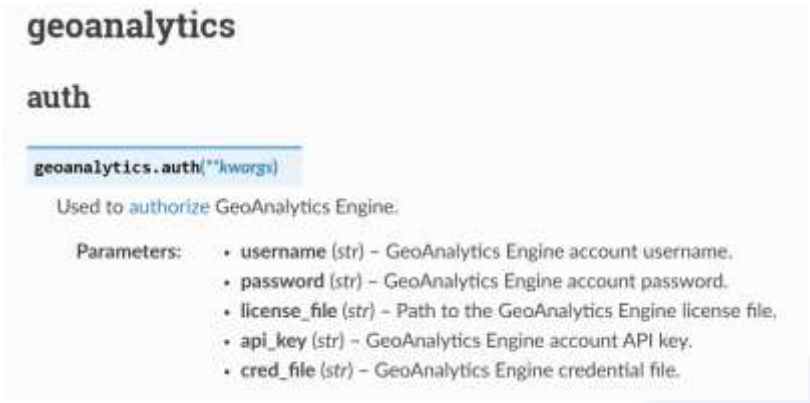
```
com.esri.geoanalytics.internal.AuthError: Not authorized
```

Para autorizar GeoAnalytics Engine podemos hacerlo de dos formas:

- Con el método geoanalytics.auth() en un notebook, script o shell.
- Definiendo una propiedad en la configuración de Spark.

Método geoanalytics.auth()

Podemos autorizar GeoAnalytics Engine en un notebook, script o shell con el método geoanalytics.auth() pero los parámetros varíaran en función del modo de autenticación:



geoanalytics

auth

geoanalytics.auth(**kwargs)

Used to authorize GeoAnalytics Engine.

Parameters:
- username (str) – GeoAnalytics Engine account username.
- password (str) – GeoAnalytics Engine account password.
- license_file (str) – Path to the GeoAnalytics Engine license file.
- api_key (str) – GeoAnalytics Engine account API key.
- cred_file (str) – GeoAnalytics Engine credential file.

- Proporcionando un usuario y contraseña de una suscripción activa de GeoAnalytics Engine. Esto autorizará la biblioteca de forma segura a través de OAuth 2.0.

```
import geoanalytics
geoanalytics.auth(username="User1", password="p@ssw0rd")
```

- Con una API key creada en el cuadro de mando, esto también se base en OAuth 2.0.

```
geoanalytics.auth(api_key="AAPTxy8BH1VEsoebNVZXo8HurKKch0YH8jNg1Ky5-
ShozdrTFkcl6Y8EbrIxQH4RqcMP-33_GNrZR7iKND0uZ_yjWZXs8-
rHjPFVJLT6eP7TCB2YhqSx2RU_1UMRPEwaSkWfYtCZrK470jGnV0-s-
eHF1LfHvo89bYamilUzzBEd8dsdDJgV5UBwI0rPePzOoNoCBXUdKuje6TxNB2ImMio4rkWVY5NNGVbm1MAERbUsH
xY.AT1_iUlET9vO")
```

```
import geoanalytics
geoanalytics.auth(api_key="XXXXX")
```

- A través de una ruta a un archivo con las credenciales que contenga el usuario y contraseña o una API key. En ambos casos, el uso de un archivo de credenciales autoriza la biblioteca de forma segura a través de internet mediante OAuth 2.0. Hay que tener en cuenta que el archivo debe ser accesible para el controlador de Spark.

```
import geoanalytics
geoanalytics.auth(cred_file="/engine/creds.txt")
```

- Para una autorización sin conexión deberemos proporciona la ruta al archivo de licencia y éste debe ser accesible para el controlador de Spark.

```
import geoanalytics

# Ejemplo 1:  Archivo de autenticación en un path local
geoanalytics.auth(license_file="/engine/license.ecp")
# Ejemplo 2: Archivo de autenticación guardado en S3
geoanalytics.auth(license_file="s3://my_bucket/engine/license.ecp")
```

Propiedades de configuración de Spark

Podemos autorizar GeoAnalytics Engine a través de la configuración de Spark usando la línea de comandos, un archivo de configuración o directamente en el SparkContext en la sesión de Spark. Tendremos que definir una de las dos propiedades de Spark que aparecen a continuación:

- `spark.geoanalytics.auth.cred.file`: la ruta a un archivo que contenga el usuario y contraseña o una API key de una suscripción activa de GeoAnalytics Engine.
- `spark.geoanalytics.auth.license.file:` la ruta a un archivo de licenciamiento para el modo desconecta. El archivo debe ser accible para el driver de Spark.

Verificación la autorización

Como hemos comentado, si no estamos autorizados o no hemos iniciado sesión, no se ejecutará nada de la biblioteca de GeoAnalytics Engine y seremos consciente de que algo ha fallado en el proceso de autenticación. Aún así, podemos comprobarlo con la función geoanalytics.auth_info() que nos devolverá un DataFrame con las propiedades de uso y su valor correspondiente:

- **session_uptime:** cantidad de tiempo en milisegundos que GeoAnalytics Engine ha estado autorizado en la sesión actual.
- **auth:** el tipo de autorización en uso. Las opciones son:
  - **token/auth:** cuando se ha iniciado sesión con usuario y contraseña.
  - **token/apikey:** cuando se ha utilizado una API key.
  - **license/file:** cuando se ha autorizado con un archivo de licencia.
- **scope:** scope de la autenticación actual.
- **offline:** booleano que nos devolverá un verdadero si estamos en modo desconectado.
- **metered:** booleano que aparecerá como verdadero si está midiendo el uso de la biblioteca. Solo se mide para licencias conectadas.
- **authorized:** aparecerá como verdadero si el módulo está correctamente autorizado y listo para su uso.

Si estamos en modo desconectado también nos devolverá la propiedad:

- **expires:** la fecha de caducidad de la licencia actual.

Por el contrario, si estamos en modo conectado nos devolverá también las siguientes propiedades:

- **billing_type:** el tipo de plan para las licencias conectadas.
- **available_hours:** horas de ejecución restantes disponibles.
- **session_ussage:** los milisegundos de computación utilizados en la sesión actual.

Si no estamos autorizados, la función de geoanalytics.auth_info() nos devolverá un DataFrame vacío.

Con la función geoanalytics.deauth() podremos desautorizar GeoAnalytics Engine sin necesidad de cerrar la sesión de Spark; aunque, al cerrar sesión de Spark, la biblioteca se desautoriza automáticamente.

Carga de datos

Al utilizar la capacidad de analítica de GeoAnalytics Engine sobre un entorno de Spark podemos extender las capacidades de Spark. De forma que, por ejemplo, podemos cargar datos espaciales de nuevas fuentes de datos a parte de los ya integrados por Spark. La mayoría de las fuentes de datos admiten la carga desde un único archivo o desde una carpeta que contenga varios archivos con el mismo formato y esquema. Las fuentes de datos espaciales admitidas suelen permitir tanto la carga de datos como el guardado del mismo, pero en la siguiente tabla podemos ver todos los formatos admitidos y las excepciones:

Fuente de datos	*Formato*	*Cargar*	*Guardar*
CSV	csv	Sí	Sí
Feature service	feature-service	Sí	Sí
File geodatabase	filegdb	Sí	No
GeoJSON	geojson	Sí	Sí
GeoParquet	geoparquet	Sí	Sí
JDBC	jdbc	Sí	Sí
ORC	orc	Sí	Sí
Parquet	parquet	Sí	Sí
Esri shapefile	shapefle	Sí	Sí
Vector tiles	vector-tile	No	Sí

La carga y el guardado de datos es muy sencilla ya que solo tenemos que especificar el formato de la fuente de datos a la hora de la lectura o escritura de los datos y seguir la sintaxis clásica en Spark para ello:

```
## Lectura de Shapefile
df = spark.read.format("shapefile").load("S3://my-data/hurricanes")

## Guardado de datos como shapefile
df.write.format("shapefile").save("hdfs://nn1home:8020/hurricanes")
```

Acceso a contenido privado

Ante de revisar uno a uno cada tipo de datos es interesante revisar cómo podemos acceder a contenido privado alojado en ArcGIS ya que, independientemente del formato, habrá que iniciar sesión si los datos están ahí alojados.

Aunque un alto porcentaje de contenido alojado en ArcGIS es de acceso público, como el que podemos encontrar en el ArcGIS Living Atlas of the World, también hay contenido privado. Para poder leer datos privados o escribir datos en alguna organización deberemos iniciar sesión en ella. GeoAnalytics Engine permite la autenticación registrando una organización de ArcGIS Online o ArcGIS Enterprise con la función register_gis.

register_gis

geoanalytics.register_gis(*name, url='https://arcgis.com', '', username=None, password=None, cert_file=None, cert_password=None, client_id=None, authorization_code=None*)

Registers a GIS with the system. The GIS can be ArcGIS Online or ArcGIS Enterprise.

The name of the GIS can be supplied in place of credentials when accessing resources within the GIS (e.g. feature services).

Parameters:
- **name** (*str*) – a user-defined name that identifies the GIS.
- **url** (*str*) – URL for the GIS (e.g. https://arcgis.com or https://enterprise.example.com/portal). The default is ArcGIS Online.
- **username** (*str*) – username for the GIS.
- **password** (*str*) – password for the GIS.
- **cert_file** (*str*) – Path to certificate file.
- **cert_password** (*str*) – Password for certificate.
- **client_id** (*str*) – Client id used for registering an OAuth 2.0 protected GIS.
- **authorization_code** (*str*) – Authorization code is required when *client_id* is provided. Generate an authorization code through the following URL (replace "<url>" and "<client_id>" with the *url* and *client_id*): <url>/sharing/oauth2/authorize?client_id=<client_id>&response_type=code&redirect_uri=urn:ietf:wg:oauth:2.0:oob.

Si no se especifica el parámetro URL, por defecto, la función register_gis tratará de hacer la conexión con ArcGIS Online por lo que deberemos especificarlo si queremos conectarnos con ArcGIS Enterprise.

```
import geoanalytics

## Conexión a ArcGIS Online
geoanalytics.register_gis("myGIS", username="User", password="p@ssw0rd")

## Conexión a ArcGIS Enterprise
geoanalytics.register_gis("myPortal", "https://example.com/portal", username="User", password="p@ssw0rd")
```

Este tipo de conexiones son únicos, es decir, si queremos trabajar en dos organizaciones diferentes tendremos que desautorizar una conexión antes de realizar otra. Con el método unregister_gis podemos desautorizar la conexión de forma sencilla.

A parte de este método de autorización, también podemos hacerlo a través de una autenticación protegida a nivel web con PKI y con autenticación de usuario con OAuth 2.0.

Autenticación a nivel web con PKI

Para conectarse a un ArcGIS Enterprise protegido con PKI, podemos utilizar un archivo de certificado con formato PKCS12, como .pfx o .p12, junto a la contraseña al iniciar sesión en ArcGIS Enterprise mediante autenticación basada en certificado de cliente.

```
geoanalytics.register_gis("myPortal", "https://example.com/portal",
                          cert_file=r"\\path\to\mycert.pfx", cert_password="p@ssw0rd")
```

Autenticación de usuario con OAuth 2.0

GeoAnalytics Engine admite OAuth 2.0 como método de autenticación y actúa como una aplicación nativa sin servidor cuando se utilizar la autorización OAuth 2.0 con ArcGIS Online o ArcGIS Enterprise. Este modo de autenticación requiere un **client ID** que se puede crear fácilmente desde la organización de ArcGIS Online o ArcGIS Enterprise.

Crear client ID

Los pasos a seguir para crear un client ID son:

1. Iniciar sesión en la organización (ArcGIS Online o ArcGIS Enterprise).
2. Ir a la pestaña de contenido.
3. Hacer click en **Nuevo elemento** y seleccionar **Aplicación.**
4. En la ventana emergente que parece, seleccionar **Otra aplicación.**
5. Dar un título al elemento.
6. Guardar elemento.
7. En la página de detalles del nuevo elemento creado, navegar hasta la sección de **Credenciales** y copiar el Client ID.

Una vez tenemos este código, tan solo tendremos que pasarlo como un parámetro más:

```
geoanalytics.register_gis("myPortal", "https://example.com/portal", client_id="drg5d5r575rfb")
```

Fuentes de datos

CSV

Un archivo CSV (comma-separated values) es un tipo de archivo de texto delimitado por comas u otros caracteres para separar campos. La forma en la que aparecen los datos geográficos puede estar en formato número o texto.

```
## Lectura de un único archivo CSV
#### Opción 1
df = spark.read.option("header", "true").csv("path/to/your/file.csv")
#### Opción 2
df = spark.read.format("csv").option("header", "true").load("path/to/your/file.csv")

## Varios archivos CSV en una carpeta
df = spark.read.option("header", "true").csv("path/to/your/csv/files/*.csv")

## Guardado del df como CSV
#### Opción 1
df.write.option("header", "true").csv("path/to/output/directory")
#### Opción 2
df.write.format("csv").option("header", "true").save("path/to/output/directory")
```

Una vez cargado el o los archivos CSV, podemos utilizar las funciones específicas de GeoAnalytics Engine para transformar los campos con información geográfica en geometrías correctas para poder hacer análisis. Algunas de las diferentes funciones SQL que podemos usar son:

- ST_PointFromText: toma una columna de tipo cadena y devuelve una columna con información del punto.

```
from geoanalytics.sql import functions as ST

df = spark.createDataFrame([('POINT (-72.047 -49.292)',)], ["wkt"])

df.select(ST.point_from_text("wkt", srid=4326).alias("point_from_text")).show(truncate=False)

## Resultado:
+--------------------------+
|point_from_text           |
+--------------------------+
|{"x":-72.047,"y":-49.292}|
+--------------------------+
```

- ST_PolyFromShape: función que toma una columna binaria y devuelve una columna de polígonos.

```
from geoanalytics.sql import functions as ST

poly_shape = bytearray(b'\x05\x00\x00\x00\x00\x00\x00\x00\x00\x00\x02@\x00\x00\x00\x00\x00\x00\.....')

df = spark.createDataFrame([(poly_shape, )], ["shape"])

df.select(ST.poly_from_shape("shape", sr=4326).alias("poly_from_shape")).show(truncate=False)

## Resultado:
+-----------------------------------------+
|poly_from_shape                          |
+-----------------------------------------+
|{"rings":[[[20,30],[10,28],[22,35],[20,30]]]}|
+-----------------------------------------+
```

Feature Service

Los feature services o servicios de entidades son servicios de datos alojados que pueden contar con información espacial o no. En este tipo de servicios, los conjuntos de datos espaciales son **capas de entidades** y los conjuntos de datos no espaciales son **tablas**. Este tipo de servicios nos permiten consultar, editar y analizar los datos que almacenan. Las URLs de estos servicios suelen seguir este esquema:

```
https://<host>/<uniqueID>/ArcGIS/rest/services/<serviceName>/FeatureServer/<Layer or Table ID>
```

Un ejemplo de lectura de un feature service es:

```
spark.read.format("feature-service").option().load(URL)
```

De esta forma estamos leyendo los datos del servicio y cargándolos como Spark DataFrame y así poder usar cualquier operación soportada para DataFrames en GeoAnalytics Engine o Spark.

GeoAnalytics Engine también nos permite guardar datos en este tipo de formato al extender las capacidades de Spark. Hay que tener en cuenta que el tipo *geometry* genérico no es compatible con los feature services por lo que debe seguir el formato correcto de punto, línea o polígono.

```
df.write.format("feature-service").option().save()
```

File Geodatabase

Una geodatabase de archivos o file geodatabase es una colección de archivos de una carpeta del disco que puede almacenar, consultar y administrar datos espaciales y no espaciales. Una vez cargada la file geodatabe en un DataFrame de Spark, podemos ejecutar análisis y visualizar los datos usando las funciones y herramientas de GeoAnalytics Engine y también las expuestas por Spark. Sin embargo, **no** podemos **guardar** datos en este formato desde GeoAnalytics Engine.

Como decimos, una geodatabase de archivos puede tener muchos datos en su interior por lo que deberemos especificar cuáles son los datos que queremos cargar exactamente. La función de lectura de datos nos permite ver el catálogo de datos que hay en una geodatabase de archivos con el parámetro ***gdbPath*** de forma que, al ejecutar ese comando obtendremos un pequeño dataframe con los datos que contiene.

```
# Carga de una file GDB
us_lakes = spark.read.format("filegdb").option("gdbPath", "path/to/your/example.gdb").option("gdbName", "us_lakes").load()

# Resultado
+-------------+-------------+------------+
|         Name|  DatasetType|GeometryType|
+-------------+-------------+------------+
|ca_population|        Table|        null|
|     ca_parks|Feature Class|       Point|
|     us_lakes|Feature Class|     Polygon|
|    us_rivers|Feature Class|    Polyline|
|        cells|Feature Class|  MultiPoint|
+-------------+-------------+------------+
```

Una vez que sabemos el dato concreto que queremos cargar de la geodatabase de archivos podemos especificarlo con el parámetro ***gdbName***:

```
us_lakes = spark.read.format("filegdb").option("gdbPath", "path/to/your/example.gdb").option("gdbName", "us_lakes").load()
```

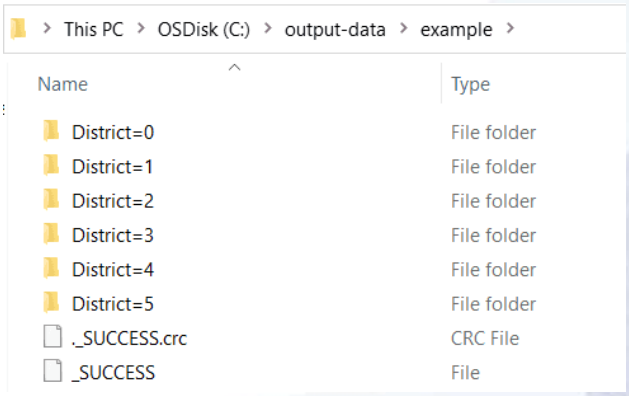
GeoJSON

GeoJSON es un formato abierto estándar para datos espaciales que representa elementos geográficos y sus atributos. Las geometrías que admite son puntos, líneas y polígonos, simples y complejas. Está basado en JavaScript Object Notation (JSON) y utiliza el sistema de referencia de coordenadas geográficas WGS84. Al cargar datos GeoJSON, la columna geometría se crea automáticamente en el DataFrame creado y podemos ejecutar análisis y visualizar los datos con funciones propias de Spark y de GeoAnalytics Engine.

Podemos leer datos en formato GeoJSON especificando el formato; el resto de los parámetros podemos verlos en la documentación de Spark:

```
df = spark.read.format("geojson").load("path/to/your/file.geojson")
```

Además de leer archivos concretos, Spark puede inferir columnas a partir de directorios que siguen el formato *columna=valor*. De forma que, si los archivos GeoJSON están organizados en subdirectorios llamados Distrito=0, Distrito=1, etc., Spark utilizará los nombres de las subcarpetas para crear un único DataFrame a partir de todos los archivos.




```
df = spark.read.format("geojson").load("path/to/your/files/*.geojson")
```

Una vez procesados los datos o transformados desde otra fuente de datos, podemos guardarlos en formato GeoJSON. Por defecto, GeoJSON usa WGS84 (SRID:4326) por lo que hay que tener en cuenta que, si la referencia espacial es diferente, lo transformará automáticamente. GeoAnalytics Engine

permite la opción de guardado de datos en este formato con una referencia espacial diferente usando el parámetro *customCrs*.

```
df.write.format("geojson").save("path/to/output/directory")
```

GeoParquet

GeoParquet es un formato de almacenamiento de código abierto estandarizado que amplía Apache Parquet al definir cómo deben almacenarse los datos geoespaciales, incluida la representación de geometrías y los metadatos. La estructura de GeoParquet permite la interoperabilidad entre cualquier sistema que lea o escriba datos espaciales en formato Parquet.

Podemos leer datos en este formato:

```
df = spark.read.format("geoparquet").load("path/to/your/file.parquet")
```

Y guardarlos:

```
df.write.format("geoparquet").option("version", "1.0.0").save("path/to/output/directory")
```

JDBC

Apache Spark admite de forma nativa la lectura y escritura de datos directamente desde y hacia varios tipos diferentes de bases de datos gracias a JDBC o Java Database Connectivity. JDBC de Spark lee y escribe datos utilizando directamente Spark DataFrames, por lo que, GeoAnalytics Engine puede leer y escribir datos en cualquiera de las bases de datos compatibles con Apache Spark como Microsoft SQL Server, Oracle o PostgreSQL. Cada tipo de base de datos requiere un controlador JDBC específico ya que son diferentes para cada tipo de base de datos.

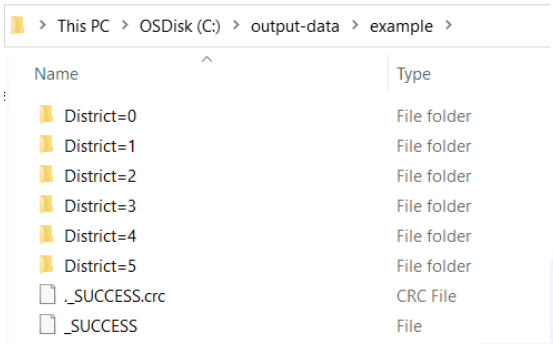
```
# Ejemplo de lectura de datos en PostgreSQL
df = spark.read \
          .format("jdbc") \
          .option("url", "jdbc:postgresql://localhost:5432/sample") \
          .option("dbtable", "locations") \
          .option("user", "user1") \
          .option("password", "test123") \
          .option("driver", "org.postgresql.Driver") \
          .load()

# Ejemplo de escritura de datos en PostgreSQL
df.write \
  .format("jdbc") \
  .option("url", "jdbc:sqlserver://localhost\demoInstance;databaseName=sampleTwo") \
  .option("dbtable", "places") \
  .option("user", "username2") \
  .option("password", "p@ssw0rd") \
  .option("driver", "com.microsoft.sqlserver.jdbc.SQLServerDriver") \
  .save()
```

ORC

Apache ORC (Optimized Row Columnar) es un formato de almacenamiento de datos orientado a columnas, gratuito y de código abierto del ecosistema Apache Hadoop. Este formato de datos es autodescriptivo e integra soporte para encontrar rápidamente las filas necesarias. Gracias a esto, los datos en formato ORC se leen en menos tiempo y reduce el tamaño de los datos en disco. Además, ORC admite tipos de datos complejos como estructuras, listas, mapas y uniones. ORC es compatible de forma nativa con Spark y Hive. Una vez cargados los datos, podremos analizarlos y visualizarlos con las funciones propias de Spark y las incluidas en GeoAnalytics Engine y también podremos guardar los datos en este formato.

```
# Lectura de datos ORC en un único archivo
## Opción 1
df = spark.read.orc("path/to/your/file.orc")
## Opción 2
df = spark.read.format("orc").load("path/to/your/file.orc")
```

Al igual que con los archivos en formato GeoJSON, Spark puede inferir la partición de columnas a partir de varias subcarpetas en un único directorio con el formato *columna=valor*. Spark se encargará de crear una columna con el valor del nombre de cada carpeta.




```
# Lectura de datos ORC en varios archivos
df = spark.read.orc("path/to/your/files/*.orc")
```

También hay varias opciones de escritura de datos:

```
# Opción 1
df.write.orc("path/to/output/directory")
# Opción 2
df.write.format("orc").save("path/to/output/directory")
```

Parquet

Apache Parquet (.parquet) es un formato de almacenamiento de datos en columnas de código abierto que reconoce los tipos y puede almacenar datos anidados en un formato de columnas plano. Los datos Parquet pueden ser almacenados en un sistema de archivos distribuidos como HDFS, almacenamiento en la nube, en local o cualquier otra localización accesible para Spark.

Tras cargar todos los archivos Parquet como un DataFrame de Spark, podemos crear una columna de geometría y definir su referencia espacial con las funciones SQL de GeoAnalytics Engine. Este paso es

imprescindible para poder ejecutar análisis y visualizar datos con las funciones y herramientas de GeoAnalytics Engine. Este tipo de formatos nos permite leer y escribir en él.

```
## Lectura de datos en formato parquet en un único archivo
# Opción 1
df = spark.read.parquet("path/to/your/file.parquet")
# Opción 2
df = spark.read.format("parquet").load("path/to/your/file.parquet")

## Lectura de datos en formato parquet en varios archivos
df = spark.read.parquet("path/to/your/files/*.parquet")


## Guardado de datos en formato parquet
# Opción 1
df.write.parquet("path/to/output/directory")
# Opción 2
df.write.format("parquet").save("path/to/output/directory")
```

Shapefile

El formato Esri Shapefile (shp) es un formato vectorial de almacenamiento digital donde se guarda la localización de los elementos geográficos y los atributos asociados a ellos. Los datos shapefiles se pueden almacenar en un sistema de archivos distribuidos como HDFS, en un almacenamiento en la nube o un directorio local. Al cargar los datos, automáticamente se crea una columna de geometría en el DataFrame resultante y se establecerá su referencia espacial. El formato shapefile admite colecciones de puntos, líneas, polígonos y multipartes de los mismos. Tras haber cargado los datos como DataFrame de Spark, se pueden empezar a ejecutar análisis y visualizar los datos con funciones de Spark y de GeoAnalytics Engine. Una vez que guarde un DataFrame como archivo shapefile, podemos almacenar los archivos o acceder a ellos y visualizarlos a través de otros sistemas.

```
## Lectura de datos en formato shapefile
df = spark.read.format("shapefile").load("path/to/your/file")

## Escritura de datos en formato shapefile
df.write.format("shapefile").save("path/to/output/directory")
```

Vector tiles

Los datos en formato vector tiles contienen representaciones vectoriales de datos en una amplia gama de escalas y se pueden utilizar para visualizar geometrías en un DataFrame de Spark. Aunque no podemos leer datos en este formato, si que podemos guardarlos y que sean posteriormente visualizados desde otros programas como ArcGIS Pro o Mapbox.

```
df.write.format("vector-tile") \
                .option("layerName", "earthquakes") \
                .option("maxLevel", 12) \
                .option("maxPointsPerTile", 10000) \
                .option("prioField", "magnitude") \
                .option("attributes", "magnitude") \
                .save(r"path/to/output/earthquakes")
```

Análisis espacial

Una vez cargados los datos como un DataFrame de Spark ya podemos empezar a hacer análisis espaciales gracias a las funciones y herramientas que proporciona GeoAnalytics Engine. En general, podemos hablar de grandes capacidades analíticas que nos proporciona la biblioteca a través de diferentes herramientas y funciones. Esas capacidades son:

- **Análisis de patrones.** Teniendo en cuenta las variables espaciales podemos analizar patrones para así hacer cálculos de densidades, localizar puntos calientes y fríos, agrupar los datos en función de diferentes criterios, clasificarlo y hacer regresiones.
- **Resumir los datos.** Cuando trabajamos con grandes cantidades de datos, la simplificación de los datos es crucial para poder manejar y entender mejor las variables. Hay muchas formas de resumir ya sea por agregación de puntos, construir mallas multivariables, crear cubos espacio-temporales, reconstruir rutas o resumir atributos.
- **Gestión de datos.** Las herramientas para ellos nos permiten añadir nuevos datos, calcular nuevas variables, seleccionar los datos de estudio, disolver límites, unir capas o superponerlas.
- **Uso de la proximidad.** Podremos hacer análisis como encontrar los vecinos más cercanos, crear buffers o agrupar en función de la cercanía.
- **Encontrar localizaciones.** De forma que podamos detectar comportamientos anómalos, encontrar ubicaciones similares o geocodificar lugares.
- **Enriquecimiento de datos.** Permite complementar los datos existentes mediante el cálculo de estadísticas y variables adicionales de contexto.

Aunque hemos estado hablando indistintamente **de funciones SQL y herramientas** ya que ambas nos permiten hacer análisis espaciales, hay que tener en cuenta sus diferencias. La más importante es con el número de columnas que trabajan cada una de ellas, las funciones SQL utilizan una o dos columnas, mientras que las herramientas pueden llegar a utilizar todas las del DataFrame.

Funciones SQL

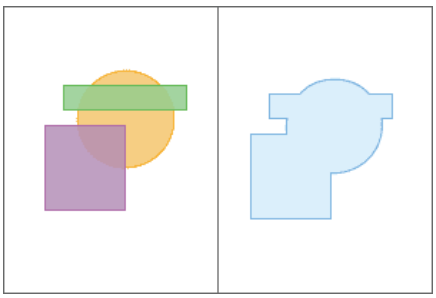
Hay más de 150 funciones disponibles en la biblioteca y en cada versión se van añadiendo más, por lo que, es importante repasar la documentación en cada nueva versión. A continuación, haremos un repaso de algunas de las funciones que más se usan:

ST_Area

A partir de una columna de geometría, devuelve una columna con el valor del área calculado. Las unidades que se usan para el cálculo del área son las mismas que las de las geometrías de entrada. Por lo tanto, es muy importante saber qué unidad de medida utilizar la referencia espacial. Si nuestras geometrías están en un sistema de coordenadas geográfico es más recomendable utilizar la función ST_GeodesicArea para calcular el área.

```
from geoanalytics.sql import functions as ST, Polygon

data = [
    (Polygon([[[0,0],[0,10],[10,10],[10,0],[0,0]]]),),
    (Polygon([[[20,0],[30,20],[40,0],[20,0]]]),),
    (Polygon([[[20,30],[25,35],[30,30],[20,30]]]),)
]
df = spark.createDataFrame(data, ["polygon"])

df.select(ST.area("polygon").alias("st_area")).show(truncate=False)

## Resultado
+-------+
|st_area|
+-------+
|100.0  |
|200.0  |
|25.0   |
+-------+
```

ST_Union



Esta función toma dos o más geometrías y devuelve una única en una columna de tipo geométrico nueva. Para poder hacer correctamente la unión, el tipo de geometrías que se quieren unir debe ser el mismo.

```
from geoanalytics.sql import functions as ST, Point, Polygon

df = spark.createDataFrame([(Point(0,0), Polygon([[[0,0],[10,0],[10,10],[0,10]]]),)], ["point", "polygon"])

df.select(ST.union(ST.buffer("point", 5), "polygon")).st.plot()
```

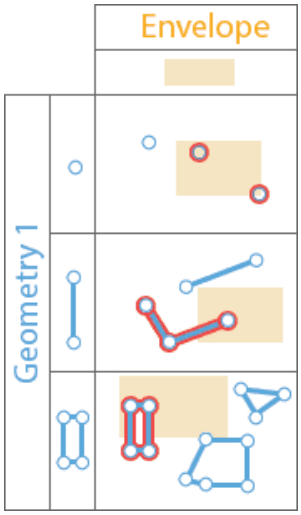
ST_SRID

Funciona tanto para obtener como para definir la referencia espacial, dependerá de los parámetros. Hay que tener en cuenta que está funciona define la referencia espacial pero no transforma las coordenadas si están en otro sistema, para ello habría que utilizar ST_Transform.

```
# Definir referencia espacial
from geoanalytics.sql import functions as ST, Point

df = spark.createDataFrame([(Point(-178, -17),)], ["point"])
df.withColumn("srid", ST.srid("point")).show()
df.select(ST.srid("point", 4326).alias("point")).withColumn("srid", ST.srid("point")).show()

## Resultado
+------------------+----+
|             point|srid|
+------------------+----+
|{"x":-178,"y":-17}|   0|
+------------------+----+

+------------------+----+
|             point|srid|
+------------------+----+
|{"x":-178,"y":-17}|4326|
+------------------+----+
```

```
# Obtener referencia espacial

from geoanalytics.sql import functions as ST

df = spark.createDataFrame([("POINT (-2533858.73 8107527.81)",)], ["wkt"])\
          .select(ST.point_from_text("wkt", srid=54008).alias("geometry"))
df.select(ST.srid("geometry").alias("srid")).show()


## Resultado
+-----+
| srid|
+-----+
|54008|
+-----+
```

ST_BboxIntersects



Esta función toma una geometría y las coordenadas que definen la *bounding box* y devuelve un valor booleano en función de si la geometría interseca con ella o no.

```
from geoanalytics.sql import functions as ST

data = [
    ("POINT (10 30)",),
    ("MULTIPOINT (0 0, 5 5, 0 5)", ),
    ("LINESTRING (15 15, 10 15, 12 2)", ),
    ("POLYGON ((20 30, 10 20, 22 35, 40 20))", )
]

df = spark.createDataFrame(data, ["wkt"])\
        .select(ST.geom_from_text("wkt").alias("geometry"))

df.select(ST.bbox_intersects("geometry", xmin=0.0, ymin=0.0, xmax=15.0, ymax=15.0).alias("bbox_intersects")) \
  .show()


### Resultado
+---------------+
|bbox_intersects|
+---------------+
|          false|
|           true|
|           true|
|          false|
+---------------+
```

Herramientas

Las diferentes herramientas incluidas en GeoAnalytics Engine nos permiten gestionar, enriquecer, resumir y analizar conjuntos de datos completos. Al contrario que las funciones SQL que operan fila a fila, las herramientas utilizan todas las filas para calcular el resultado si así lo necesitan.

Cada herramienta está representada como una clase con métodos *setter* para configurar los parámetros de la herramienta. También cuentan con un método de ejecución *run* con el que llamar al DataFrame y obtener el resultado, en formato DataFrame o como tupla.

La mayoría de las herramientas necesitan que los DataFrame sobre los que van a utilizarse tengan una columna de geometría. El tipo de la geometría variará en función de la herramienta que se quiera ejecutar. En general, cada DataFrame tendrá un único campo geométrico pero si nuestros datos tuvieran más de uno, podemos especificar cuál es el primario con el método *set_geometry_field* y será sobre el que se realicen los análisis.

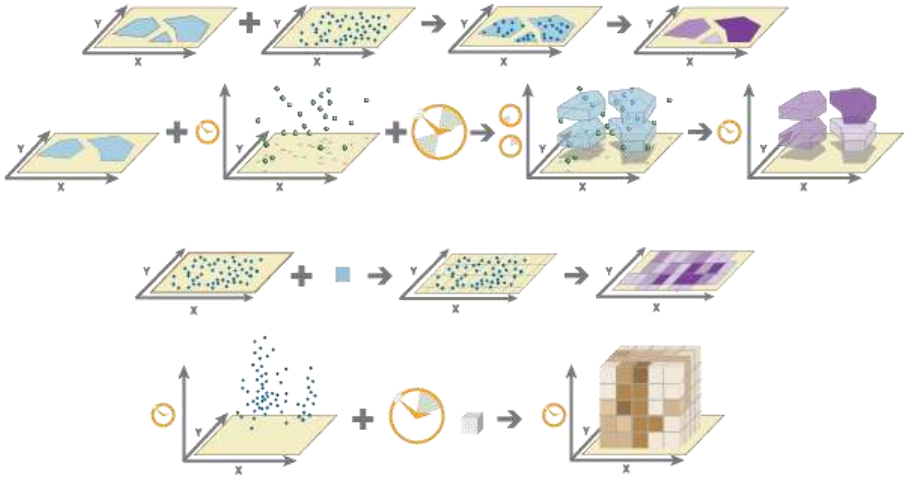
Hay algunas herramientas que utilizan también la variable temporal, GeoAnalytics Engine puede trabajar con dos tipos de tiempo:

- Instantes. Un momento único representando en el tiempo representado con una única columna de tipo *timestamp*.
- Intervalos. Una duración de tiempo representado con una columna de inicio y otra de fin.

Si una herramienta requiere información temporal y el DataFrame contiene una única columna de tipo *timestamp*, esta se utilizará automáticamente como el campo de tiempo instantáneo del DataFrame. Si existen varias columnas de tipo *timestamp*, será necesario especificar explícitamente una como campo de tiempo instantáneo, o bien dos como campos de tiempo de intervalo, mediante la función *set_time_fields*.

Hay más de 25 herramientas, vamos a revisar algunas de las más utilizadas:

Agregación de puntos



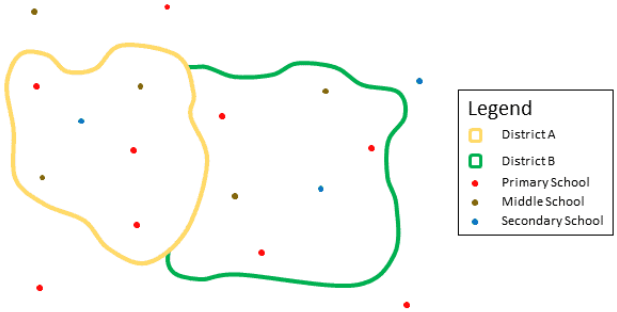
Esta herramienta nos permite resumir puntos dentro de polígonos o celdas. El objetivo principal es simplificar grandes volúmenes de datos de puntos y obtener estadísticas que faciliten el análisis y la interpretación de la información espacial.

La herramienta *Aggreate Points* funciona asignando cada punto a un polígono de agregación y calculando estadísticas sobre los atributos de los puntos contenidos en cada área. Las áreas de agregación pueden proceder de una capa de polígonos existente o generarse dinámicamente mediante una malla regular (por ejemplo, cuadrados o hexágonos) de un tamaño especificado.

La siguiente imagen muestra un ejemplo de uso, en ella tenemos dos polígonos que representan dos distritos en cuyo interior hay varios tipos de centro educativos. Si ejecutamos la herramienta de agregación de puntos obtendremos las estadísticas (conteo, suma, mínimo, máximo, media, varianza y desviación estándar) de toda la población que acude a dichos centros en función del distrito.

Hay que tener en cuenta que está **herramienta necesita coordenadas proyectadas**, es decir, puede que tengamos que transformar nuestros datos. Recuerda revisar las notas de uso.

- Analysis with hexagonal or square bins requires that your input DataFrame's geometry has a projected coordinate system. If your data is not in a projected coordinate system, the tool will



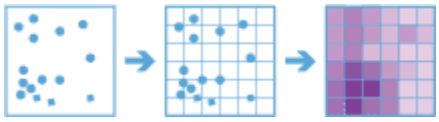
ObjectID	District	Type	Population
1	A	Primary School	280
2	A	Primary School	408
3	A	Primary School	356
4	A	Middle School	361
5	A	Middle School	450
6	A	Secondary School	713
7	B	Primary School	370
8	B	Primary School	422
9	B	Primary School	495
10	B	Middle School	607
11	B	Middle School	574
12	B	Secondary School	932

Para ejecutar esta herramienta, deberemos seguir un flujo clásico:

- Cargar las herramientas y funciones necesarias.
- Cargar los datos en un DataFrame de Spark.
- Definir y ejecutar la herramienta.
- Y, como recomendación, mostrar los resultados en formato tabla.

```
# Carga de funciones y herramientas
from geoanalytics.tools import AggregatePoints
from geoanalytics.sql import functions as ST
from pyspark.sql import functions as F

# Ruta a los datos
data_path = r"https://sampleserver6.arcgisonline.com/arcgis/rest/services/" \
    "Earthquakes_Since1970/FeatureServer/0"

# Carga de los datos como DataFrame de Spark y transformación a la referencia espacial 54034
df = spark.read.format("feature-service").load(data_path) \
                    .withColumn("shape", ST.transform("shape", 54034))

# Uso de la herramienta de Agregación de puntos para contar los terremotos de los últimos 10
años en celdas hexagonales de 800km de lado a lado (apotema doble). Obteniene estadísticas de
magnitudes mínimas y máximas
result = AggregatePoints() \
            .setBins(bin_size=800, bin_size_unit="Kilometers", bin_type="Hexagon") \
            .setTimeStep(interval_duration=10, interval_unit="years") \
            .addSummaryField(summary_field="magnitude", statistic="Min") \
            .addSummaryField(summary_field="magnitude", statistic="Max") \
            .run(df)

# Muestra los primeros cinco resultados
result = result.select("COUNT", "bin_geometry", "MIN_magnitude", "MAX_magnitude",
                       F.date_format("step_start", "yyyy-MM-dd").alias("step_start"),
                       F.date_format("step_end", "yyyy-MM-dd").alias("step_end"))
result.sort("COUNT", "MIN_magnitude", ascending=False).show(5)


##### RESULTADO
+-----+--------------------+-------------+-------------+----------+----------+
|COUNT|        bin_geometry|MIN_magnitude|MAX_magnitude|step_start|  step_end|
+-----+--------------------+-------------+-------------+----------+----------+
| 18.0|{"rings":[[[78519...|          4.7|          7.4|1999-12-31|2009-12-31|
| 15.0|{"rings":[[[85447...|          4.6|          7.5|1989-12-31|1999-12-31|
| 15.0|{"rings":[[[57735...|          4.5|          6.1|1999-12-31|2009-12-31|
| 14.0|{"rings":[[[30022...|          4.0|          7.5|1979-12-31|1989-12-31|
| 13.0|{"rings":[[[1.616...|          5.2|          7.2|1999-12-31|2009-12-31|
+-----+--------------------+-------------+-------------+----------+----------+
only showing top 5 rows
```

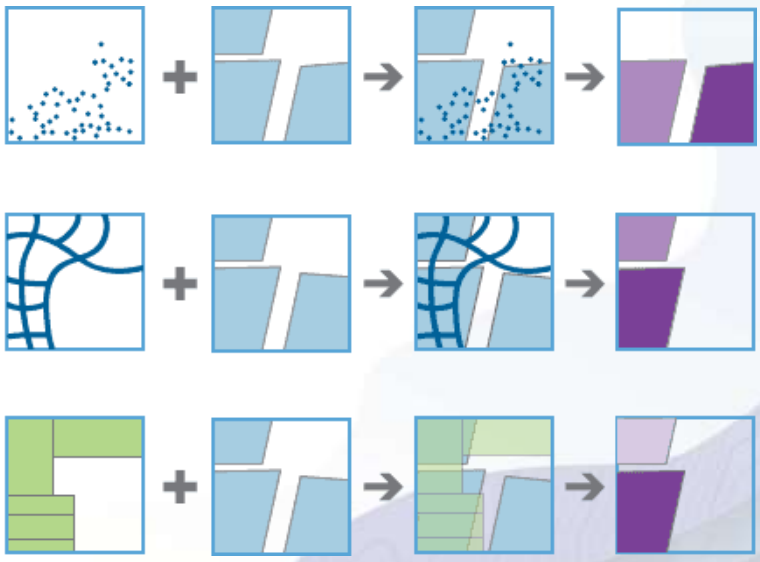
Cálculo de densidad



Esta herramienta nos permite calcular la densidad espacial de puntos sobre un área de análisis determinada. El objetivo principal es identificar zonas con mayor o menor concentración de elementos, transformando datos discretos en continuos. *Calculate density* funciona evaluando la cantidad de entidades que se encuentran dentro de un radio de búsqueda definido alrededor de cada ubicación.

```
# Carga de funciones y herramientas
from geoanalytics.tools import CalculateDensity
from geoanalytics.sql import functions as ST
from pyspark.sql import functions as F

# Ruta a los datos
data_path = r"https://sampleserver6.arcgisonline.com/arcgis/rest/services/" \
            "Earthquakes_Since1970/FeatureServer/0"

# Carga de los datos como DataFrame y transformación a la referencia espacial 2100
earthquakes_df = spark.read.format("feature-service").load(data_path) \
                              .withColumn("shape", ST.transform("shape", 2100))

# Creación de un DataFrame para los terremotos cercanos a Grecia usando ST_EnvIntersects
greece_earthquakes_df = earthquakes_df.where(ST.bbox_intersects("shape",
                                                                xmin=100000,
                                                                xmax=1100000,
                                                                ymin=3500000,
                                                                ymax=4800000))

# Uso de la herramienta para encontrar áreas con una alta densidad de terremotos. Como
parámetros usa celdas cuadradas de 10millas de lado y vecinos a 50millas
result = CalculateDensity() \
            .setWeightType(weight_type="Uniform") \
            .setBins(bin_size=10, bin_size_unit="Miles", bin_type="Square") \
            .setNeighborhood(distance=50, distance_unit="Miles") \
            .setAreaUnit(area_unit="SquareMiles") \
            .run(dataframe=greece_earthquakes_df)

# Muestra los 5 primeros resultados
result.select("bin_geometry", F.round("density",9).alias("density")).sort("density",
ascending=False).show(5)

## RESULTADO
+--------------------+-----------+
|        bin_geometry|    density|
+--------------------+-----------+
|{"rings":[[[28968...|0.001234568|
|{"rings":[[[22530...|0.001234568|
|{"rings":[[[22530...|0.001234568|
|{"rings":[[[22530...|0.001234568|
|{"rings":[[[22530...|0.001234568|
+--------------------+-----------+
only showing top 5 rows
```

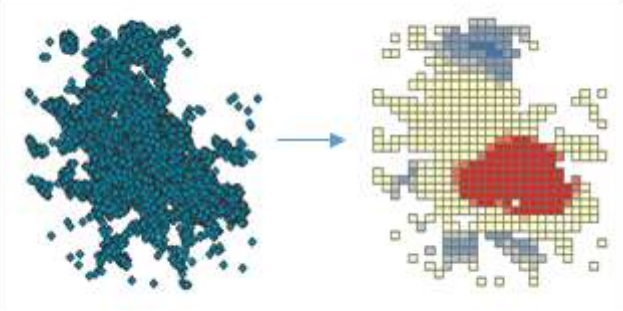
Resumir dentro de



La herramienta SummarizeWithin calcula estadísticas sobre entidades contenidas o superpuestas dentro de límites definidos. Los límites pueden definirse mediante una columna de geometría poligonal o a partir de teselas regulares como *bins* hexagonales o cuadrados. Las estadísticas que devolverá para cada área son conteo de entidades, suma, promedio, valores mínimos y máximos, desviación estándar, varianza, etc.

```
# Carga de bibliotecas
from geoanalytics.tools import SummarizeWithin
from geoanalytics.tools import ReconstructTracks
from geoanalytics.sql import functions as ST
from pyspark.sql import functions as F

# Ruta a los datos de huracanes
hurricanes_data_path = r"https://services2.arcgis.com/FiaPA4ga0iQKduv3/arcgis/rest/" \
            "services/IBTrACS_ALL_list_v04r00_points_1/FeatureServer/0"

# Cargar los huracanes como DataFrme pero solo aquellos que están dentro del bounding box (ST.bbox_intersects)
hurricanes_df = spark.read.format("feature-service").load(hurricanes_data_path) \
    .withColumn("bbox_intersects", ST.bbox_intersects("shape",-10512137.72, -9527997.38,3278846.39,4303954.46)) \
    .where("bbox_intersects == 'true'") \
    .where("BASIN == 'NA'")

# Reconstruir ruta seguida por los huracanes
rt_result = ReconstructTracks() \
            .setTrackFields("NAME") \
            .setDistanceMethod(distance_method="Planar") \
            .run(dataframe=hurricanes_df)

# Usar Summarize Within para resumir las rutas de huracanes dentro de cada bin y crear un mapa de calor
## Las celdas creadas son hexagonales de 200km de lado
result = SummarizeWithin() \
           .setSummaryBins(bin_size=200, bin_size_unit="Kilometers", bin_type='hexagon') \
           .includeShapeSummary(include=True, units="Kilometers") \
           .run(dataframe=rt_result)

# Mostrar los 5 primeros resultados con estadísticas de:
## - Count: número de trayectorias que intersecan en cada bin
## - Sum_lenght_kilometers: longitud total acumulada delas trayectorias
result.output.select("COUNT", F.round("sum_length_kilometers",9).alias("sum_length_kilometers")) \
             .sort("COUNT", "sum_length_kilometers", ascending=False).show(5)

###### RESULTADO
+-----+---------------------+
|COUNT|sum_length_kilometers|
+-----+---------------------+
| 72.0|      48134.767914587|
| 68.0|      52540.066478513|
| 68.0|      47582.974239464|
| 68.0|      46936.688568699|
| 68.0|      43070.032315087|
+-----+---------------------+
only showing top 5 rows
```

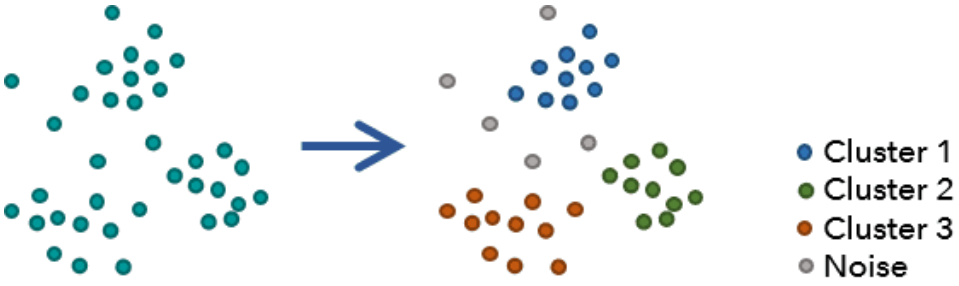
Encontrar puntos calientes



A partir de un DataFrame, la herramienta *Find hot spots* identifica estadísticamente puntos calientes y fríos usando la estadítica de Getis-Ord Gi*. Este estadístico nos informa sobre una autocorrelación espacial, es decir, evalúa si los valores altos (puntos calientes) o bajos (puntos fríos) están concentrados espacialmente en torno a una entidad de lo que podría esperar por azar. Más en detalle, esta herramienta agrupa las geometrías de entrada en bins de un tamaño definido y genera un DataFrame con valores de z-score, p-value y nivel de confianza (Gi_Bin) para cada celda. Este análisis requiere que los bins tengan variabilidad en los conteos, ya que evalúa si los patrones observados difieren de lo que se esperaría bajo una distribución aleatoria. Los valores altos y positivos de z-score indican un clustering intenso; mientras que los z-scores negativos indican ausencia.

```
# Carga de bibliotecas
from geoanalytics.tools import FindHotSpots
from geoanalytics.sql import functions as ST

# Ruta a los datos de terremotos
data_path = r"https://sampleserver6.arcgisonline.com/arcgis/rest/services/" \
            "Earthquakes_Since1970/FeatureServer/0"

# Creación del DataFrame y transformación de la geometría a 54034
df = spark.read.format("feature-service").load(data_path) \
                  .withColumn("shape", ST.transform("shape", 54034))

# FindHotSpots para encontrar puntos calientes a nivel global
result = FindHotSpots() \
            .setBins(bin_size=250, bin_size_unit="Miles") \
            .setNeighborhood(distance=500, distance_unit="Miles") \
            .run(dataframe=df)

# Mostrar los 5 primeros resultados
result.sort("Gi_Bin", ascending=False).show(5)


## RESULTADOS
+-----+------------------+--------------------+------+--------------------+
|value|          GiZScore|           GiPValue|Gi_Bin|        bin_geometry|
+-----+------------------+--------------------+------+--------------------+
|  5.0|3.7607595687467907|1.693981921161063...|   3.0|{"rings":[[[15601...|
| 17.0|  3.83909573966328|1.234882743127410...|   3.0|{"rings":[[[23648...|
|  6.0|  2.66405317591594|0.007720535726935918|   3.0|{"rings":[[[15601...|
|  9.0| 3.785564505436045|1.533600453779985...|   3.0|{"rings":[[[19624...|
| 19.0| 4.102294557788669|4.890730293773235...|   3.0|{"rings":[[[19624...|
+-----+------------------+--------------------+------+--------------------+
only showing top 5 rows
```

Encontrar agrupaciones de puntos



La herramienta Find Point Cluster identifica grupos de puntos (clusters) del ruido circundante según su distribución espacial o espacio-temporal. Los datos de entrada deben ser puntos y estar en un sistema de coordenadas proyectadas. Hay dos métodos de clustering:

- DBSCAN (Density-Based Spatial Clustering of Applications with Noise): detecta clusters basados en una distancia fija de búsqueda. Este método es muy sensible a la elección de la distancia y funciona muy bien con cluster compactos y bien separados.
- HDBSCAN (Hierarchical DBSCAN): este método es más avanzado ya que no usa una distancia fija, sino que se adapta a la densidad de los datos.

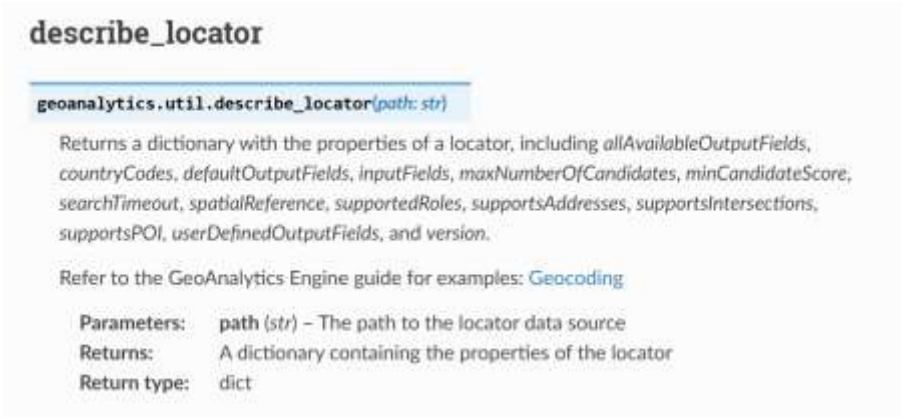
```
# Carga de bibliotecas
from geoanalytics.tools import FindPointClusters
from geoanalytics.sql import functions as ST
from pyspark.sql import functions as F

# Ruta a los terremotos
data_path = r"https://sampleserver6.arcgisonline.com/arcgis/rest/services/" \
            "Earthquakes_Since1970/FeatureServer/0"

# Creación del DataFrame y transformación de la geometría a 54034
df = spark.read.format("feature-service").load(data_path) \
                      .withColumn("shape", ST.transform("shape", 54034))

# Usar FindPointCluster para encontrar los terremotos a escala global
result = FindPointClusters() \
           .setClusterMethod(cluster_method="HDBSCAN") \
           .setMinPointsCluster(min_points_cluster=7) \
           .run(dataframe=df)

# Mostrar los 5 primeros resultados agrupados por el ID del cluster creado
result.filter(result["CLUSTER_ID"] == 115) \
      .select("name", F.date_format("date_", "yyyy-MM-dd").alias("date_"), "CLUSTER_ID", "PROB", "STABILITY") \
      .sort("name", "date_", ascending=False).show(5, truncate=False)

### RESULTADO
+----------------------------+----------+----------+-------------------+-------------------+
|name                        |date_     |CLUSTER_ID|PROB               |STABILITY          |
+----------------------------+----------+----------+-------------------+-------------------+
|TAJIKISTAN: SHURAB, NEFTEABAD|1980-07-10|115       |0.188143468626724  |0.25173588399009866|
|TAJIKISTAN: SHARORA, GISSAR  |1989-01-21|115       |0.14929508441065453|0.25173588399009866|
|TAJIKISTAN: ROSHTKALA, KHOROG|1988-09-24|115       |0.17619107744283952|0.25173588399009866|
|TAJIKISTAN: ROGHUN           |2002-02-02|115       |0.18251933904618436|0.25173588399009866|
|TAJIKISTAN: ROGHUN           |2002-01-08|115       |0.18251933904618436|0.25173588399009866|
+----------------------------+----------+----------+-------------------+-------------------+
only showing top 5 rows
```

Geocodificación

La **geocodificación** es el proceso por el cual podemos convertir una dirección en unas coordenadas para poder localizar el punto exacto. El proceso contrario por el que obtenemos una dirección a partir de unas coordenadas se conoce como **geocodificación inversa**. Para ambos procesos también es necesario un **localizador** o *locator* ya que contendrá los datos de referencia que se utilizan para la geocodificación de las direcciones o de las coordenadas. Los localizadores podremos encontrarlo principalmente en dos

formatos distintos: en *.mmpk (Mobile Map Package)* o en *.loc* y *.loz*. Para poder utilizarlo tendremos que cargarlo con el método *.setLocator()* dentro de la clase que queramos usar.

Para poder utilizar correctamente un localizador debemos conocer sus propiedades, de esta forma podremos configurar correctamente los parámetros de las herramientas y comprender mejor los resultados obtenidos. Podemos utilizar la función *describe_locator(path)* para ver las propiedades del localizador.



Las propiedades que nos devolverá la función son:

- **allAvailableOutputFields:** todos los campos disponibles en este localizador que nos devuelven las herramientas de geocodificación y geocodificación inversa.
- **countryCode:** listado de códigos de países que admite el localizador.
- **defaultOutputFields:** los campos que devuelve el localizador.
- **inputFields:** lista de campos que se usan para encontrar el mejor candidato.
- **maxNumberOfCandidates:** número de máximo de candidatos que devuelve la herramienta de geocodificación.
- **minCadidateScore:** puntuación mínima para una dirección ser considerada candidata en la herramienta de geocodificación.
- **searchTimeout:** valor entero en milisegundos que establece el tiempo de espera para buscar una coincidencia candidata. La herramienta de geocodificación devuelve como resultado “sin coincidencias” en aquellos registros que no encuentran una coincidencia dentro del tiempo de espera establecido.
- **spatialReference:** identificador de la referencia espacial que utiliza el localizador.
- **supportedRoles:** lista de tipos de direcciones compatibles con el localizador. Por ejemplo, direcciones, códigos postales, POIs, etc.
- **supportsAddresses:** booleano que indica si el localizador admite la geocodificación de direcciones.
- **supportsIntersections:** booleano que indica si el localizador admite la geocodificación de intersecciones.
- **supportsPOI:** booleano que indica si el localizador admite la geocodificación de puntos de interés.
- **userDefinedOutputFields:** lista de campos de salida definida por el usuario que devuelven las herramientas de geocodificación y geocodificación inversa.

- **version:** versión de localizador.

```
import geoanalytics
import json

locator_info = geoanalytics.util.describe_locator(r"/data/example.loc")
print(json.dumps(locator_info, sort_keys=True, indent=4))

#### Resultados:
{
    "allAvailableOutputFields": [
        "Status",
        "Score",
        "Match_addr",
        "LongLabel",
        "ShortLabel",
        "Addr_type",
        ...
    ],
    "countryCodes": [
        "CAN",
        "VIR",
        "UMI",
        "MEX",
        "USA",
        ...
    ],
    "defaultOutputFields": [
        "Status",
        "Score",
        "Match_addr",
        "Addr_type"
    ],
    "inputFields": [
        "Address",
        "Address2",
        "Address3",
        "Neighborhood",
        "City",
        "Subregion",
        "Region",
        "Postal",
        "PostalExt",
        "CountryCode"
    ],
    "maxNumberOfCandidates": 50,
    "minCandidateScore": 70,
    "searchTimeout": 3000,
    "path": "C:\\data\\example.loc",
    "spatialReference": {
        "wkid": 4326
    },
    "supportedRoles": [
        "Subaddress",
        "PointAddress",
        "StreetAddress",
        "DistanceMarker",
        "StreetName",
        "StreetInt",
        "Postal",
        "Locality",
        "POI"
    ],
    "supportsAddresses": True,
    "supportsIntersections": True,
    "supportsPOI": True,
    "userDefinedOutputFields": [],
    "version": "903619 [GDM2023Q1SMP, 2023-04-09_04-42-50] (10)"
}
```

Al definir la herramienta **Geocode** para la geocodificación, es necesario configurar sus parámetros de acuerdo con los valores y la estructura del **localizador** utilizado. Por este motivo, el uso de esta herramienta puede variar en función del localizador y de las características de los datos de entrada.

```
# Carga de funciones y herramientas
from geoanalytics.tools import Geocode
from geoanalytics.sql import functions as ST

# Ruta a los datos
data_url = r"https://services1.arcgis.com/Ua5sjt3LWTPigjyD/arcgis/rest/services/" \
    "Public_School_Location_201819/FeatureServer/0"

# Carga de los datos como DataFrame y transformación a la referencia espacial 6423 y filtrado de lo
datos de California
df = spark.read.format("feature-service").load(data_url) \
                  .withColumn("shape", ST.transform("shape", 6423))\
                  .where("STATE='CA'")\
                  .select("NAME","STREET","CITY","STATE","ZIP","shape")

# Acceso al localizador, debe ser accesible para la máquina donde se ejecutar la herramienta
north_america_locator = r"/data/NA_locator.loc"

# Uso de la geocodificación para convertir direcciones en localizaciones usando los campos: name,
street, city, state y zip e indicando que se trata de direcciones en EEUU
result = Geocode() \
           .setLocator(north_america_locator) \
           .setAddressFields("NAME", "STREET", "CITY", "STATE", "ZIP") \
           .setMinScore(80)\
           .setOutFields("all") \
           .setCountryCode("USA")\
           .run(df)

# Mostrar los 5 primeros resultados
result.select("NAME", "STREET", "Score", "Status", "ZIP", "geocode_location").show(5)


### RESULTADO
+--------------------+--------------------+-----------------+------+-----+--------------------+
|                NAME|              STREET|            Score|Status|  ZIP|    geocode_location|
+--------------------+--------------------+-----------------+------+-----+--------------------+
|        Vasquez High|33630 Red Rover M...|97.38095238095238|     M|93510|{"x":-118.2164715...|
|Meadowlark Elemen...|3015 W. Sacrament...|96.17885365609683|     M|93510|{"x":-118.1857999...|
|         High Desert|3620 Antelope Woo...| 97.5609756097561|     M|93510|{"x":-118.1957035...|
|California School...|     500 Walnut Ave.|94.90566037735849|     M|94536|{"x":-121.962924,...|
|California School...| 39350 Gallaudet Dr.|95.23809523809523|     M|94538|{"x":-121.962924,...|
+--------------------+--------------------+-----------------+------+-----+--------------------+
only showing top 5 rows
```

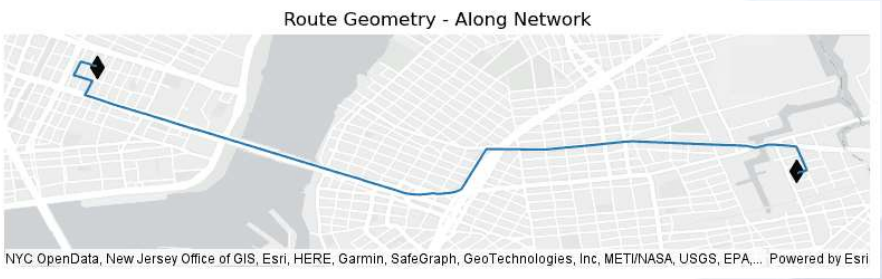
Aunque el resultado que nos devolverá dependerá de la herramienta, hay algunos campos que suelen ser comunes y claves a la mayoría de los localizadores para entender el resultado devuelto como son:

- **Status**: nos indica si se ha encontrado una dirección o no en el localizador:
  - M: encontrado
  - U: no encontrado
  - T: Más de un candidato con una puntuación muy alta, pero en diferentes lugares.
- **Score**: puntuación del candidato encontrado respecto a la dirección introducida. 100 es lo ideal.
- **Match_addr**: la dirección encontrada.
- **Addr_type**: el tipo de dirección con el que se ha emparejado. Los posibles valores que puede tomar varían entre países, pero algunos de ellos son:
  - Point Address: dirección exacta, calle y número.
  - Street Address: dirección de la calle que tiene un rango de números de edificios.
  - POI: puntos de interés como divisiones administrativas o elementos geográficos.
  - StreetName: centra en la calle pero no tiene en cuenta el número.
  - Postal: código postal.

En el caso de la herramienta **Reverse Geocode**, la configuración de los parámetros también dependerá del localizador empleado y de cómo este gestiona la obtención de direcciones a partir de coordenadas.

```
# Carga de funciones y herramientas
from geoanalytics.tools import ReverseGeocode
from geoanalytics.sql import functions as ST

# Ruta a los datos
data_url = r"https://services1.arcgis.com/Ua5sjt3LWTPigjyD/arcgis/rest/services/" \
    "Public_School_Location_201819/FeatureServer/0"

# Carga de datos como DataFrame, transformación de la referencia espacial y filtrado de datos para
California
df = spark.read.format("feature-service").load(data_url) \
                    .withColumn("shape", ST.transform("shape", 4326)) \
                    .select("shape") \
                    .where("STATE='CA'")

# Acceso al localizador, debe ser accesible para el clúster
north_america_locator = r"/data/NA_locator.loc"

# Uso de ReverseGeocode para transformar coordenadas en direcciones. Se debe definir el idioma, outputs
y campo donde están las coordenadas
result = ReverseGeocode() \
            .setLocator(north_america_locator) \
            .setOutFields("minimal") \
            .setLanguageCode("ENG") \
            .setFeatureTypes("PointAddress") \
            .run(df)

# Mostrar los 5 primeros resultados
result.show(5)


### RESULTADO
+--------------------+--------------------+------------+
|               shape|          Match_addr|   Addr_type|
+--------------------+--------------------+------------+
|{"x":-118.2159902...|33630 Red Rover M...|PointAddress|
|{"x":-118.1856342...|3015 Sacramento A...|PointAddress|
|{"x":-118.1951402...|3620 Antelope Woo...|PointAddress|
|{"x":-121.9655031...|500 Walnut Ave, F...|PointAddress|
|{"x":-121.9633661...|500 Walnut Ave, F...|PointAddress|
+--------------------+--------------------+------------+
only showing top 5 rows
```

Crear rutas



La herramienta de *Create routes* es una herramienta de análisis de redes que calcula la mejor ruta entre puntos de inicio y final de la ruta. El resultado que ofrece es un DataFrame con una línea que representa cada recorrido, pero también nos informa sobre el tiempo de la ruta en minutos y la distancia en metros. Esta herramienta nos permite generar diferentes rutas:

a) La más corta entre dos ubicaciones, por distancia o por tiempo.
b) Una ruta que recorra varios puntos según el orden que le pasemos.
c) Una ruta optimizada entre varios puntos.

Para poder utilizar esta herramienta deberemos tener una red navegable o *network dataset* que es una representación estructurada de una red de transporte para modelar desplazamientos, rutas y costes de viaje. Esta red se caracteriza porque almacena explícitamente la conectividad entre los segmentos.

Al igual que a la hora de geocodificar, los resultados que obtengamos dependerá de la red navegable y el método *describe_network_dataset* nos devuelve un diccionario con todas la propiedades de la red.

geoanalytics.util.describe_network_dataset(*path: str, extended=False*)

Returns a dictionary with the properties of the network dataset, including *name*, *path*, *spatialReference*, *defaultTravelMode*, *travelModeNames*, and *costAttributeNames*. With *extended* set to True, additional properties will be returned, including *travelModes*, *restrictions*, *restrictUsageParamNames*, and *maxLocatingDistanceMeters*.

Refer to the GeoAnalytics Engine guide for examples: Network Analysis

Parameters:
- **path** (*str*) – The path to the network data source
- **extended** (*bool, optional*) – If True, the extended version of the network dataset properties will be returned

Returns: A dictionary containing the properties of the network dataset

Return type: dict

En función de las propiedades del localizador podremos configurar el análisis de redes más o menos personalizado. En general, este tipo de redes nos permiten definir el modo de viaje (a pie, en coche, transporte especial, etc.), la geometría resultante más o menos detallada o una ruta optimizada.

```
# Carga de bibliotecas
from pyspark.sql import functions as F
from geoanalytics.sql import functions as ST
from geoanalytics.tools import CreateRoutes

# Ruta a los datos con puntos de inicio y fin en Nueva York
vehicle_locations_path = r"C:\nyc_vehicle_start_end_points"

# Ruta a la red navegable
network_dataset_path = r"C:\New_York.nmpk"

# Creación de una columna con un array con los puntos de inicio y fin de las rutas
## Con la función ST.point creo el punto pasándole longitud, latitud y referencia espacial
trip_locations_df = spark.read.format("geoparquet").load(vehicle_locations_path)\
                    .withColumn("empire_state_building", ST.point(F.lit(-73.9814406), F.lit(40.7342447), sr=4326))\
                    .withColumn("area_53", ST.point(F.lit(-73.9282016), F.lit(40.7122077), sr=4326))\
                    .withColumn("trip_locations", F.array("trip_start", "area_53", "empire_state_building", "trip_end"))

# Ejecutar CreateRouts para generar una ruta para cada vehículo minimizando el tiempo de conducción entre los puntos
# y preservando el origen y final de la ruta
optimal_trip_routes = CreateRoutes()\
                      .setNetwork(network_dataset_path)\
                      .setStops("trip_locations")\
                      .setTravelMode("Driving Time")\
                      .setRouteGeometry("AlongNetwork")\
                      .setSequence(find_best=True, preserve_first=True, preserve_last=True)\
                      .run(trip_locations_df)

# Mostrar los 5 primeros
optimal_trip_routes.select("unique_id", "trip_locations", "travel_time", "travel_distance", "route_geometry").show(5)


### RESULTADO
+---------+--------------------+-----------------+-----------------+--------------------+
|unique_id|      trip_locations|      travel_time|   travel_distance|      route_geometry|
+---------+--------------------+-----------------+-----------------+--------------------+
|        0|[{"x":-73.0718784...|47.274114166349998| 23385.418413897 14|{"paths":[[[-73.0...|
|        1|[{"x":-73.9837799...|55.219451548641 22|22991.525708181702|{"paths":[[[-73.9...|
|        2|[{"x":-73.9828556...| 65.9056857881696|25997.134204527312|{"paths":[[[-73.9...|
|        3|[{"x":-74.0005111...|47.145276206564 58|19792.443211289775|{"paths":[[[-74.0...|
|        4|[{"x":-73.0650591...|51.053054740384 47| 25497.14207067214|{"paths":[[[-73.0...|
+---------+--------------------+-----------------+-----------------+--------------------+
only showing top 5 rows
```

Crear áreas de servicio

La herramienta ***Create Service Area*** se utiliza para determinar el área accesible desde una o varias ubicaciones de origen dentro de una red navegalbe, teniendo en cuenta criterios como el tiempo de viaje, la distancia o cualquier otro coste de desplazamiento. El resultado es un conjunto de áreas que representan el alcance de un servicio o recurso dentro de la red.

Este análisis también es un análisis de redes por lo que también tenemos que cargar la red navegable de la zona con el parámetro *setNetwork*.

```
# Carga de la biblioteca
from geoanalytics.tools import CreateServiceAreas
import matplotlib.pyplot as plt

# Ruta a los datos
data_url = r"https://services.arcgis.com/P3ePLMYs2RVChkJx/ArcGIS/rest/services/SDFireStations/FeatureServer/0"

# Crear DataFrame
df = spark.read.format("feature-service").load(data_url) \
          .select("FACILITYID", "NAME", "FULLADDR", "PHONE", "ACTIVE", "shape")

# Acceso a los datos de red, necesita ser accesible para todo el clúster
california_network = r"/data/California.smpk"

# Ejecuta la herramienta Create Service Areas
result = CreateServiceAreas() \
        .setNetwork(california_network) \ # Red navegable
        .setTravelMode("driving time") \ # Modo de viaje
        .setTravelDirection("FromFacilities") \ # Desde los elementos (puede ser configurado hacia ellos)
        .setCutoffs([2, 4], "minutes") \ # Cortes de tiempo y unidad
        .setPolygonDetail("standard") \
        .setGeometryAtCutoff("rings") \
        .run(df)

# Mostrar los 5 primeros resultados
result.sort("FACILITYID", "cutoff").show(5)

## RESULTADO
+----------+----------+----------------+------------+------+--------------------+---------+--------------------+
|FACILITYID|      NAME|        FULLADDR|       PHONE|ACTIVE|               shape|   cutoff|service_area_polygon|
+----------+----------+----------------+------------+------+--------------------+---------+--------------------+
|         1|SD FS 1/2|1222 First Avenue|619-533-4300|   Yes|{"x":-117.1644945...|2 minutes|{"rings":[[[-117....|
|         1|SD FS 1/2|1222 First Avenue|619-533-4300|   Yes|{"x":-117.1644945...|4 minutes|{"rings":[[[-117....|
|        10|   SD FS 10| 4605 62nd Street|619-533-4300|   Yes|{"x":-117.0637437...|2 minutes|{"rings":[[[-117....|
|        10|   SD FS 10| 4605 62nd Street|619-533-4300|   Yes|{"x":-117.0637437...|4 minutes|{"rings":[[[-117....|
|        11|   SD FS 11|  945 25th Street|619-533-4300|   Yes|{"x":-117.1400215...|2 minutes|{"rings":[[[-117....|
+----------+----------+----------------+------------+------+--------------------+---------+--------------------+
only showing top 5 rows
```

Funciones *tracks*

Este módulo de GeoAnalytics Engine proporciona un conjunto de funciones para gestionar y analizar datos de seguimiento. Los *tracks* son líneas que representan el cambio de ubicación de una entidad a lo largo

del tiempo, donde cada vértice tiene un *timestamp* almacenado como valor M y están ordenados secuencialmente. Este tipo de funciones son útiles para analizar rutas de vehículos de reparto y detectar anomalías, analizar movimientos históricos de buques a partir de datos AIS, etc.

Al igual que las funciones SQL, hay muchas funciones documentadas, pero vamos a ver solo las más utilizadas:

TRK_Length

Esta función toma la columna con la geometría de la línea y devuelve una DataFrame con una nueva columna con la longitud de cada elemento.

```
from geoanalytics.tracks import functions as TRK
from geoanalytics.sql import functions as ST
from pyspark.sql import functions as F

data = [
    ("LINESTRING M (-117.27 34.05 1633455010, -117.22 33.91 1633456062, -116.96 33.64 1633457132)",),
    ("LINESTRING M (-116.89 33.96 1633575895, -116.71 34.01 1633576982, -116.66 34.08 1633577061)",),
    ("LINESTRING M (-116.24 33.88 1633575234, -116.33 34.02 1633576336)",)
]

df = spark.createDataFrame(data, ["wkt",]) \
          .withColumn("track", ST.line_from_text("wkt", srid=4326))

df.select(F.round(TRK.length("track", "miles"), 3).alias("length")).show()

### RESULTADO
+------+
|length|
+------+
|33.947|
|16.507|
|10.947|
+------+
```

TRK_Speed

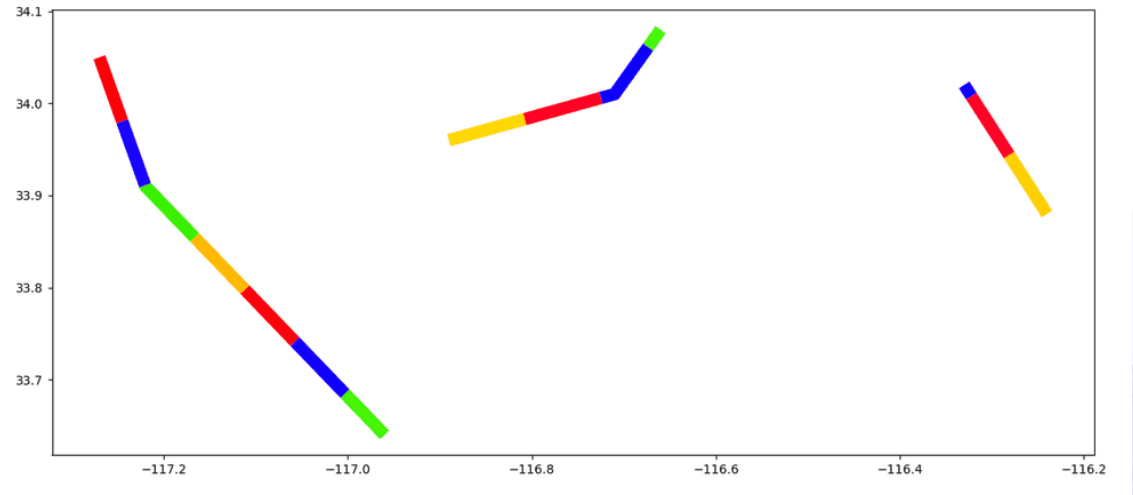
Esta función calcula la velocidad de una ruta y crea una nueva columna numérica donde añade el resultado. El resultado es obtenido a partir de la división de los datos de longitud de la ruta y su duración. La unidad del resultado pueden ser metros por segundo, millas náuticas por hora, pies por segundo, kilómetros por hora o millas por hora.

```
from geoanalytics.sql import functions as ST
from geoanalytics.tracks import functions as TRK
from pyspark.sql import functions as F

data = [
    ("LINESTRING M (-117.27 34.05 1633455010, -117.22 33.91 1633456062, -116.96 33.64 1633457132)",),
    ("LINESTRING M (-116.89 33.96 1633575895, -116.71 34.01 1633576982, -116.66 34.08 1633577061)",),
    ("LINESTRING M (-116.24 33.88 1633575234, -116.33 34.02 1633576336)",)
]

df = spark.createDataFrame(data, ["wkt",]) \
          .withColumn("track", ST.line_from_text("wkt", srid=4326))

df.select(F.round(TRK.speed("track", "MilesPerHour"), 3).alias("speed")).show()

### RESULTADO
+------+
| speed|
+------+
|57.591|
|50.966|
|35.761|
+------+
```

TRK_SplitByDistance

Esta función toma la columna de la ruta y el parámetro de distancia y devuelve una nueva línea. Este array resultante contiene la ruta dividida en fragmentos de la longitud especificada. Para definir esa distancia con su unidad podemos usar la función *ST_CreateDistance* o una tupla con la información como `(5, "meters")`.

```
from geoanalytics.sql import functions as ST
from geoanalytics.tracks import functions as TRK
from pyspark.sql import functions as F

data = [
    ("LINESTRING M (-117.27 34.05 1633455010, -117.22 33.91 1633456062, -116.96 33.64 1633457132)",),
    ("LINESTRING M (-116.89 33.96 1633575895, -116.71 34.01 1633576982, -116.66 34.08 1633577061)",),
    ("LINESTRING M (-116.24 33.88 1633575234, -116.33 34.02 1633576336)",)
]

df = spark.createDataFrame(data, ["wkt"]).withColumn("track", ST.line_from_text("wkt", srid=4326))

result = df.withColumn("split_by_distance", TRK.split_by_distance("track", (5, "miles")))

result.select(F.explode("split_by_distance"), F.monotonically_increasing_id().alias("id")) \
      .st.plot(is_categorical=True, cmap_values="id", cmap="prism", linewidths=10, figsize=(15, 8))
```



Visualización de resultados

La visualización de los datos espaciales es un paso clave para dar contexto a los datos y ayudar a la hora de encontrar patrones, tendencias y relaciones de los datos. Además, no se debe ver como un resultado final sino como un paso intermedio que nos permite iterar sobre el procesamiento y análisis de los datos hechos hasta hora para valorar y, quizás, reajustar el planteamiento o el análisis. Con GeoAnalytics Engine hay dos formas de visualizar los datos:

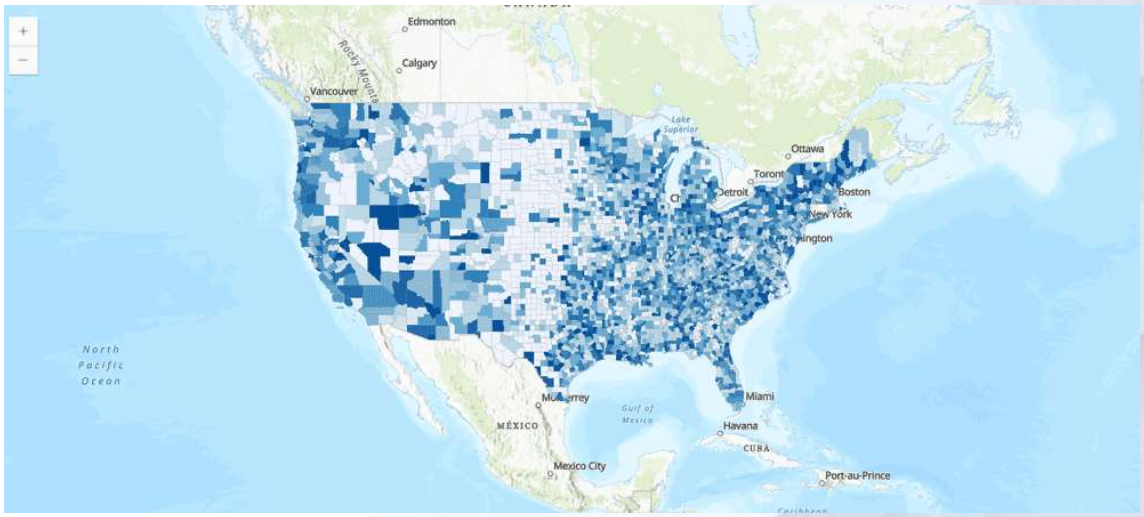
- Dentro del propio notebook lo que nos permite hacer visualizaciones rápidas a la vez que se analizan los datos.
- Visualizarlo en aplicaciones externas para poder, por ejemplo, completar los resultados obtenidos, compartir con el resto del equipo de forma sencilla o crear aplicaciones nuevas.

Visualizaciones con la ArcGIS API for Python

La ArcGIS API for Python contiene un módulo de mapa que permite extender las capacidades de visualización de ArcGIS GeoAnalytics Engine a través del widget de mapa interactivo. El uso de este widget nos permite usar más mapas base, hacer zoom, mover el mapa e interactuar con los elementos. Para poder hacer visualizaciones con la ArcGIS API for Python, debemos convertir el DataFrame de pandas a Spatially Enabled DataFrame (sedf) con la función st.to_pandas_sdf(). El **sedf** es un un DataFrame que tiene capacidades espaciales integrada, es decir, permite almacenar, manipular y analizar datos geográficos. Al convertir un DataFrame de Spark en un Spatially Enabled DataFrame, todas las particiones del conjunto de datos se mueven a un único nodo del clúster. En el caso de conjuntos de datos muy grandes, esto puede ser muy lento y podría fallar si no hay suficientes recursos disponibles para almacenar los datos en un único nodo.

Hay que tener en cuenta que el mapa de widget solo está disponible para notebooks de Jupyter y JupyterLab. No está disponible en otro tipo de distribuciones incluyendo: JupyterHub, Zeppelin, Databricks y EMR.

```
# Carga ArcGIS API for Python
from arcgis import GIS

# Carga el widget de mapa y define el centro (EEUU)
gis = GIS()
usa_map = gis.map("USA")

# Ruta a los datos
url = "https://services.arcgis.com/P3ePLMYs2RVChkJx/ArcGIS/rest/services/USA_Counties_Generalized_Boundaries/FeatureServer/0"

# Carga datos como DataFrame
us_counties = spark.read.format("feature-service").load(url)


# Convierte DataFrame en SEDF
counties_sdf = us_counties.where("POPULATION IS NOT NULL").st.to_pandas_sdf()

# Define población como número
counties_sdf = counties_sdf[["shape", "POPULATION"]].astype({"POPULATION":"int64"})

# Dibuja los datos dentro del widget de mapa definido al inicio (usa_map)
counties_sdf.spatial.plot(map_widget=usa_map)

# Carga el widget de mapa con el contenido
usa_map
```



Visualizaciones en los notebooks

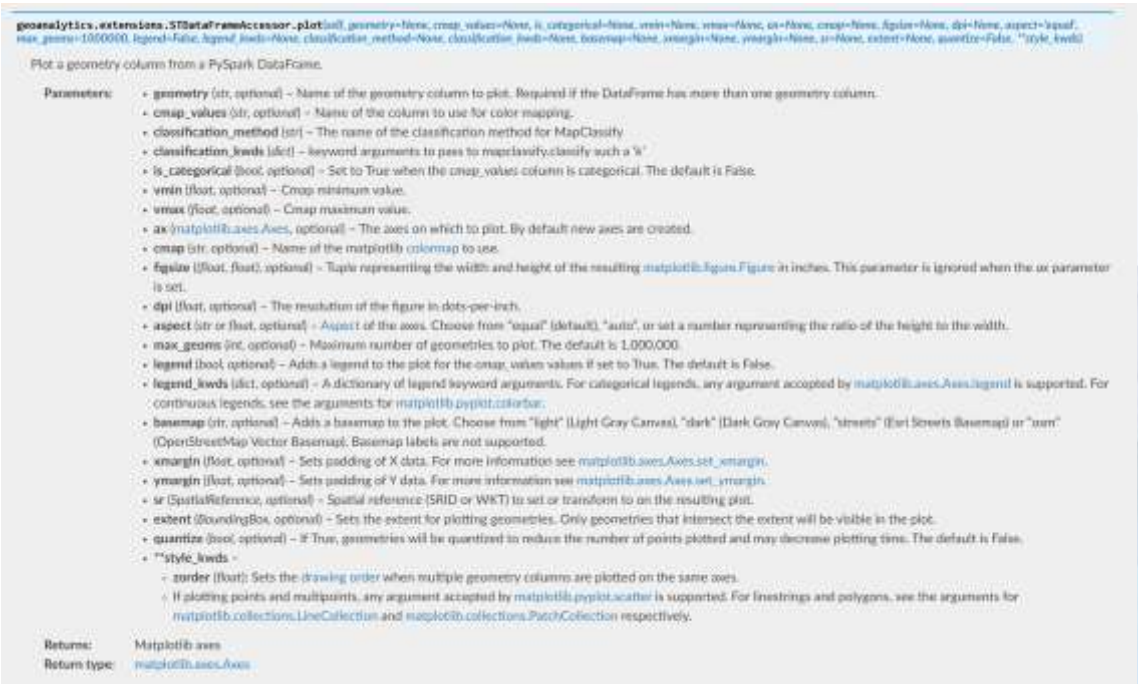
Para hacer visualizaciones dentro de los propios podemos usar el propio método ***st.plot*** de una forma rápida o usar el widget de mapa de la **ArcGIS API for Python** para crear visualizaciones más interactivas. Lo más común es usar ambas ya que las capacidades son distintas:

	GeoAnalytics Engine	ArcGIS API for Python
Basemaps	◔[1]	●
Draw multiple layers	●	●
Zooming	○	●
Identify records	○	●
Modify colors and symbols	●	●
Included with GeoAnalytics	●	○
Record limit	●	●

● *Full support* ◔ *Partial support* ○ *No support*

1. New in version 1.1.0. Requires a connection to ArcGIS Online

El método *st.plot* de GeoAnalytics Engine extiendo las capacidades de matplotlib permitiendo la visualización de geometrías de uno o más de un DataFrame.



geoanalytics.extensions.STDataFrameAccessor.plot(self, geometry=None, cmap_values=None, is_categorical=None, vmin=None, vmax=None, ax=None, cmap=None, figsize=None, dpi=None, aspect='equal', max_geoms=1000000, legend=False, legend_kwds=None, classification_method=None, classification_kwds=None, basemap=None, xmargin=None, ymargin=None, sr=None, extent=None, quantize=False, **style_kwds)

Plot a geometry column from a PySpark DataFrame.

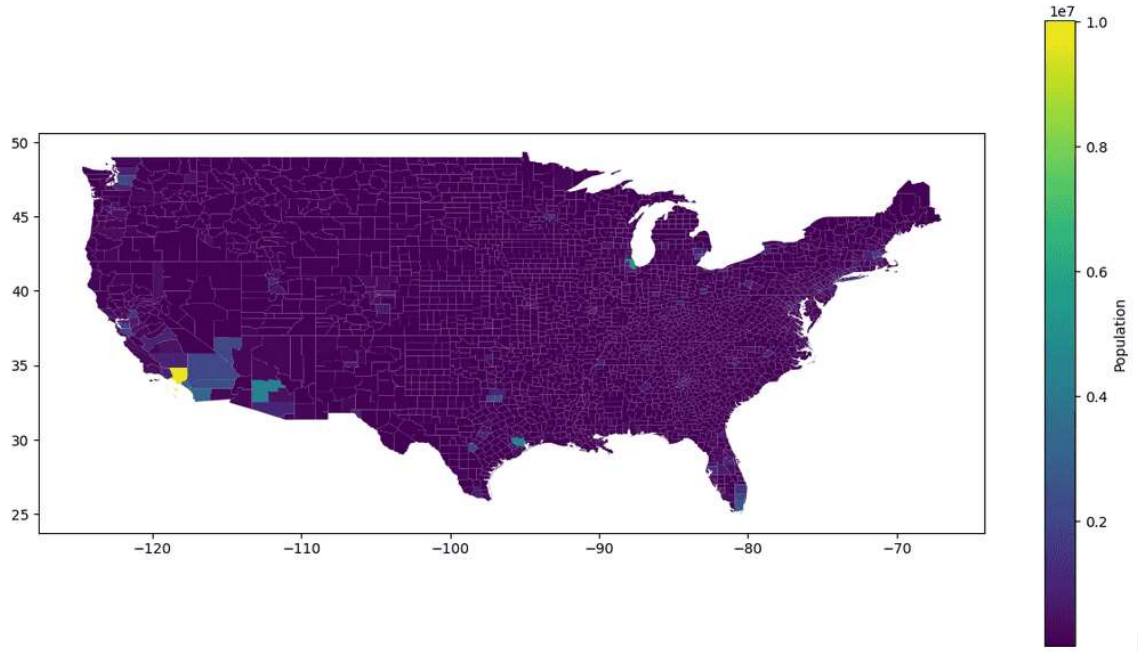
Parameters:

- **geometry** (str, optional) – Name of the geometry column to plot. Required if the DataFrame has more than one geometry column.
- **cmap_values** (str, optional) – Name of the column to use for color mapping.
- **classification_method** (str) – The name of the classification method for MapClassify.
- **classification_kwds** (dict) – keyword arguments to pass to mapclassify.classify such a 'k'.
- **is_categorical** (bool, optional) – Set to True when the cmap_values column is categorical. The default is False.
- **vmin** (float, optional) – Cmap minimum value.
- **vmax** (float, optional) – Cmap maximum value.
- **ax** (matplotlib.axes.Axes, optional) – The axes on which to plot. By default new axes are created.
- **cmap** (str, optional) – Name of the matplotlib colormap to use.
- **figsize** ((float, float), optional) – Tuple representing the width and height of the resulting matplotlib.figure.Figure in inches. This parameter is ignored when the ax parameter is set.
- **dpi** (float, optional) – The resolution of the figure in dots-per-inch.
- **aspect** (str or float, optional) – Aspect of the axes. Choose from "equal" (default), "auto", or set a number representing the ratio of the height to the width.
- **max_geoms** (int, optional) – Maximum number of geometries to plot. The default is 1,000,000.
- **legend** (bool, optional) – Adds a legend to the plot for the cmap_values values if set to True. The default is False.
- **legend_kwds** (dict, optional) – A dictionary of legend keyword arguments. For categorical legends, any argument accepted by matplotlib.axes.Axes.legend is supported. For continuous legends, see the arguments for matplotlib.pyplot.colorbar.
- **basemap** (str, optional) – Adds a basemap to the plot. Choose from "light" (Light Gray Canvas), "dark" (Dark Gray Canvas), "streets" (Esri Streets Basemap) or "osm" (OpenStreetMap Vector Basemap). Basemap labels are not supported.
- **xmargin** (float, optional) – Sets padding of X data. For more information see matplotlib.axes.Axes.set_xmargin.
- **ymargin** (float, optional) – Sets padding of Y data. For more information see matplotlib.axes.Axes.set_ymargin.
- **sr** (SpatialReference, optional) – Spatial reference (SRID or WKT) to set or transform to on the resulting plot.
- **extent** (BoundingBox, optional) – Sets the extent for plotting geometries. Only geometries that intersect the extent will be visible in the plot.
- **quantize** (bool, optional) – If True, geometries will be quantized to reduce the number of points plotted and may decrease plotting time. The default is False.
- ****style_kwds** –
  - **zorder** (float): Sets the drawing order when multiple geometry columns are plotted on the same axes.
  - If plotting points and multipoints, any argument accepted by matplotlib.pyplot.scatter is supported. For linestrings and polygons, see the arguments for matplotlib.collections.LineCollection and matplotlib.collections.PatchCollection respectively.

Returns: Matplotlib axes
Return type: matplotlib.axes.Axes

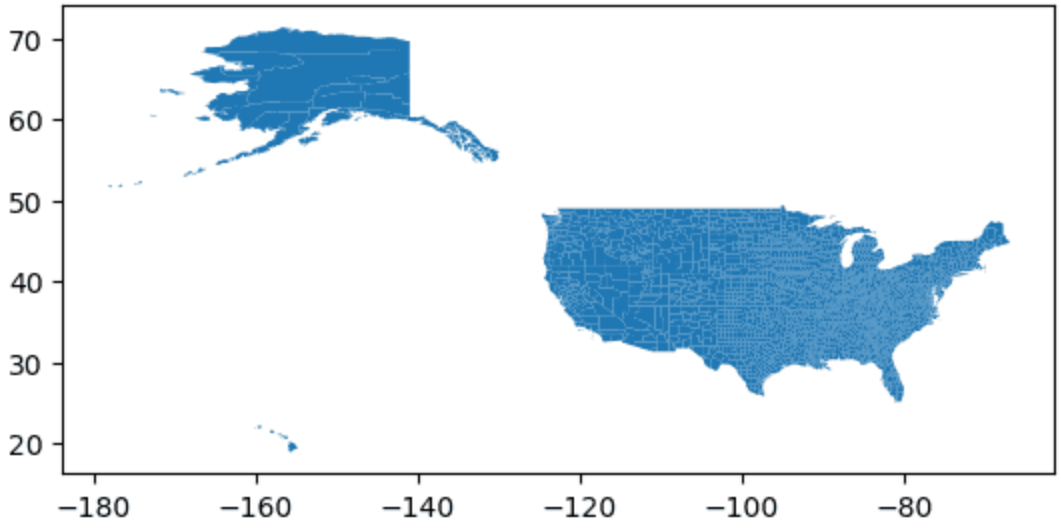
Como podemos ver en sus parámetros, este método nos permite una configuración muy detallada y con una sintaxis y variables conocidas de matplotlib.

```
# Carga de biblioteca
import geoanalytics

# Ruta de los datos
url = "https://services.arcgis.com/P3ePLMYs2RVChkJx/ArcGIS/rest/services/USA_Counties_Generalized_Boundaries/FeatureServer/0"

# Carga de los datos como DataFrame sin Alaska ni Hawái
counties = spark.read.format('feature-service').load(url).where("STATE_ABBR NOT IN ('AK','HI')")

# Dibujo de los condados en función de la variable población
counties.st.plot(cmap_values="POPULATION", figsize=(15, 8), legend=True, legend_kwds={"label": "Population"})
```



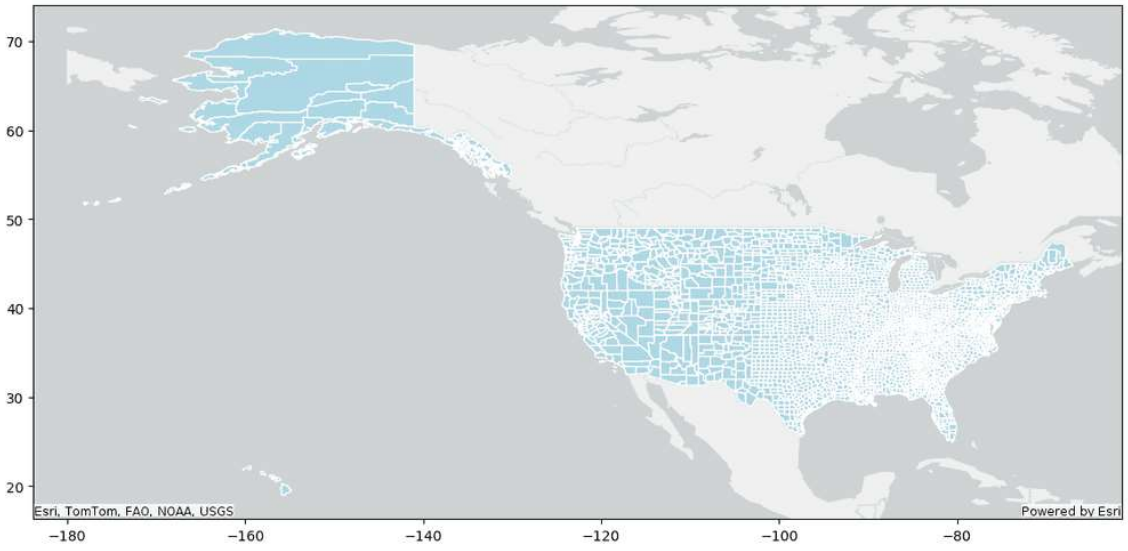
Todos esos parámetros de la función st.plot nos van a permitir crear visualizaciones personalizadas de una forma simple y rápida. De hecho, si ejecutamos la función sobre un DataFrame de datos, aunque sea sin parámetros obtendremos una sencilla visualización de ellos.

```
# Ruta a los datos
url = "https://services.arcgis.com/P3ePLMYs2RVChkJx/ArcGIS/rest/services/USA_Counties_Generalized_Boundaries/FeatureServer/0"

# Carga de datos como DataFrame
us_counties = spark.read.format('feature-service').load(url)

# Pintar el DataFrame
us_counties.st.plot()
```



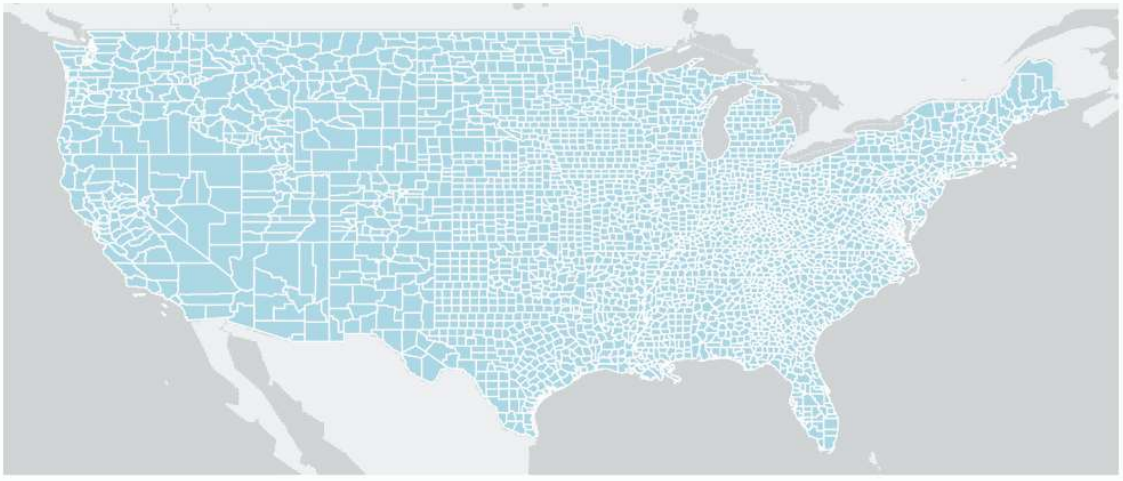
Sin embargo, con unos simples parámetros más, obtenemos una visualización con más contexto y personalizada.

```
us_counties.st.plot(facecolor="lightblue", # Color de los polígonos
                    edgecolor="white", # Borde de los polígonos
                    figsize=(14, 14), # Tamaño de la figura
                    basemap="light") # Mapa base
```



Podríamos mejorar la visualización si centrásemos el mapa un poco más. Para ello, podemos usar el método *set* de los ejes que usa matplotlib e incluso ocultarlos.

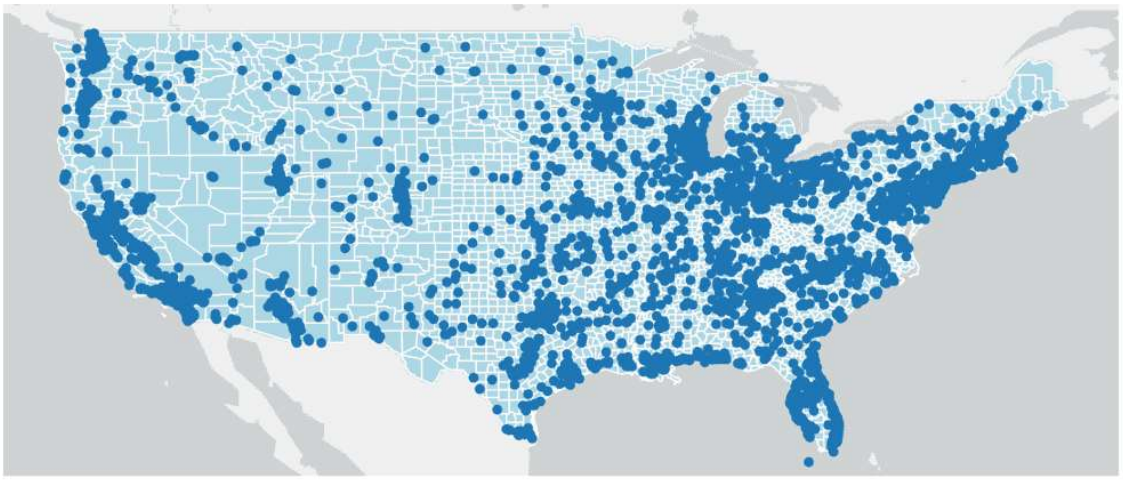
```
axes = us_counties.st.plot(facecolor="lightblue", # Color polígono
                           edgecolor="white", # Borde polígono
                           figsize=(14, 14), # Tamaño
                           basemap="light") # Mapa base

axes.set(xlim=(-127.6209447668401, -64.08109605872191), # Límites eje X
         ylim=(23.73560869553325, 50.59249779819335), # Límites eje Y
         frame_on=False, # No pinta el marco
         xticks=[], yticks=[]) # Sin valores en los ejes
```



Una web útil para definir de una forma más sencilla los ejes de nuestras visualizaciones es bboxfinder.com que nos permite dibujar una zona y extraer las coordenadas fácilmente de los límites.

Los ejes devueltos por la función *st.plot* podemos reutilizarlos para pintar más datos geográficos y de diferente naturaleza, por ejemplo, podemos añadir una capa de puntos con las ciudades más pobladas de EEUU.

```
axes = us_counties.st.plot(facecolor="lightblue", # Color polígono
                           edgecolor="white", # Borde polígono
                           figsize=(14, 14), # Tamaño
                           basemap="light") # Mapa base

axes.set(xlim=(-127.6209447668401, -64.08109685872191), # Límites eje X
         ylim=(23.73560869553325, 50.59249779819335), # Límites eje Y
         frame_on=False, # No pinta el marco
         xticks=[], yticks=[]) # Sin valores en los ejes

# Ruta a los datos de ciudades
url = "https://services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services/USA_Major_Cities_/FeatureServer/0"

# Carga como DataFrame
us_cities = spark.read.format("feature-service").load(url)

# Pintar en el mapa con las opciones por defecto e indicando los ejes anteriores
us_cities.st.plot(ax=axes)
```

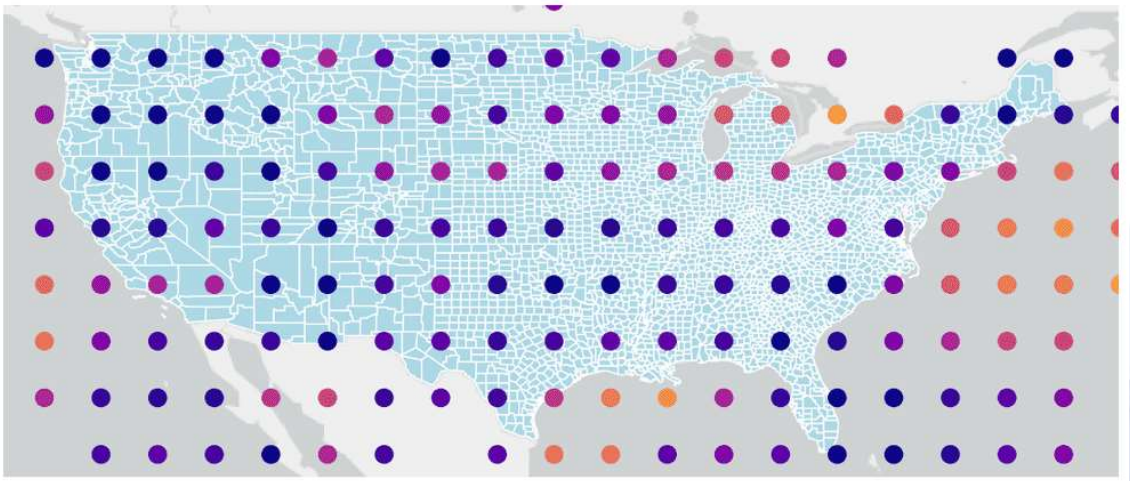


Estas visualizaciones que hemos planteado hasta ahora son útiles, pero podemos personalizarlas para sacar aún más información. Por ejemplo, podemos representar alguna variable para saber la ubicación de los datos y algún valor:

```
axes = us_counties.st.plot(facecolor="lightblue", # Color polígono
                           edgecolor="white", # Borde polígono
                           figsize=(14, 14), # Tamaño
                           basemap="light") # Mapa base

axes.set(xlim=(-127.6209447668401, -64.08109685872191), # Límites eje X
         ylim=(23.73560869553325, 50.59249779819335), # Límites eje Y
         frame_on=False, # No pinta el marco
         xticks=[], yticks=[]) # Sin valores en los ejes

# Ruta a los datos de ciudades
url = "https://services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services/USA_Major_Cities_/FeatureServer/0"

# Carga como DataFrame
us_cities = spark.read.format("feature-service").load(url)

# Pintar en el mapa en función de la variable población
us_cities.st.plot(
    ax=axes,
    column="POPULATION",
    color="red",
    alpha=0.6
)
```

Esas variables pueden ser continuas, es decir, variables que tienen un rango de valores numéricos continuos.

```
axes = us_counties.st.plot(facecolor="lightblue", # Color polígono
                           edgecolor="white", # Borde polígono
                           figsize=(14, 14), # Tamaño
                           basemap="light") # Mapa base

axes.set(xlim=(-127.6209447668401, -64.88109085872191), # Límites eje X
         ylim=(23.73560069553325, 50.59249779819335), # Límites eje Y
         frame_on=False, # No pinta el marco
         xticks=[], yticks=[]) # Sin valores en los ejes

# Ruta a los datos con información de velocidad del viento
url = "https://services9.arcgis.com/RHVPKKiFTONKtxq3/ArcGIS/rest/services/NDFD_WindForecast_v1/FeatureServer/0"

# Cargar datos como DataFrame
wind_speed = spark.read.format("feature-service").load(url)

# Obtener el instante más reciente de cada localización
most_recent_measure_time = wind_speed.agg({'IntervalStart': 'min'}).collect()[0][0]

# Filtrar por ese valor
wind_speed = wind_speed.where(wind_speed.IntervalStart == most_recent_measure_time)

wind_speed.st.plot(ax=axes, # Eje con el mapa
                   cmap_values="WindSpeed", # Variable que se va a representar
                   cmap="plasma", # Rampa de colores utilizada
                   marker_size=150, # Tamaño fijo para todos los puntos
                   vmax=30, # Valor máximo para la escala de color
                   vmin=5) # Valor mínimo para la escala de color
```

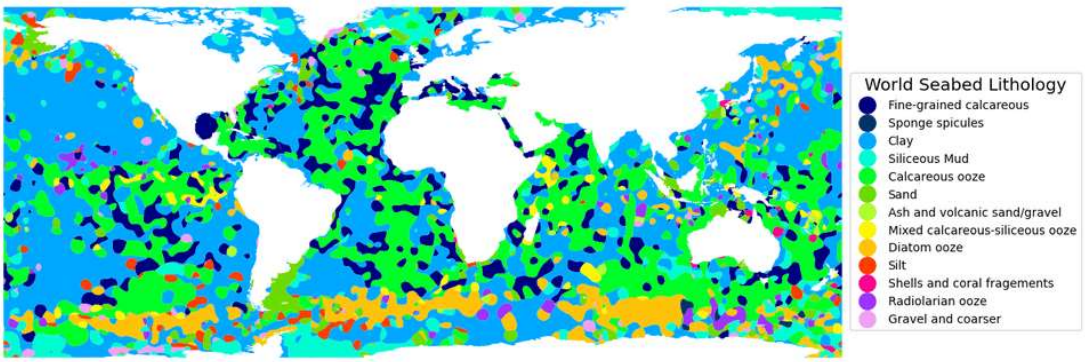


Las variables que definen la visualización de los datos provienen de matplotlib y tenemos acceso a esos parámetros detallados en la documentación de *matplotlib* y acceso directo desde la definición st.plot de la API Reference. Las *cheatsheets* de *matplotlib* complementan muy bien la documentación oficial.

```
plot_color_gradients('Perceptually Uniform Sequential',
                     ['viridis', 'plasma', 'inferno', 'magma', 'cividis'])
```

De forma muy similar podemos representar datos categóricos, es decir, datos clasificados en diferentes categorías. En el siguiente ejemplo, se representan los distintos tipos de litología marina guardados en la variable “Class” y que toma valores como: gravel and coarser, sand, silt, clay …

```
# Ruta a los datos con información de litología marina
url = "https://services.arcgis.com/P3ePLMYs2RVChkJx/arcgis/rest/services/Seabed_Lithology/FeatureServer/0"

# Cargar datos como DataFrame
seabed_lithology = spark.read.format("feature-service").load(url)

# Representación
axes = seabed_lithology.st.plot(is_categorical=True, # Indicar que la variable es categórica
                                cmap_values="Class", # Variable a representar
                                cmap="gist_ncar", # Color de la representación
                                figsize=(14,14), # Tamaño
                                legend=True, # Añadir leyenda
                                legend_kwds={"bbox_to_anchor": (1.22, 0.8),
                                             "fontsize": "medium",
                                             "markerscale": 3,
                                             "title": "World Seabed Lithology",
                                             "title_fontsize": "x-large"})

# Definir ejes
axes.set(frame_on=False, xticks=[], yticks=[])
```



ArcGIS API for Python

La API de Python de ArcGIS es una biblioteca de Python potente, moderna y fácil de usar que permite realizar tareas de visualización y análisis GIS, gestión de datos espaciales y administración de sistemas GIS. Además, gracias a poder hacerlo mediante scripts, es la biblioteca que nos permite automatizar procesos disminuyendo así la cantidad de tiempo invertido en determinadas tareas.

Se trata de una pythonic API, es decir, sigue estándares de buenas prácticas de Python en su diseño y utiliza construcciones y estructuras de datos estándar de Python con expresiones claras y legibles. La API cuenta con módulos, clases, funciones y tipos de Python para gestionar y trabajar con elementos del modelo de geoinformación de ArcGIS. La API se distribuye como el paquete de *arcgis* a través de conda y pip. Con el objetivo de facilitar el uso y la compresión de la API, ésta está organizada en módulos que contienen varios tipos y funciones centrados en un aspecto concreto del GIS. A continuación podemos ver una representación de los módulos:

El módulo de GIS es el más importante ya que es el punto de entrada y el que nos va a permitir administrar toda la organización: usuarios, grupos, contenido, etc. Sin embargo, no necesitaremos instalar la ArcGIS API de Python para compartir contenido ya que, el método register_gis hará directamente la conexión a la organización.

geoanalytics.register_gis(*name, url='https://arcgis.com', *, username=None, password=None, cert_file=None, cert_password=None, client_id=None, authorization_code=None*)

Registers a GIS with the system. The GIS can be ArcGIS Online or ArcGIS Enterprise.

The name of the GIS can be supplied in place of credentials when accessing resources within the GIS (e.g. feature services).

Parameters:
- **name** (*str*) – a user-defined name that identifies the GIS.
- **url** (*str*) – URL for the GIS (e.g. https://arcgis.com or https://enterprise.example.com/portal). The default is ArcGIS Online.
- **username** (*str*) – username for the GIS.
- **password** (*str*) – password for the GIS.
- **cert_file** (*str*) – Path to certificate file.
- **cert_password** (*str*) – Password for certificate.
- **client_id** (*str*) – Client id used for registering an OAuth 2.0 protected GIS.
- **authorization_code** (*str*) – Authorization code is required when *client_id* is provided. Generate an authorization code through the following URL (replace "<url>" and "<client_id>" with the *url* and *client_id*): <url>/sharing/oauth2/authorize?client_id=<client_id>&response_type=code&redirect_uri=urn:ietf:wg:oauth:2.0:oob.

Como hemos comentado en el apartado de visualización, la API de Python nos va a permitir crear visualizaciones personalizadas, pero no en entornos como Databricks o Amazon EMR, por lo que no profundizaremos más en esta API en este curso. Sin embargo, ArcGIS API for Python cuenta con una amplia documentación y una gran cantidad de ejemplos disponibles.

https://developers.arcgis.com/python/latest/

Compartir resultados de análisis

En el apartado de "Carga de datos" hemos revisado uno a uno los datos principales que se pueden integrar en Spark gracias a ArcGIS GeoAnalytics Engine. Además, en ese apartado, a parte de hablar de cómo cargar datos también se ha hablado de cómo guardarlos. Guardar los datos espaciales puede ser un método válido para compartir los resultados, pero siguiendo el modelo de geoinformación, en este apartado vamos a ver cómo compartir los resultados de nuestros análisis con nuestra organización.

Con el método *register_gis* podemos registrar una organización, de ArcGIS Online o ArcGIS Enterprise. Esto nos va a permitir tener una conexión directa para añadir nuevo contenido o actualizarlo. Esta funcionalidad usa DataFrameWriter de Spark y el flujo, simplificado, sería el siguiente:

1. Acceso a la organización. En este ejemplo se hace por usuario y contraseña pero hay más métodos de autenticación disponibles.

```
# Registro de la organización
geoanalytics.register_gis("myGIS",  # Alias al que haremos referencia
                          "https://arcgis.com", # URL de la organización
                          username="User",
                          password="p@ssw0rd")
```

2. Manejo de datos espaciales en DataFrame.
3. Publicación del DataFrame en la organización como un nuevo servicio.

```
service_name = "continents"
df.write.format("feature-service") \
        .option("gis", "myGIS") \
        .option("serviceName", service_name) \
        .option("layerName", "layer") \
        .option("tags", "continents, boundaries") \
        .option("description", "This is an example feature service showing boundaries of world continents") \
        .save()
```

Si quisiéramos actualizar un servicio ya publicado tan solo tendríamos que indicar la URL del servicio de datos e indicar la acción en el parámetro ***mode.***

```
north_america = df.where("continent = 'North America'")

service_url = "https://<host>/<uniqueID>/ArcGIS/rest/services/<serviceName>/FeatureServer"
north_america.write.format("feature-service") \
             .option("gis", "myGIS") \
             .option("serviceUrl", service_url) \
             .option("layerName", "layer") \
             .mode("overwrite") \
             .save()
```

Si, por el contrario, quisiéramos añadir nuevos datos, el parámetro *mode* admite el valor ***append.***

```
south_america = df.where("continent = 'South America'")

south_america.write.format("feature-service") \
             .option("gis", "myGIS") \
             .option("serviceUrl", service_url) \
             .option("layerName", "layer") \
             .mode("append") \
             .save()
```

Ejemplos:

Análisis migratorio de las grullas (*Grus grus)*

A partir de los datos de la plataforma eBird vamos a hacer un análisis exploratorio de los datos para ver la migración de las grullas. Los pasos a seguir para completar este análisis son:

1. **Carga de bibliotecas**. Para hacer este análisis necesitaremos cargar las siguientes bibliotecas:

```
from pyspark.sql.functions import col, to_timestamp, year, month, to_date

import matplotlib.pyplot as plt

from pyspark.sql import functions as F


import geoanalytics

import geoanalytics.sql.functions as ST

from geoanalytics.tools import AggregatePoints, CalculateDensity
```

2. **Iniciar sesión** en GeoAnalytics Engine y así tener acceso a todas las herramientas y funciones.

```
geoanalytics.auth(api_key='XXXXXX')
print('La versión de GeoAnalytics Engine es: ', geoanalytics.version())
```

3. **Añadir los datos que analizaremos**. Están disponibles en formato CSV en la siguiente ruta: `s3://datos-tecnologia/ebird/data_ebird.csv`

```
df = spark.read.csv("s3://datos-tecnologia/ebird/data_ebird.csv", header=True)

rows = df.count()
print(f"Numero de filas : {rows:,}")
cols = len(df.columns)
print(f"Numero de columnas : {cols}")
```

   Explorar los datos y limpiarlos. El método *printSchema()* nos dará información relevante sobre qué variables y de qué tipo son los datos. Hay que hacer hincapié en las columnas que nos informan sobre las geometrías y las fechas. Como queremos ver los cambios a lo largo de los meses, necesitamos una columna con la información del mes (número del mes).

```
##### ERRORES COORDENADAS
df = df.filter(col("LATITUDE").isNotNull())
df = df.filter(col("LONGITUDE").isNotNull())

##### ERRORES FECHA OBSERVACIÓN
df = df.withColumn("OBSERVATION DATE", to_timestamp(col("OBSERVATION DATE"), "yyyy-MM-
dd"))

##### Filtrar las filas donde la conversión a timestamp no es nula (es decir, son fechas
válidas)
df = df.filter(col("OBSERVATION DATE").isNotNull())

##### Añadir columna mes y año
df = df.withColumn("YEAR", year("OBSERVATION DATE"))
df = df.withColumn("MONTH", month("OBSERVATION DATE"))
```

   Definir los campos de geometría y tiempo. Una vez que tenemos limpios los datos, tendremos que definir la geometría con la función *ST_Point* y también definir la variable temporal con *st.set_time_fields*.

```
df = df.selectExpr("*", "ST_Point(Longitude, Latitude, 4326) as SHAPE")\
    .st.set_geometry_field("SHAPE")\
    .withColumn("FECHA", to_timestamp('LAST EDITED DATE', "yyyy-MM-dd
HH:mm:ss.SSSSSS")).st.set_time_fields("FECHA")
```

4. **Visualizar los datos**. Al ser una gran cantidad de datos, vamos a agregarlos para poder visualizarlos con la herramienta *AggregatePoints*. Ten en cuenta que para ejecutar esta herramienta necesitamos trabajar con **coordenadas proyectadas**.

```
result = AggregatePoints().setBins(bin_size=600,
bin_size_unit="Meters",bin_type="Hexagon").run(df_proyectadas)

axes = result.st.plot(cmap_values="COUNT",
                      basemap="light",
                      cmap="cool",
                      vmax=1000,
                      vmin=5,
                      figsize = (16,10),
                      legend=True,
                      legend_kwds={"label": "Total posiciones aves"}
                     )

%matplot plt
```

5. **Centrar el mapa**. A parte de ver la distribución de toda la península, vamos a centrarnos en Andalucía ya que el Estrecho de Gibraltar es uno de los puntos clave para las aves migratorias.

   Listar todas las especies de los datos con el número de aves. Aunque nos centraremos en los siguientes pasos en las grullas, esto nos servirá para tener una visión general de los avistamientos registrados.

```
df_count = df.groupBy("SCIENTIFIC NAME").count()
df_count_sorted = df_count.orderBy(col("count").desc())
df_count_sorted.show()
```

6. **Datos de grullas.** Para ver la migración de las grullas vamos a utilizar un nuevo conjunto de datos que contengan datos de la especie en toda Europa. Los datos están en esta ruta: `s3://datos-tecnologia/ebird/Grullas.csv`

7. **Explorar y limpiar estos datos**. Ten en cuenta que provienen ambos datasets de eBird, es decir, el proceso es muy parecido.

8. **Distribución de los avistamientos.** Para visualizar los avistamientos de esta especie a nivel global, ejecutaremos la herramienta de cálculo de densidad. De esta forma podremos hacernos una idea del hábitat de la misma y cómo se distribuye por Europa.

```
result_grulla = CalculateDensity() \
            .setWeightType(weight_type="Uniform") \
            .setBins(bin_size=10, bin_size_unit="Kilometers", bin_type="Square") \
            .setNeighborhood(distance=20, distance_unit="Kilometers") \
            .setAreaUnit(area_unit="SquareKilometers") \
            .run(dataframe=df_grulla)

result_grulla.st.plot(cmap_values="density",
                                cmap="YlOrRd",
                                legend=False,
                                 figsize=(20, 10),
                                basemap="light")
plt.title("Densidad grulla")
```

```
%matplot plt
```

9. **Avistamientos en la migración.** Haremos la misma representación de antes, pero en fijándonos en dos momentos concretos que nos permitirá ver claramente el movimiento de la especie, en agosto y diciembre.

```
df_grulla_08 = df_grulla.filter(col("MONTH") == 8)
```


esri España
THE SCIENCE OF WHERE

Análisis de la distribución de cotorras argentinas y su relación con las grandes ciudades

Utilizaremos datos de eBird para ver la distribución de las cotorras argentinas a escala global y estatal para visualizar en qué zonas hay una mayor presencia de esta especie invasora. Los pasos que seguiremos son:

1. **Carga de bibliotecas.** Para hacer este análisis necesitaremos cargar las siguientes bibliotecas:

```
from pyspark.sql.functions import col, to_timestamp, year, month, to_date, desc
import matplotlib.pyplot as plt
from pyspark.sql import functions as F

import geoanalytics
import geoanalytics.sql.functions as ST

from geoanalytics.tools import AggregatePoints, SummarizeWithin, FindPointClusters,
FindHotSpots
```

2. **Iniciar sesión** en GeoAnalytics Engine y así tener acceso a todas las herramientas y funciones.

3. **Añadir los datos** que analizaremos. Están disponibles en formato CSV en la siguiente ruta: `s3://datos-tecnologia/ebird/CotorraArgentina.csv`

4. **Explorar y limpiar los datos.** Haciendo hincapié en los datos de geometría y tiempo.

5. **Representación de la distribución global de la especie**. Como en el caso anterior, agruparemos los datos para simplificar su visualización con la herramienta de agregación de puntos. Centrar el mapa sobre América del Sur para ver la distribución original.

6. **Cotorra argentina en España.** Filtraremos los datos globales para seleccionar solo los que pertenecen a España y los representaremos tras agregar los puntos.

```
df_spain = df_cotorra.filter(col("COUNTRY") == 'Spain')
```

7. **Relación entre grandes ciudades y mayor abundancia de individuos.** Las ciudades más grandes y mayor población suelen ser los hábitats preferidos para las cotorras, por esa razón vamos a relacionar los centros urbanos con los avistamientos de cotorra mediante la herramienta *Summarize within*. Los centros urbanos están disponibles en esta capa: https://services1.arcgis.com/nCKYwcSONQTkPA4K/arcgis/rest/services/Poblaciones_de_Espa%C3%B1a/FeatureServer/0
   El resultado de este planteamiento es interesante verlo en un gráfico de barras y en un mapa con una rampa de color al ser una variable continua. Centra el mapa en la ciudad que más cotorras hay para verlo mejor.

```
url =
"https://services1.arcgis.com/nCKYwcSONQTkPA4K/arcgis/rest/services/Poblaciones_de_Espa%
C3%B1a/FeatureServer/0"
poblaciones_df = spark.read.format("feature-service").load(url)

result = SummarizeWithin() \
           .setSummaryPolygons(poblaciones_df) \
           .includeShapeSummary(include=True, units="Kilometers") \
           .run(dataframe=df_spain)
```

8. **Puntos calientes.** Como conclusión del punto anterior veremos que las cotorras no se reparten uniformemente, utiliza la herramienta de puntos calientes para ver esas zonas con mayor cantidad de cotorras.

```
result = FindHotSpots() \
            .setBins(bin_size=800, bin_size_unit="Meters") \
            .setNeighborhood(distance=5000, distance_unit="Meters") \
            .run(dataframe=df_spain)

result_plot = result.st.plot(cmap_values="Gi_Bin",
                             cmap="Wistia",
                             legend=True,
                            basemap="dark",
                             figsize=(15,15))

%matplot plt
```

9. **Agrupación de poblaciones.** Por último, se emplea la herramienta *Find Point Clusters* para identificar los principales núcleos poblacionales de cotorra argentina, así como detectar poblaciones aisladas**.**

```
result = FindPointClusters() \
            .setClusterMethod(cluster_method="HDBSCAN") \
            .setMinPointsCluster(min_points_cluster=3000) \
            .run(dataframe=df_proyectadas)
```

Análisis de la asistencia a grandes conciertos en el Metropolitano (Madrid)

En este ejemplo vamos a analizar cómo afecta al distrito del Metropolitano la celebración de grandes conciertos. Para ello, compararemos días entre semana y fines de semana así como días con y sin concierto. Partiremos de un conjunto de datos descargado del portal de datos del Ministerio de Transporte y Movilidad Sostenible (MTMS) . Los pasos que seguiremos son:

1. **Carga de bibliotecas** y métodos necesarios para el análisis.

```
from pyspark.sql.functions import col
from pyspark.sql import functions as F
from pyspark.sql.types import IntegerType

import matplotlib.pyplot as plt

import geoanalytics
import geoanalytics.sql.functions as ST
from geoanalytics.tools import ReverseGeocode, AggregatePoints
```

2. **Iniciar sesión en ArcGIS GeoAnalytics Engine**.

3. **Iniciar sesión en ArcGIS Online** para tener acceso a datos de ArcGIS Living Atlas.

```
geoanalytics.register_gis(name="GIS",
                          url="https://arcgis.com",
                          username=usernameAGOL,
                          password=passwordAGOL )
```

4. **Carga de datos del MTMS.** Utilizaremos cinco datasets distintos para tener una representación de los diferentes casos.

```
df_lunes = spark.read.option("header", "true").option("delimiter", "|").csv("s3://datos-
tecnologia/UC_25/20240708_Viajes_distritos.csv")
df_jueves = spark.read.option("header", "true").option("delimiter",
"|").csv("s3://datos-tecnologia/UC_25/20240711_Viajes_distritos.csv")
df_sab = spark.read.option("header", "true").option("delimiter", "|").csv("s3://datos-
tecnologia/UC_25/20240713_Viajes_distritos.csv")

df_feid = spark.read.option("header", "true").option("delimiter", "|").csv("s3://datos-
tecnologia/UC_25/20240727_Viajes_distritos.csv")
df_metallica = spark.read.option("header", "true").option("delimiter",
"|").csv("s3://datos-tecnologia/UC_25/20240712_Viajes_distritos.csv")
```

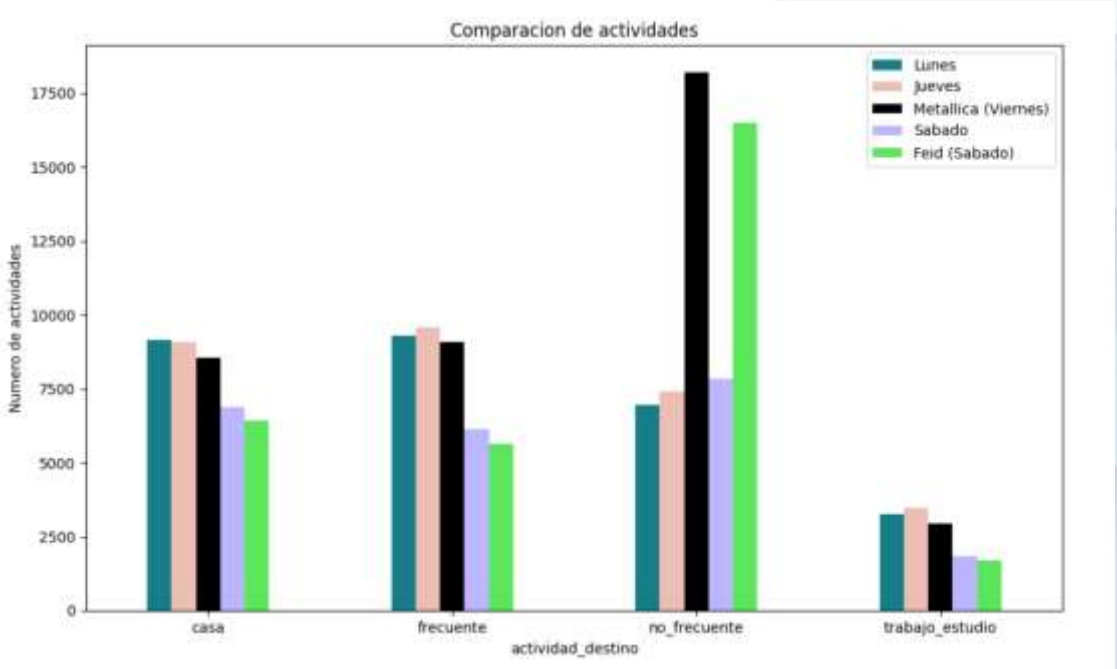
5. **Exploración y limpieza de datos.** Seleccionaremos los datos solo del distrito metropolitano (2807920) entre un rango determinado de horas.

```
def clean_df(df):
    metropolitano = "2807920"

    df = df.withColumn("hora_inicio", col("periodo").cast(IntegerType()))

    # Filtrar por destino
    df = df.filter(col("destino") == metropolitano)

    # Filtrar por rango de horas
    df = df.filter((col("hora_inicio") > 10) & (col("hora_inicio") < 21))

    return df
```

6. **Viajes al Distrito de Metropolitano.** Los datos del MTMS nos dan información sobre la actividad de destino llevada a cabo, utilizaremos esa variable *actividad_destino* para hacer una comparativa entre todos los días. La mejor forma de verlo es graficando esos resultados.

```
df_lunes_act_destino = df_lunes.groupBy("actividad_destino").count()
```

```
df_lunes_act_destino.show()

df_jueves_act_destino = df_jueves.groupBy("actividad_destino").count()
df_sab_act_destino = df_sab.groupBy("actividad_destino").count()
df_feid_act_destino = df_feid.groupBy("actividad_destino").count()
df_metallica_act_destino = df_metallica.groupBy("actividad_destino").count()

df_lunes_act_destino_pd = df_lunes_act_destino.toPandas().rename(columns={"count":
"Lunes"})
df_jueves_act_destino_pd = df_jueves_act_destino.toPandas().rename(columns={"count":
"Jueves"})
df_sab_act_destino_pd = df_sab_act_destino.toPandas().rename(columns={"count":
"Sabado"})
df_metallica_act_destino_pd =
df_metallica_act_destino.toPandas().rename(columns={"count": "Metallica"})
df_feid_act_destino_pd = df_feid_act_destino.toPandas().rename(columns={"count":
"Feid"})

merged = df_lunes_act_destino_pd.merge(df_jueves_act_destino_pd,
on="actividad_destino")\
            .merge(df_metallica_act_destino_pd, on="actividad_destino") \
            .merge(df_sab_act_destino_pd, on="actividad_destino") \
            .merge(df_feid_act_destino_pd, on="actividad_destino")

ax = merged.plot(
    x="actividad_destino",
    kind="bar",
    figsize=(12,7),
    rot=0,
    color=["#177E89", "#ECBEB4", "black", "#BCB6FF", "#5CE65C"]
)

plt.ylabel("Numero de actividades")
plt.title("Comparacion de actividades")
plt.legend(["Lunes", "Jueves", "Metallica (Viernes)", "Sabado",  "Feid (Sabado)"])
%matplot plt
```



7. **Residencia habitual de los asistentes.** Una vez comprobado que la celebración de este tipo de eventos aumenta el número de viajes no frecuentes al distrito pasaremos a analizar el origen de los asistentes del público; para ello nos deberemos quedar solo con esos viajes no frecuentes. Lo que tendremos que hacer será:
    a. Filtrar por el tipo de actividad para seleccionar solo la “no_frecuente”.
    ```
    df_metallica_no = df_metallica.filter("actividad_destino = 'no_frecuente'")
    ```
    b. Utilizando los códigos de provincia del INE, crear una columna nueva con la provincia de residencia utilizando el campo residencia de los datos del MTMS.

```
    ## Datos del INE
codigos_provincia = {
    "01": "Álava",
    "02": "Albacete",
    "03": "Alicante",
    "04": "Almería",
    "05": "Ávila",
    "06": "Badajoz",
    "07": "Islas Baleares",
    "08": "Barcelona",
    "09": "Burgos",
    "10": "Cáceres",
    "11": "Cádiz",
    "12": "Castellón",
    "13": "Ciudad Real",
    "14": "Córdoba",
    "15": "A Coruña",
    "16": "Cuenca",
    "17": "Girona",
    "18": "Granada",
    "19": "Guadalajara",
    "20": "Gipuzkoa",
    "21": "Huelva",
    "22": "Huesca",
    "23": "Jaén",
    "24": "León",
    "25": "Lleida",
    "26": "La Rioja",
    "27": "Lugo",
    "28": "Madrid",
    "29": "Málaga",
    "30": "Murcia",
    "31": "Navarra",
    "32": "Ourense",
    "33": "Asturias",
    "34": "Palencia",
    "35": "Las Palmas",
    "36": "Pontevedra",
    "37": "Salamanca",
    "38": "Santa Cruz de Tenerife",
    "39": "Cantabria",
    "40": "Segovia",
    "41": "Sevilla",
    "42": "Soria",
    "43": "Tarragona",
    "44": "Teruel",
    "45": "Toledo",
    "46": "Valencia",
    "47": "Valladolid",
    "48": "Bizkaia",
    "49": "Zamora",
    "50": "Zaragoza",
    "51": "Ceuta",
    "52": "Melilla"
}

mapping_expr = F.create_map([F.lit(x) for x in sum(codigos_provincia.items(), ())])
```

```
df_metallica_residencia = df_metallica_no.withColumn("provincia_residencia",
mapping_expr[F.lpad(df_metallica_no["residencia"].cast("string"), 2, "0")])
```

8. **Origen del viaje de los asistentes.** Ese dato nos lo proporciona el ministerio a través de la variable *origen* que tiene un código que podemos unir con el centro del distrito utilizando la capa alojada en ArcGIS Living Atlas: https://services1.arcgis.com/nCKYwcSONQTkPA4K/arcgis/rest/services/Zonificacion_centroides_MITMA/FeatureServer/1

   Por lo que tendremos que cargar los datos como DataFrame

```
centroide_path =
"https://services1.arcgis.com/nCKYwcSONQTkPA4K/arcgis/rest/services/Zonificacion_centroides_MITMA/FeatureServer/1"

df_centroide = spark.read.format("feature-service") \
                .option("gis", "GIS") \
                .load(centroide_path)
```

   Y unirlo a los datos originales:

```
df_metallica_origen = df_metallica_residencia.join(df_centroide,
df_metallica_residencia.origen == df_centroide.ID, "inner")
```

9. **Obtener la provincia de origen del viaje.** Como resultado de la operación anterior hemos obtenido el pueblo de origen y las coordenadas, pero no tenemos las provincias para compararlo con la provincia de residencia habitual. Para obtenerlas vamos a hacer una geocodificación inversa con la herramienta *ReverseGeocode*.

```
df_metallica_origen = df_metallica_origen_proy.filter(col("name").isNotNull())
df_metallica_origen_geo = ReverseGeocode() \
            .setLocator("Spain.mmpk") \
            .setOutFields("all") \
            .setLanguageCode("ES") \
            .setFeatureTypes("Locality") \
            .run(df_metallica_origen)
```

10. **Comparativa entre provincia de residencia y origen del viaje.** Haremos una gráfica para compararlo visualmente.

```
fig, axes = plt.subplots(1, 2, figsize=(18,6), sharey=True)

df_metallica_residencia_count =
df_metallica_residencia.groupBy("provincia_residencia").count().orderBy("count",
ascending=False)

df_metallica_residencia_pd = df_metallica_residencia_count.toPandas()
df_residencia = df_metallica_residencia_pd.head(10)
axes[0].bar(df_residencia["provincia_residencia"], df_residencia["count"],
color="black")
axes[0].set_title("Distribucion por provincia de residencia")
axes[0].set_xlabel("Provincia")
axes[0].set_ylabel("Número de viajes")
axes[0].tick_params(axis='x', rotation=90)

df_origen_viaje = df_metallica_spain_geocoded_pd.head(10)
axes[1].bar(df_origen_viaje["Subregion"], df_origen_viaje["count"], color="#8e8f88")
axes[1].set_title("Distribucion por origen del viaje (12/7/2024)")
axes[1].set_xlabel("Provincia")
axes[1].tick_params(axis='x', rotation=90)

plt.tight_layout()
%matplot plt
```

11. **Asistentes de Barcelona**. Representa en un mapa el origen del viaje de los asistentes cuya residencia habitual es Barcelona (residencia = 08).

12. **Representación en un mapa.** Para simplificar la visualización de los datos, utilizaremos la herramienta de agregación de puntos para representar en un mapa el origen del viaje de los asistentes al concierto.

13. **Extra**. Puedes realizar el mismo flujo con el concierto de Feid para ver algunas diferencias entre el público.

Análisis del choque entre dos buques en la Bahía de Algeciras

El 29 de agosto de 2022 hubo un choque entre dos buques en la Bahía de Algeciras que provocó el vertido de varias toneladas de combustible al océano. Vamos a utilizar datos de AIS para analizar qué pasó y qué rutas llevaban los dos buques implicados.

1. Carga de bibliotecas.
2. Inicio de sesión con ArcGIS GeoAnalytics Engine.
3. Acceso a los datos: s3a://ga-engine-esri-es/gibraltar/*.csv
4. Visualización de las ubicaciones de los barcos; también como agrupaciones de puntos.
5. Filtrado de datos de posicionamiento del 29 de agosto de 2022.
6. Reconstrucción de las rutas.
7. Encontrar ubicaciones permanentes con *FindDwellLocations*.
8. Ruta de los buques implicados en el choque OS 35 (572852210) y ADAM LNG (538005753).
9. Utilizar la herramienta *TraceProximityEvents* para identificar dónde se produjo el choque entre ambos.

Casos de uso

A. Geocodificación y creación de rutas

A partir de los datos de domicilio de reparadores y riesgos, se procederá a su geocodificación para obtener las coordenadas geográficas de ambos conjuntos. Posteriormente, se asignará a cada riesgo el reparador correspondiente utilizando el identificador común. Finalmente, se calculará la ruta entre ambos puntos para completar la relación entre reparadores y riesgos.

Los datos necesarios para este caso de uso son:

- Domicilios reparadores:
  `s3://ga-engine-esri-es/ClaseGAE_Enero/PoC_Anomalias_domicilios_reparadores.csv`
- Domicilios riesgos:
  `s3://ga-engine-esri-es/ClaseGAE_Enero/PoC_Anomalias_domicilios_riesgo.csv`

Los pasos a seguir son:

1. **Carga de bibliotecas** y funciones.

```
import json
from pyspark.sql.functions import array
import geoanalytics
from geoanalytics.sql import functions as ST
from pyspark.sql import functions as F
from geoanalytics.tools import Geocode, CreateRoutes
```

2. **Inicio de sesión** en ArcGIS GeoAnalytics Engine.

3. **Carga y exploración del localizador** que vamos a utilizar que tenemos en la ruta: `s3://ga-engine-esri-es/SMP/Spain.mmpk`

```
locator_path= "s3://ga-engine-esri-es/SMP/Spain.mmpk"
sc.addFile(locator_path)
geoanalytics.util.describe_locator(locator_path)
```

4. **Carga y geocodificación de los domicilios de los reparadores.**

```
reparadores_path = "s3://ga-engine-esri-
es/ClaseGAE_Enero/PoC_Anomalias_domicilios_reparadores.csv"
df_reparadores = spark.read.option("header", "true") \
                .option("sep", ";") \
                .option("inferSchema", "true") \
                .csv(reparadores_path)

result_reparadores = Geocode() \
            .setLocator("Spain.mmpk") \
            .setAddressFields("domicilio_reparador") \
            .setOutFields("Minimal")\
            .setCountryCode("ES")\
            .run(df_reparadores)
```

5. **Carga y geocodificación de los domicilios de los riesgos.** Los datos de los domicilios de los riesgos están en la ruta: `s3://ga-engine-esri-es/ClaseGAE_Enero/PoC_Anomalias_domicilios_riesgo.csv`

6. **Crear rutas entre reparadores y riesgos.** Para conocer qué reparador tiene asignado cada riesgo, deberemos unir ambos DataFrames mediante la variable “nencargo”. Además, como podemos ver en la documentación de la herramienta de *Create Routes*, las geometrías del inicio y final de la ruta debemos pasárselas como un array con ambos puntos.

```
df_rutas = result_reparadores.join(result_riesgos, on="nencargo", how="inner")
df_rutas = df_rutas.withColumn("stops", array("location_reparador", "location_riesgo"))
df_rutas = df_rutas.drop("location_reparador", "location_riesgo")

trip_routes = CreateRoutes()\
    .setNetwork("Spain.mmpk")\
    .setStops("stops")\
    .run(df_rutas)
```

7. **Compartir las primeras 5 rutas en ArcGIS Online.**

```
service_name = "rutas"
first_ten_routes.write.format("feature-service") \
        .option("gis", "myGIS") \
        .option("serviceName", service_name) \
        .option("layerName", "layer") \
        .option("tags", "rutas") \
        .option("description", "10 rutas creadas con GeoAnalytics Engine") \
        .save()
print("Datos guardado en ArcGIS Online")
```

B. Anomalías y facturación

Para completar el ejercicio anterior y facilitar la detección de posibles anomalías, se utilizará la distancia recorrida por cada reparador para representar los riesgos asignados mediante una rampa de color, diferenciando los más cercanos de los más lejanos.

Para este caso de uso necesitaremos las rutas creadas en el anterior caso de uso y los pasos a seguir serían:

1. Explorar las distancias de rutas.

2. Representar en un mapa las riesgos que no cumplan nuestro criterio.

C. Distribución de riesgos

Para comprobar si los riesgos se reparten de forma azarosa o hay una mayor concentración en alguna zona del país usaremos las herramientas de *FindHotSpot* y *CalculateDensity* para ver la distribución de los riesgos en España.

D. Repartidores más cercanos

Utilizando los datos de riesgos y repartidores, vamos a ver cuáles son los tres repartidores más cercanos a cada riesgo con la herramienta *FindClosestFacility*. Para completar el estudio, sacaremos las estadísticas medias para evaluar los tiempos y distancias medias.

E. Organización de rutas

Vamos a seleccionar las cuatro primeras direcciones de los datos de riesgos y vamos a suponer que los tendría que hacer un único reparador. Vamos a utilizar la herramienta de creación de rutas para:

- Obtener la ruta respetando el orden del conjunto de datos.
- Obtener la ruta optimizada entre esos puntos.

F. Áreas de servicio

A partir de los datos de reparadores crearemos un área de servicio para ver a qué zonas pueden llegar en menos de 30 y 60 minutos.

G. Distribución del mercado

A partir de unos datos simulados de venta, vamos a analizar la demanda por territorio. Para ello, a parte de los datos de ventas, tendremos que usar los polígonos administrativos como son de comunidades autónomas y provincias para obtener estadísticas a ese nivel de detalle. Si queremos ampliar el detalle, podemos usar también los datos de secciones censales.

Los pasos que debemos seguir son:

1. Cargar bibliotecas
2. Iniciar sesión en ArcGIS GeoAnalytics Engine
3. Cargar datos de ventas simuladas: s3://datossantalucia/resultados/incidencias_3857.shp
4. Cargar datos de comunidades autónomas, provincias o secciones censales del Living Atlas.
5. Agregar los datos con la herramienta *SummarizeWithin*

Recursos

- Documentación: https://developers.arcgis.com/geoanalytics/
- Tutoriales: https://developers.arcgis.com/geoanalytics/tutorials/
- API Reference: https://developers.arcgis.com/geoanalytics/api-reference/index.html
- Vídeos de ArcGIS GeoAnalytics Engine: https://mediaspace.esri.com/search?keyword=Arcgis%20geoanalytics%20engine
- Glosario de terminología GIS: https://developers.arcgis.com/documentation/glossary/
- GeoAnalytics Engine Blog: https://community.esri.com/t5/geoanalytics-engine-blog/bg-p/geoanalytics-engine-blog
- Esri Community: https://community.esri.com/t5/geoanalytics-engine-questions/bd-p/geoanalytics-engine-questions
- GitHub ArcGIS GeoAnalytics Engine samples: https://github.com/Esri/geoanalytics-engine-samples/tree/main
- ArcGIS GeoAnalytics Engine en Databricks: https://www.databricks.com/blog/2022/12/07/arcgis-geoanalytics-engine-databricks.html
- ArcGIS GeoAnalytics Engine en Amazon EMR: https://aws.amazon.com/es/blogs/apn/big-data-analytics-with-amazon-emr-and-esri-arcgis-geoanalytics-engine/
- ArcGIS GeoAnalytics Engine en Azure Synapse Analytics: https://techcommunity.microsoft.com/t5/azure-synapse-analytics-blog/spatial-analysis-in-azure-synapse-analytics-with-arcgis/ba-p/3728969