MEMORIA PROYECTO INGENIERÍA Y CIENCIA DE DATOS II

Grupo 6

Iván José Alba García

Yasser Aoujil

Joaquín Trillo Escribano

ÍNDICE

PALABRAS CLAVES

MongoDB, Neo4j, Cassandra, NoSQL, Python, Pymongo, Tableau, Clustering, Clasificación, Limpieza y carga de datos (en todo el documento LC), Pandas, Dataframe, Cypher, aggregate, MapReduce.

LISTA DE FIGURAS

LISTA DE TABLAS

1. Introducción

El mundo de los negocios está experimentando un cambio masivo a medida que industria tras otra se están trasladando a la economía digital. Es una economía impulsada por Internet y otras tecnologías del siglo XXI: la nube, los dispositivos móviles, las redes sociales y el big data. En el corazón de cada negocio de la Economía Digital se encuentran sus aplicaciones web, móvil e Internet de las Cosas (IoT), estas tecnologías son la principal forma en que las empresas interactúan con los clientes actuales y cómo las empresas administran cada vez más sus negocios. Las experiencias que las empresas entregan a través de esas aplicaciones determinan en gran medida la satisfacción y la lealtad de los clientes.

2. Contextualización del trabajo

Las aplicaciones web, móviles e IoT actuales comparten una o más (si no todas) de las siguientes características, y que también las diferencian de las aplicaciones convencionales:

- Admite grandes cantidades de usuarios simultáneos (decenas de miles, quizás millones).
- Entregar experiencias altamente responsiva y fluida a una base de usuarios distribuida globalmente.
- Estar siempre disponible, sin tiempo de inactividad.
- Manejar datos semi y no estructurados.
- Rápidamente adaptarse a los requisitos cambiantes con actualizaciones frecuentes y nuevas características.

La creación y ejecución de estas aplicaciones web, móviles e IoT ha creado un nuevo conjunto de requisitos tecnológicos. La nueva arquitectura de tecnología empresarial necesita ser mucho más ágil que nunca, y requiere un enfoque de administración de datos en tiempo real que pueda acomodar niveles sin precedentes de escalabilidad, velocidad y variabilidad de datos. Las bases de datos relacionales no pueden cumplir con estos nuevos requisitos y, por lo tanto, las empresas recurren a la tecnología de bases de datos NoSQL.

Un principio básico hoy día en los desarrollos de aplicaciones es adaptarse a los requisitos de aplicación que evolucionan, de forma que cuando cambien los requisitos, el modelo de datos también cambia. Este es un problema para las bases de datos relacionales porque el modelo de datos es fijo y está definido por un esquema estático. Entonces, para cambiar el modelo de datos, los desarrolladores tienen que modificar el esquema o, lo que es peor, solicitar un "cambio de esquema" a los administradores de la base de datos. Esto ralentiza o detiene el desarrollo, no solo porque es un proceso manual y lento, sino que también afecta otras aplicaciones y servicios.

En comparación, una base de datos de documentos NoSQL es totalmente compatible con el desarrollo ágil, ya que no tiene esquemas y no define de manera estática cómo se deben modelar los datos. En cambio, difiere a las aplicaciones y servicios, y por lo tanto a los desarrolladores en cuanto a cómo se deben modelar los datos. Con NoSQL, el modelo de datos está definido por el modelo de la aplicación. Las aplicaciones y servicios modelan los datos como objetos.

Por todo lo explicado, en este trabajo se ha optado por explorar tres tipos de bases de datos NoSQL:

- Mongodb: base de datos orientada a documento
- Neo4j: base de datos basada en grafo para datos fuertemente relacionados
- Cassandra: base de datos distribuida y orientada a columnas

En nuestro trabajo también hemos explorado las posibles herramientas que existen para trabajar con los sistemas de bases de datos anteriormente mencionados, a nivel de gestión, explotación, tratamiento y visualización.

3. Herramientas de desarrollo y librerías involucradas

3.1. Python

Python [1] es un lenguaje de scripting de alto nivel, interpretado, interactivo y orientado a objetos. Python está diseñado para ser altamente legible. Utiliza palabras clave en inglés con frecuencia cuando otros idiomas usan signos de puntuación, y tiene menos construcciones sintácticas que otros idiomas.

- Python es interpretado - Python es procesado en tiempo de ejecución por el intérprete. No necesita compilar su programa antes de ejecutarlo. Esto es similar a PERL y PHP.
- Python es interactivo: usando la consola de Python, se puede interactuar directamente con el intérprete para escribir los programas.
- Python está orientado a objetos: Python es compatible con el estilo orientado a objetos o la técnica de programación que encapsula el código dentro de los objetos.
- Python es un lenguaje para principiantes: Python es un excelente lenguaje para los programadores de nivel principiante y es compatible con el desarrollo de una amplia gama de aplicaciones, desde el simple procesamiento de texto hasta desarrollo web, juegos y mucho más.

3.2. Pandas

Pandas [2] es una biblioteca de código abierto Python, que proporciona estructuras de datos y herramientas de análisis de datos de alto rendimiento y fácil de usar para el lenguaje de programación Python.
Python junto con Pandas se utiliza en una amplia gama de campos, incluidos los dominios académicos y comerciales, que incluyen finanzas, economía, estadística, análisis, etc.

Las características más importantes de pandas son:

- Objeto DataFrame rápido y eficiente con indexación predeterminada y personalizada.
- Herramientas para cargar datos en objetos de datos en memoria desde diferentes formatos de archivo.
- Alineación de datos y manejo integrado de datos incompletos.
- Remodelación y formateo de conjuntos de fechas.
- Etiquetado de corte, indexación y subconjunto de grandes conjuntos de datos.
- Las columnas de una estructura de datos se pueden eliminar o insertar.
- Agrupar los datos para agregación y transformaciones.
- Alto rendimiento de fusión y unión de datos.
- Funcionalidad de la serie de tiempo

Un Dataframe es una estructura de datos bidimensional, es decir, los datos se alinean de forma tabular en filas y columnas. Este concepto es muy importante entenderlo, ya que los Dataframes se van a usar de forma intensiva en Python al recuperar información de las bases de datos.

La figura siguiente ilustra cómo se estructura un Dataframe:

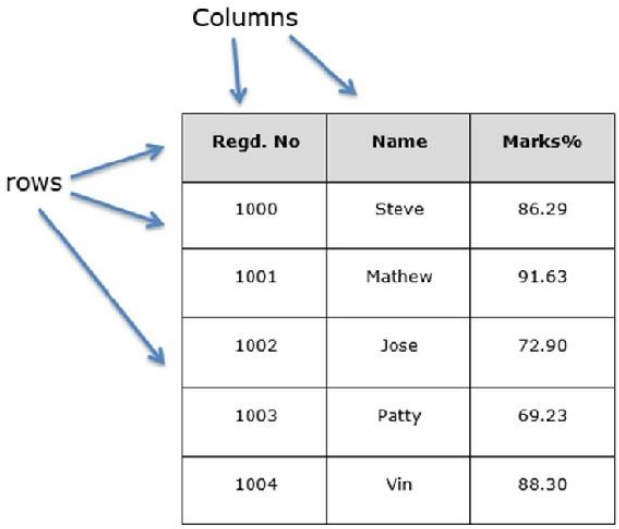


Figura 1. Estructura Dataframes en pandas.

3.3. Conjunto de módulos Python usados en todo el proyecto

La Tabla 1 muestra los módulos de Python más importantes usados en este proyecto.

Módulo	Descripción
neo4j-driver	Driver de Python para neo4j, permite interactuar con neo4j desde Python.
pymongo	Driver de Python para MongoDB, permite interactuar con MongoDB desde Python.
cassandra-driver	Driver de Python para Cassandra, permite interactuar con Cassandra desde Python.
matplotlib	Librería para generación de gráficos.
pandas	Librería para tratamiento de datos.
folium	Librería para generar mapas usando datos geoespaciales.
numpy	Librería para computación científica.
sklearn	Librería de aprendizaje computacional.

Tabla 1. Principales módulos Python usados.

4. Bases de datos

4.1. MongoDB

4.1.1. Conceptos generales

MongoDB es una base de datos multiplataforma y orientada a documentos que ofrece alto rendimiento, alta disponibilidad y escalabilidad fácil. MongoDB trabaja con el concepto de colección y documento. A continuación, vamos a definir los términos específicos de esta base de datos.

- **Base de datos:** La base de datos es un contenedor físico para colecciones. Cada base de datos obtiene su propio conjunto de archivos en el sistema de archivos. Un único servidor MongoDB generalmente tiene múltiples bases de datos.
- **Colección**: es un grupo de documentos MongoDB. Es el equivalente de una tabla RDBMS. Una colección no puede repetirse dentro de una única base de datos. Las colecciones no siguen un esquema fijo como pasa en las tablas relacionales. Los documentos dentro de una colección pueden tener diferentes campos. Por lo general, todos los documentos en una colección tienen un propósito similar o relacionado.
- **Documento:** Un documento es un conjunto de pares clave-valor en formato BSON. Los documentos tienen un esquema dinámico que significa que los documentos en la misma colección no necesitan tener el mismo conjunto de campos o estructura, y los campos comunes en los documentos de una colección pueden contener diferentes tipos de datos, de este dinamismo proviene el hecho de que no se sigue un esquema fijo.

En la Tabla 2 se muestra un ejemplo de cómo se corresponden los conceptos en las bases de datos relacionales (RDBMS) con los de MongoDB:

RDBMS	MongoDB
Base de datos	Base de datos
Tabla	Colección
Tupla/Registro/Fila	Documento
Columna	Campo
Clave primaria	Clave primaria de cada documento "_id", generada por mongoDB automáticamente.
Join de tablas	Documentos anidados

Tabla 2. Correspondencias entre conceptos RDBMS con los de MongoDB.

MongoDB como cualquier otro sistema de bases de datos, se compone de al menos un servidor y un cliente. El servidor de MongoDB se ejecuta usando el comando “mongod”, para el cliente depende de la implementación.
En este proyecto hemos optado por desplegar MongoDB en local en un sistema OSx.

4.1.2. Driver de MongoDB para Python

PyMongo es una distribución de Python que contiene herramientas para trabajar con MongoDB. Desarrollado por los propios creadores de MongoDB, es la forma más recomendada de trabajar con MongoDB desde Python. La Figura 2 muestra la arquitectura que enlaza el lenguaje Python con MongoDB usando el driver pymongo.

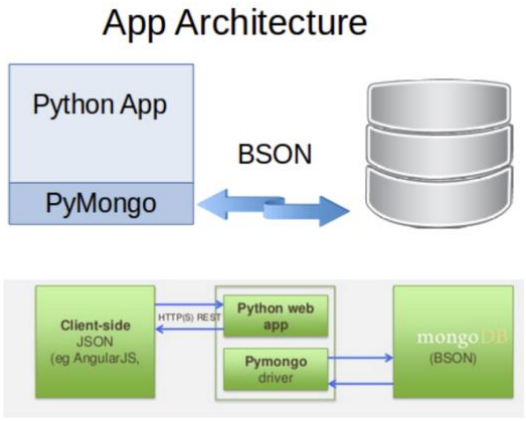


Figura 2. Arquitectura de conexión entre PyMongo y MongoDB.

La comunicación se hace a través de la conversión de tipos de MongoDB hacia los de Python y viceversa usando los objetos BSON (Binay json). En la Figura 3 se ve la correspondencia entre tipos y objetos de Python y los de MongoDB.

Python Type	BSON Type	Supported Direction
None	null	both
bool	boolean	both
int [1]	int32 / int64	py -> bson
long	int64	py -> bson
bson.int64.Int64	int64	both
float	number (real)	both
string	string	py -> bson
unicode	string	both
list	array	both
dict / *SON*	object	both
datetime.datetime [2] [3]	date	both
bson.regex.Regex	regex	both
compiled re [4]	regex	py -> bson
bson.binary.Binary	binary	both
bson.objectid.ObjectId	oid	both
bson.dbref.DBRef	dbref	both
None	undefined	bson -> py
unicode	code	bson -> py
bson.code.Code	code	py -> bson
unicode	symbol	bson -> py
bytes (Python 3) [5]	binary	both

Figura 3. Correspondencia de tipos entre Python y MongoDB.

El driver pymongo se encarga de las funcionalidades que aparecen en la siguiente lista, todas ya están implementadas en el módulo, además todas esas funcionalidades tienen una API que facilita su uso de forma intuitiva y flexible. La API de Pymongo tiene una implementación para una variedad de lenguajes de programación.

1. Autenticación y seguridad
2. Conversión de Python<=>BSON
3. Manejo de errores o excepciones y de las recuperaciones
4. Gestión del protocolo Wire de MongoDB, este protocolo facilita la comunicación con MongoDB a través de Sockets TCP/IP
5. Gestión de la topología de la base de datos
6. Gestión del pool de conexiones

En los siguientes apartados veremos el driver en práctica y todo el potencial que ofrecen juntos Python y MongoDB.

4.1.3. Proceso de carga de datos

4.1.3.1. Proceso LC

En este apartado se explicarán los pasos seguidos para cargar los datos en una base de datos MongoDB.

El proceso ha pasado por varias fases, la primera es el análisis de los diferentes tipos de fichero que ofrece la página de datos públicos de la ciudad de San Francisco, para poder elegir el que mejor se adapte. Los principales tipos son json y csv, este último ha sido el elegido por el tamaño reducido que tiene en comparación con json.

En segundo lugar, se ha realizado una limpieza del dataset, haciendo los cambios necesarios en las columnas que no estén en un formato adecuado para poder explotar su contenido. Por último, se ha realizado la inserción de los datos en la base de datos.

Como una ampliación se ha descargado un dataset[3] que contiene la división de la ciudad de San Francisco en polígonos.

En el apartado siguiente se va a detallar el proceso LC sobre el dataset de incidencias en la base de datos MongoDB.

Se ha usado el lenguaje de programación *Python* para realizar este proceso por la rapidez de desarrollo y la facilidad de manejo de grandes cantidad datos que permite usando la librería *Pandas*.

El proceso de LC se ha realizado en el mismo fichero Python *process_insert.py*, a continuación, se explican las partes más importantes del código que se ve en la figura X.

La función *import_content* en la línea 31 es la que arranca el proceso LC, recibe la ruta del fichero csv que contiene el dataset. Esta función realiza llamadas a otras funciones auxiliares.

En primer lugar, se crear una conexión a servicio de Mongodb local, y en la línea 33 creamos la base de datos a la que llamamos *san_francisco_incidents*.

El siguiente paso es crear una colección que representara los documentos de las incidencias en la base de datos. En este caso, hemos usado una colección para representar las incidencias,

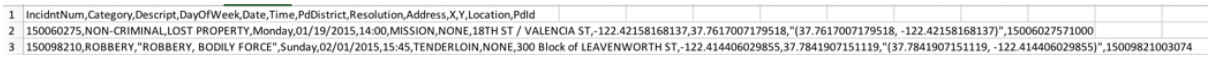
además de una colección para representar los polígonos que componen los distritos de San Francisco, esta colección se explicará más adelante en último capítulo de este documento.
En la línea 36, usando la librería *panda*, leemos el fichero entero en memoria usando la función *read_csv* que devuelve un dataframe con todo el dataset.
A continuación, procedemos con la limpieza del dataframe, empezando con las columnas X e Y representando la longitud y latitud consecutivamente. En las líneas 37 y 38 se llama a la función auxiliar *parse_float* que recibe una variable y la convierte a float.
Luego pasamos a transformar la columna de localización (Location) que está en formato string y que normalmente no es lo más adecuado para poder realizar consultas geoespaciales sobre mongodb. Entonces en la línea 40 se llama a la función *parse_location* pasándolo cada fila del dataframe, y convierte el contenido de la columna Location de formato string a formato *GeoJSON* en la Figura 6 se ve la columna Location con el nuevo formato.
Una parte muy importante para el análisis es el tiempo. En este dataset, existen dos columas, Date y Time, que contiene por separado la fecha y la hora del incidente. Nos conviene mejor tener estos dos datos en conjunto por lo que usamos la función auxiliar *parse_complete_date* en la línea 42 que recibe una del dataframe y cambia en ella el campo Date, haciendo que contenga toda la información del tiempo del incidente, en la Figura 6 podemos ver cómo queda reflejada toda esa información en el mismo campo.

```
8  # This function converts each argument to a float value, in case of Exception returns a 0
9  def parse_float(x):
10     try:
11         x = float(x)
12     except Exception:
13         x = 0
14     return x
15 # This function receives a row of the dataframe and converts its date cell to a well formatted datetime object
16 # using the column Date and Time.
17 def parse_complete_date(s):
18     date = datetime.strptime(s["Date"], "%m/%d/%Y")
19     time = s["Time"].split(":")
20     date_time = date.replace(hour=int(time[0]), minute=int(time[1]))
21     return date_time
22 # This function receives a row from the dataframe
23 def parse_location(s):
24     lat = s["Y"]
25     log = s["X"]
26     location_dict = {
27         'coordinates': [log, lat],
28         'type': 'point'
29     }
30     return location_dict
31 def import_content(file_path):
32     client = MongoClient()
33     db = client['san_francisco_incidents1']  # Creates the database for the San Francisco city incidents data
34     incid = db.incidents        # Creates the collection of the documents of that will represent the incidents
35     print("Reading csv file...")
36     df = pd.read_csv(file_path)  # Reads the csv file into python's dataframe type (in my case I named it incid.csv)
37     df["X"] = df["X"].apply(parse_float)  # We make sure the fields X and Y are of type float
38     df["Y"] = df["Y"].apply(parse_float)
39     print("Parsing location...")
40     df["Location"] = df.apply(parse_location, axis=1)  # Parse location into a well-formatted geojson object
41     print("Parsing date...")
42     df["Date"] = df.apply(parse_complete_date, axis=1)  # Axis=1 to interate over the rows applying parse_complete_date to each one
43     print("Inserting data into monogodb. Might take some minutes, please be patient...")
44     incid.insert_many(df.to_dict("records"))  # Insert many python objects into the collection incidents
45     print("Done :D")
```

Figura 4. Función que recibe la ruta del fichero del dataset y realiza el proceso LC.

4.1.3.2. Resultados de LC

Después de este proceso de limpieza y carga, el resultado que se ha obtenido es que hemos pasado de tener los datos representados en el formato que se ve en la Figura 5 al formato en la Figura 6, que tiene mucha más flexibilidad y posibilidades de explotación para extraer información.



```
1 IncidntNum,Category,Descript,DayOfWeek,Date,Time,PdDistrict,Resolution,Address,X,Y,Location,PdId
2 150060275,NON-CRIMINAL,LOST PROPERTY,Monday,01/19/2015,14:00,MISSION,NONE,18TH ST / VALENCIA ST,-122.42158168137,37.7617007179518,"(37.7617007179518, -122.42158168137)",15006027571000
3 150098210,ROBBERY,"ROBBERY, BODILY FORCE",Sunday,02/01/2015,15:45,TENDERLOIN,NONE,300 Block of LEAVENWORTH ST,-122.414406029855,37.7841907151119,"(37.7841907151119, -122.414406029855)",15009821003074
```

Figura 5. Dos entradas de incidencias en formato csv.

```
_id: ObjectId("5ac1522e790de03ca663d972")
IncidntNum: 150060275
Category: "NON-CRIMINAL"
Descript: "LOST PROPERTY"
DayOfWeek: "Monday"
Date: 2015-01-19 15:00:00.000
Time: "14:00"
PdDistrict: "MISSION"
Resolution: "NONE"
Address: "18TH ST / VALENCIA ST"
X: -122.42158168136999
Y: 37.7617007179518
Location: Object
  coordinates: Array
    0: -122.42158168136999
    1: 37.7617007179518
  type: "point"
PdId: 15006027571000

_id: ObjectId("5ac1522e790de03ca663d973")
IncidntNum: 150098210
Category: "ROBBERY"
Descript: "ROBBERY, BODILY FORCE"
DayOfWeek: "Sunday"
Date: 2015-02-01 16:45:00.000
Time: "15:45"
PdDistrict: "TENDERLOIN"
Resolution: "NONE"
Address: "300 Block of LEAVENWORTH ST"
X: -122.414406029855
Y: 37.7841907151119
Location: Object
  coordinates: Array
    0: -122.414406029855
    1: 37.7841907151119
  type: "point"
PdId: 15009821003074
```

Figura 6. los mismos dos registros cambiados a documentos de Mongodb.

Un punto interesante en las capturas anteriores que hemos notado es el cambio en el tiempo en el documento en la Figura 6. Como se puede ver que la hora de incidente son 14:00, pero en el campo Date nos sale que es las 15:00. La explicación es que programa usado para mostrar esos datos, Mongodb Compass, añade una hora por la zona horaria en la que estamos.

En realidad, el campo Date es correcto a nivel de base de datos y eso se comprueba con la siguiente captura que muestra una consulta para obtener un documento, y que devuelve el primero que se corresponde con el documento de la Figura 6 en la izquierda.

```
> use san_francisco_incidents
switched to db san_francisco_incidents
> db.incidents.findOne()
{
        "_id" : ObjectId("5ac1522e790de03ca663d972"),
        "IncidntNum" : 150060275,
        "Category" : "NON-CRIMINAL",
        "Descript" : "LOST PROPERTY",
        "DayOfWeek" : "Monday",
        "Date" : ISODate("2015-01-19T14:00:00Z"),
        "Time" : "14:00",
        "PdDistrict" : "MISSION",
        "Resolution" : "NONE",
        "Address" : "18TH ST / VALENCIA ST",
        "X" : -122.42158168136999,
        "Y" : 37.7617007179518,
        "Location" : {
                "coordinates" : [
                        -122.42158168136999,
                        37.7617007179518
                ],
                "type" : "point"
        },
        "PdId" : NumberLong("15006027571000")
}
```

Figura 7. Consulta para obtener un documento (el primero por defecto).

4.1.3.3. Dataset de ampliación de los distritos de San Francisco

En este apartado se explicará el proceso seguido para añadir a la base de datos una nueva colección que va a representar los diferentes distritos de la ciudad de San Francisco [3].
El objetivo de esta ampliación es poder realizar consultas geoespaciales que permiten tener información más detallada y exacta sobre los incidentes.

- **Análisis del dataset**

  Esta vez se ha usado un fichero json en vez de un csv. Por lo que se ha desarrollado un nuevo script para realizar el proceso de transformación y carga de datos.
  En primer lugar, se ha analizado detenidamente el contenido del json, y se ha logrado identificar las claves en el json que se necesitan para realizar las siguientes tareas:

  1. Obtener los nombres de los campos que formaran el documento en la colección de Mongodb.
  2. Obtener los datos.
  3. Fusionar la lista de nombres de campos y la lista de datos en un dataframe de la librería pandas.
  4. Cargar el dataframe en la base de datos después de realizar formateos si hacen falta.

  La Figura 8 muestra el código usado, nos basaremos sobre ella para explicar los pasos.

```python
import pandas as pd
import ijson
from shapely.geometry import mapping
from shapely.wkt import loads
from pymongo import MongoClient


# Convierte datos geoespaciales en formato wkt a geojson, ya que mongodb no admite wkt
def wkt_to_geojson(wkt):
    wkt_multipol = loads(wkt)
    gj_multipol = mapping(wkt_multipol)
    return gj_multipol


# Extrae los nombres de columnas del json, y extrae los datos
def import_content(path):
    # Extract the names of the columns from the json
    with open(file_path, 'r') as f:
        objects = ijson.items(f, 'meta.view.columns.item')
        columns = list(objects)
    column_names = [col["fieldName"] for col in columns]
    # Extract the data from the json
    final_columns = ['the_geom', 'nhood']
    data = []
    with open(path, 'r') as f:
        objects = ijson.items(f, 'data.item')
        for row in objects:
            selected_row = []
            for item in final_columns:
                selected_row.append(row[column_names.index(item)])
            data.append(selected_row)
    # We convert to pandas dataframe to ease the proces
    df = pd.DataFrame(data, columns=final_columns)
    df["the_geom"] = df["the_geom"].apply(wkt_to_geojson)
    return df


# Inserta en la base de datos todas las líneas del dataframe ne forma de documentos json
def mongo_insert(df):
    client = MongoClient()
    db = client['san_francisco_incidents']
    neighbours = db.neighbours
    neighbours.insert_many(df.to_dict("records"))


if __name__ == "__main__":
    file_path = "/Users/yasos/Downloads/rows.json"
    df = import_content(file_path)
    mongo_insert(df)
```

Figura 8. Código para procesar e insertar el dataset de distritos de San Francisco.

- **Obtención de los nombres de campos**

Viendo la estructura del json, Figura 9, se ha visto que la clave “columns” dentro de la jerarquía del json, contiene un array de atributos. Entre estos atributos encontramos el campo fieldName que contiene el nombre del campo.

Entre las líneas 20 y 25, leemos el fichero, extraemos de cada elemento del array de columnas, el nombre del campo.

En nuestro caso, nos interesan sólo dos campos, él de los datos de geolocalización y él del nombre del distrito, por lo que definimos una lista (línea 25), con los nombres de esos campos, esa lista nos servirá más adelante para extraer solo los datos que nos interesan.


```
{
  "meta" : {
    "view" : {
      "id" : "xfcw-9evu",
      "name" : "San Francisco Analysis Neighborhoods",
      "averageRating" : 0,
      "createdAt" : 1432059760,
      "displayType" : "geoRows",
      "downloadCount" : 952,
      "hideFromCatalog" : false,
      "hideFromDataJson" : false,
      "indexUpdatedAt" : 1502157120,
      "newBackend" : true,
      "numberOfComments" : 0,
      "oid" : 23391098,
      "provenance" : "official",
      "publicationAppendEnabled" : false,
      "publicationDate" : 1472169502,
      "publicationGroup" : 5806175,
      "publicationStage" : "published",
      "rowsUpdatedAt" : 1472169502,
      "rowsUpdatedBy" : "s7k8-df3k",
      "tableId" : 12734925,
      "totalTimesRated" : 0,
      "viewCount" : 3,
      "viewLastModified" : 1494383183,
      "viewType" : "tabular",
      "childViews" : [ "" ],
      "columns" : [ {
        "id" : -1,
        "name" : "sid",
        "dataTypeName" : "meta_data",
        "fieldName" : ":sid",
        "position" : 0,
        "renderTypeName" : "meta_data",
        "format" : { },
        "flags" : [ "hidden" ]
      }, {
```


Figura 9. json del dataset de distritos.

4.1.3.4. Dataframe, transformaciones y carga en base de datos.

Entre las líneas 27 y 35, se procesa el fichero json para recuperar los valores de los dos campos que nos interesan del dataset (the_geom, nhood).

En la línea 36 realizamos una transformación sobre el campo the_geom que contiene los polígonos que forman los distritos de la ciudad, esa transformación consiste en convertir los datos del formato WKT a GeoJson usando la liberaría *shapely* de Python. La transformación se realiza sobre cada fila del dataframe usando la función auxiliar wkt_to_geojson.

Comparando las Figuras 10 y 11 se puede ver la diferencia en el formato de ese campo.

Al realizar las transformaciones, la inserción en Mongodb es inmediata, convirtiendo el dataframe a un diccionario de Python, como se ve en la línea 45.

```
MULTIPOLYGON (((-122.38157774241415 37.75307043091241, -122.38156949251606 37.753060959298274,
-122.38159239626694 37.75309424492931, -122.38155614326205 37.753045901366754, -122.38155137472305
37.75304127677009, -122.38154650687193 37.753036719547374, -122.3815415385967 37.75303223061754,
-122.38153647334677 37.75302781172802, -122.38153131112232 37.75302346287867, -122.38152605423795
37.753019185835115, -122.38152070389688 37.75301498328154, -122.38151526236827 37.75301085518192,
-122.38150973196667 37.75300680330162, -122.38150411158058 37.75300282855954, -122.38149840577098
37.7529989317842, -122.38149261460651 37.75299511567804, -122.38148674033339 37.75299137930393,
-122.38148078528907 37.75298772532827, -122.38147475063121 37.75298415463372, -122.38146863865168
37.752980668084966, -122.38146245050791 37.752977266564635, -122.38145618849175 37.752973950937466,
-122.38144985494098 37.752970723869645, -122.38144345094439 37.752967583541384, -122.38143697999709
37.7529645335017...
```

Figura 10. the_geom en formato wkt.

df

	the_geom	nhood
0	{'type': 'MultiPolygon', 'coordinates': [[[[-122.38157774241415, 37.75307043091241), (-122.38156949251606, 37.7530...	Bayview Hunters Point
1	{'type': 'MultiPolygon', 'coordinates': [[[[-122.40361299982803, 37.74933700015653), (-122.40378100004523, 37.748...	Bernal Heights
2	{'type': 'MultiPolygon', 'coordinates': [[[[-122.42655500055683, 37.769484999847016), (-122.42694800056495, 37.7...	Castro/Upper Market
3	{'type': 'MultiPolygon', 'coordinates': [[[[-122.4062259995664, 37.79755900029376), (-122.40551299953815, 37.797...	Chinatown
4	{'type': 'MultiPolygon', 'coordinates': [[[[-122.42398200023331, 37.731551999755176), (-122.42391999958627, 37.73...	Excelsior
5	{'type': 'MultiPolygon', 'coordinates': [[[[-122.3875252162534, 37.78279734438729), (-122.3875205579595, 37.78279...	Financial District/South Beach
6	{'type': 'MultiPolygon', 'coordinates': [[[[-122.44737500017301, 37.74648199962739), (-122.44728499988267, 37.746...	Glen Park
7	{'type': 'MultiPolygon', 'coordinates': [[[[-122.45932399954084, 37.78752100021499), (-122.45927999992254, 37.78...	Inner Richmond
8	{'type': 'MultiPolygon', 'coordinates': [[[[-122.44092200041324, 37.773634999987), (-122.44073499971395, 37.77271...	Golden Gate Park
9	{'type': 'MultiPolygon', 'coordinates': [[[[-122.43199799957345, 37.771430999654044), (-122.4319039996054, 37.770...	Haight Ashbury
10	{'type': 'MultiPolygon', 'coordinates': [[[[-122.4208080001382, 37.77399700010538), (-122.42108899962406, 37.7737...	Hayes Valley
11	{'type': 'MultiPolygon', 'coordinates': [[[[-122.45294699979522, 37.76637399965159), (-122.4527579997282, 37.765...	Inner Sunset
12	{'type': 'MultiPolygon', 'coordinates': [[[[-122.42468700019211, 37.785335000403094), (-122.42489900016167, 37.78...	Japantown
13	{'type': 'MultiPolygon', 'coordinates': [[[[-122.40666500014113, 37.719214999860796), (-122.40698199987852, 37.718...	McLaren Park

Figura 11. Dataframe resultante al final de las transformaciones, the_geom en formato geojson.

4.1.4. Algunas consultas con Python

En este bloque vamos a ver algunas de las consultas más importantes que se han realizado contra la base de datos de MongoDB.
Los apartados siguientes se organizan de forma que el propio código explique el objetivo de la consulta, además se muestran consultas realizadas con Python usando el driver de pymongo y otras consultas realizadas directamente en el Shell de mongo.

4.1.4.1. Creación de la sesión con MongoDB

```
def get_session(db_name):
    """
    :param db_name: Nombre de la base de datos de los incidentes y los distritos
    :return: Objeto de tipo DataBase con una conexión a la base de datos local
    """
    client = MongoClient()
    db = client[db_name]
    return db
```

4.1.4.2. Creación de los índices

```
def create_indexes(db):
    """
    GEOSPHERE: para procesar coordenadas esféricas
    :param db:  enlace a la base de datos, el código añade un índice sobre los campos que contienen
información geoespacial, y otros que se van a usar en consultas, de esta forma se agiliza el proceso.
    """
    db.incidents.create_index([("Location", GEOSPHERE)])
    db.neighbours.create_index([("the_geom", GEOSPHERE)])
```

```
db.incidents.create_index([("Date", pymongo.ASCENDING)])
db.incidents.create_index([("Category", pymongo.ASCENDING)]
```

4.1.4.3. Consultas Geoespaciales

La consulta first_query realiza una query geoespacial con la que se obtienen todas las incidencias que están a una distancia de 1000 metros de un punto definido con coordinadas.

```
def first_query(db):
    """
    :param db: referencia a la sesión de la bd
    :return: Esta función obtiene los incidentes que están a una distancia máximo de 1000 metros
    desde un punto representado por coordinadas geográficas en formato geojson
    """
    # Montamos la query
    query_incidents = {
        "Location" :{
            "$near": {
                "$geometry" : SON([
                    ("type", "Point"),
                    ("coordinates", [-122.42158168136999, 37.7617007179518])
                ]),
                "$maxDistance": 1000
            }
        }
    }
    # Ejecutamos la querry sobre la colección de incidencias
    query_results = db.incidents.find(query_incidents)
    df = pd.DataFrame(list(query_results))
    return df
```

La consulta second_query obtiene el distrito del dataset de distritos que contiene las coordenadas pasadas por parámetro a MongoDB.

```
def second_query(db):
    """
    :param db: referencia a la sesión de la bd
    :return: devuelve el distrito que contiene las coordinadas usadas en el
    operado de intersección
    """
    # Montamos la query
    query_distrito = {"the_geom":
        {"$geoIntersects":
            {"$geometry": SON([
                ("type", "Point"),
                ("coordinates", [-122.42158168136999, 37.7617007179518])])
            }
        }
    }
    # Ejecutamos la querry sobre la colección de incidencias
    query_results = db.neighbours.find_one(query_distrito)
    return query_results
```

Otra consulta interesante que mezcla los dos datasets, esta vez se obtienen todos los incidentes contenidos geográficamente en un distrito.

```
def third_query(db):
    """
    :param db: referencia a la sesión de la bd
    :return: Devuelve el todos los incidentes, en dataframe, de un distrito, usando
    operadores geo-espaciales, la consulta se realiza en fases sobre las dos
    colecciones, primero encontramos el distrito, y luego buscamos todos los
    incidentes que tienen las coordinadas dentro del polígono
    """
    # Empezamos con el distrito
    query_distrito = {"the_geom": {"$geoIntersects": {
        "$geometry": SON([("type", "Point"), ("coordinates", [-122.42158168136999,
37.7617007179518])])}}}
    distrito = db.neighbours.find_one(query_distrito)
    # Ahora encontramos los incidentes
    query_incidents = {"Location": {"$geoWithin": {"$geometry": distrito['the_geom']}}}
    query_results = db.incidents.find(query_incidents)
    df = pd.DataFrame(list(query_results))
    return df
```

4.1.4.4. Consultas con filtros

Las siguientes consultas obtienen los incidentes usando operadores de filtro implementados en pymongo y que se convierten a operadores de condición sobre consultas en MongoDB. En este caso se realizan consultas para obtener incidentes según fechas de ocurrencia.

```
def since_february(db):
    """
    :param db: referencia a la sesión de la bd
    :return: Devuelve los incidentes que han ocurrido desde febrero de 2018
    """
    fecha = datetime(2018, 2, 1)
    incidents = db.incidents.find({"Date": {"$gt": fecha}})
    df = pd.DataFrame(list(incidents))
    return df


def date_querry(op, sdate, edate):
    """
    :param opr: operador B: between dates, L: less than, G: greater than
    :return: devuelve el filtro de la consulta según lo que se recibe por parámetro
    en op
    """
    switcher = {
        'B': {"Date": {"$gte": sdate, "$lte": edate}},  # Between
        'GE': {"Date": {"$gte": sdate}},            # Greater than or equal
        'LE': {"Date": {"$lte": sdate}},           # less than or equal
    }
    return switcher.get(op)
```

```
def generic_date_search(db, op, sdate, *args, **kwargs):
    """
    :param db: referencia a la sesión de bd
    :param op: operador para seleccionar
    :param sdate: primera fecha, mandaterio
    :param args:
    :param kwargs: Contiene argumento opcional de la seguna fecha
    :return: incidentes que cumplen con el filtro sobre fechas
    """
    edate = kwargs.get('edate', None)
    if not edate is None:
        query = date_querry(op, sdate, edate)
    else:
        query = date_querry(op, sdate, None)
    query_results = db.incidents.find(query)
    df = pd.DataFrame(list(query_results))
    return df
```

4.1.4.5. Consultas con agregación

En la consulta siguiente se ha usado el framework de agregación de pymongo para obtener el total de incidentes por cada categoría.

```
def total_per_category(db):
    """
    Esta función usa el framework de aggregate para obtener el total de cada categoría
    :param db: referencia a la sesión de bd
    :return: Dataframe composed of two columns, categories and count of each category
    """
    #Agrupamos por categoría y los contamos, y luego ordenamos
    pipeline = [
        {"$group": {"_id": "$Category", "count": {"$sum": 1}}},
        {"$sort": SON([("count", -1), ("_id", -1)])}
    ]
    aggregate_results = db.incidents.aggregate(pipeline)
    return pd.DataFrame(list(aggregate_results))
```

La siguiente consulta es del mismo estilo que la anterior, pero usa un pipeline más avanzado que primero proyecto del campo de fecha, la hora en que ha ocurrido el incidente, después se realiza una agrupación de los incidentes que han ocurrido en la misma hora y se cuentan.

```
def total_per_hour(db):
    """
    Esta función usa el framework de aggregate para obtener el total de incidentes de cada hora
    :param db: referencia a la sesión de bd
    :return: Dataframe composed of two columns, hours of the day and count of total incidents in that
hour
    """
    # Primero proyectamos cada documento en otro sacando solo el _id y la hora
    pipeline = [
        {"$project": { "h": { "$hour": "$Date" } }}, # nuevo campo h que ha sido proyecto desde la fecha
```

```
        {"$group": {"_id": "$h", "count": {"$sum": 1}}},
        {"$sort": SON([("_id", 1), ("count", 1)])}
    ]
    aggregate_results = db.incidents.aggregate(pipeline)
    df = pd.DataFrame(list(aggregate_results))
    return df
```

4.1.5. Algunas consultas con Mongo nativo

Antes de hacer la consulta, elegimos la base de datos que vamos a usar en el Shell de mongo:

```
> use san_francisco_incidents
```

4.1.5.1. Creación de índices

Antes de realizar las consultas, vamos a crear algunos índices que nos ayudarán a realizar consultas más optimas. Crearemos un índice sobre los campos Category, DayOfWeek y PdDistrict. El valor 1 en la función createIndex denota que queremos que sea incremental el orden. Crearemos además un índice espacial sobre la clave Location para poder realizar consultas geoespaciales, y otro sobre el campo the_geom de la colección de distritos. Los índices geo-espaciales se definen pasando por parámetro el valor "2dsphere" para la clave que representa el campo sobre el que se va a crear el índice.

```
db.incidents.createIndex( { category: 1 } )
db.incidents.createIndex( { DayOfWeek: 1 } )
db.incidents.createIndex( { PdDistrict: 1 } )
db.incidents.createIndex( { location : "2dsphere" } )
db.neighbours.createIndex( { the_geom : "2dsphere" } )
```

4.1.5.2. Obtener todas las incidencias

```
db.incidents.find( {} )
```

4.1.5.3. Obtener el total de incidencias

```
db.incidents.count()
```

4.1.5.4. Todas las incidencias del distrito MISSION y el total

```
db.incidents.find({PdDistrict: "MISSION"})
db.incidents.count({PdDistrict: "MISSION"})
```

4.1.5.5. Incidencias con resolución “ARREST, BOOKED” y que han ocurrido en marzo de 2015

```
db.incidents.find({
    Resolution: 'ARREST, BOOKED',
    Date: {
        $gt: ISODate('2015-03-01T00:00:00.000Z'),
        $lt: ISODate('2015-03-31T00:00:00.000Z')
        }
    }
)
```

4.1.5.6. Total de incidencias usando el framework de MapReduce

```
db.incidents.mapReduce(
    function() { emit(this._id, 1); },
    function(key, values) { var total = 0;
                  for (var i = 0; i < values.length; i++) {
                    total += values[i];
                  }
                  return total; },
    { out: "MapReduceQuery" }
)
```

4.1.5.7. Total incidencias usando el framework aggregate

```
db.incidents.aggregate(
    [{
        $group:{
            _id : "$Category",
            total:{$sum: 1}
        }
    }]
)
```

4.1.5.8. Total de incidencias por cada día de semana usando el framewrok de aggregate

```
db.incidents.aggregate(
    [{
        $group:{
            _id : "$DayOfWeek",
            total:{$sum: 1}
        }
    }]
)
```

4.1.5.9. Incidencias en un perímetro entre 500 y 1000 metros: Geoespacial

```
db.incidents.find({
    Location: {
        $near:{
            $geometry:  {  type:  "point",    coordinates:  [  -122.4112952,
37.7813411 ]},
            $minDistance: 500,
            $maxDistance: 1000
        }
    }
})
```

4.1.5.10. Obtener el distrito de una incidencia: Geoespacial

La siguiente consulta responde al siguiente requisito: dadas las coordinadas de un incidente, por ejemplo (-122.4112952, 37.7813411), obtener el poligono de la colección de "neighbours" que forma al distrito en que ha ocurrido el incidente. Para esta consulta, se usa el operador $geoIntersects que recibe como parámetro un punto en formato geo-json y realiza una búsqueda geo-espacial sobre el campo "the_geom" de la colección "neighbours".

```
db.neighbours.findOne({
    the_geom: {
        $geoIntersects: {
            $geometry: {
                "type": "Point",
                "coordinates": [-122.42158168136999, 37.7617007179518]
            }
        }
    }
})
```

4.1.5.11. Incidencias de un distrito: Geoespacial

La siguiente consulta, obtiene primero un distrito representado por un multipoligono usando las coordinadas de un punto. Luego usando ese multipoligono realiza una búsqueda geo-espacial en la que usa el operador $geoWithin, que como su nombre lo sugiere, devuelve los documentos de incidencias que sus coordinadas estén situadas dentro de ese polígono.

```
var neighborhood = db.neighbours.findOne({
    the_geom: {
        $geoIntersects: {
            $geometry: {
                type: "Point",
                coordinates: [-122.42158168136999, 37.7617007179518]
            }
        }
    }
})
```

```
db.incidents.find({
    Location: {
        $geoWithin: {
            $geometry: neighborhood.the_geom
        }
    }
})
```

4.2. Cassandra

4.2.1. Conceptos generales

Cassandra es una base de datos NoSQL de tipo clave-valor. Su peculiaridad principal es que se creará una tabla diferente por cada consulta que se quiera realizar.
Sus principales características son:

- Es un sistema descentralizado: todos los nodos tienen el mismo rol, no existe un nodo maestro.
- Replicación: en el clúster y en múltiples datacenters.
- Escalabilidad lineal: las lecturas y escrituras aumentan a medida que se agregan más nodos al clúster.
- Tolerancia a fallas: la información puede estar en más de un nodo (redundancia) y en múltiples centros de datos.
- Consistencia: posibilidad de obtener el registro con la fecha de grabación de un nodo, del clúster o de todos los datacenter.

4.2.2. Modelo de datos

Keyspace
Un keyspace puede ser visto como el contenedor más exterior para datos en Cassandra. Todos los datos en Cassandra deberían vivir dentro de un keyspace. Esto puede ser como una base de datos en el Sistema de gestión de bases de datos relacionales, que es una colección de tablas. En el caso de Cassandra, un keyspace es una colección de familias de columnas.

Familia de Columnas
Una familia de columnas puede ser vista como una colección de filas, y cada fila es una colección de columnas. Es análoga a una tabla en RDBMS pero tiene algunas diferencias. Las familias de columnas son definidas, pero no es necesario para cada fila tener todas las columnas, y las columnas pueden ser añadidas o eliminadas de una fila cuando se requiera.

Columna

La columna es la unidad básica de datos en Cassandra. Puede ser descrita como:

- ”Skinny row”: tamaño casi fijo y un número relativamente pequeño de columnas.

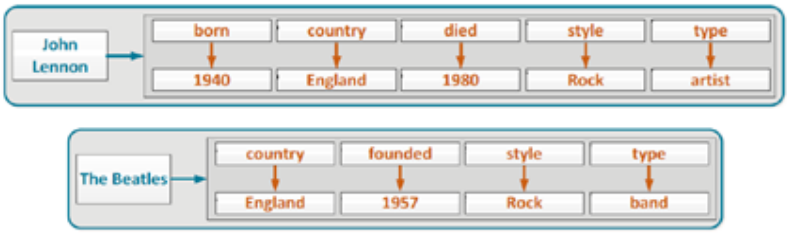


Figura 12. Fila descrita como "Skinny row".

- “Wide row”: tamaño relativamente grande de clave de columnas (100, 1000), este número puede ir aumentando cuando se insertan nuevos datos.

Figura 13. Fila descrita como "Wide row".

La Figura 14 muestra la estructura de Cassandra.

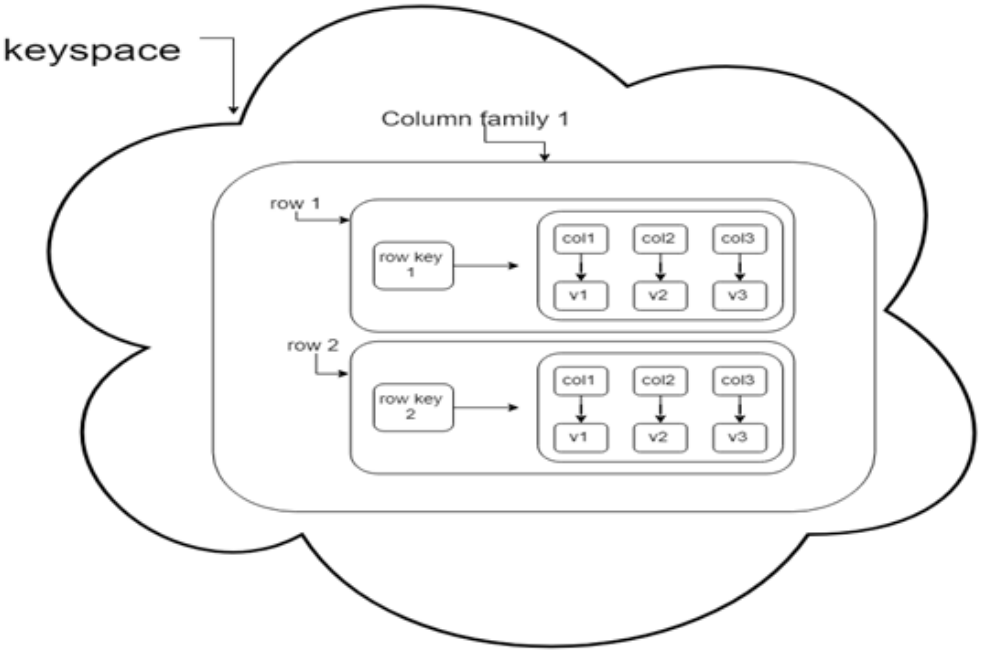


Figura 14. Estructura de Cassandra.

Clave de partición

La clave de partición no es otra cosa que la “primera parte” de la clave primaria e identifica de forma unívoca a una partición. Por otro lado, la clave primaria puede ser simple (una única columna) o estar compuesta por más de una columna (clave compuesta). En el caso de una tabla en la que la clave primaria es simple, la clave de partición es la única columna que existe

en la clave primaria, por lo tanto, coinciden clave primaria y clave de partición. clave de partición es especialmente importante en Cassandra, ya que es la clave responsable de distribuir los datos a lo largo del total de nodos disponibles.

Clave compuesta

Como ya hemos visto, la clave primaria puede tener más de una columna, en estos casos hablamos de clave compuesta. Una característica interesante de la clave compuesta es que únicamente la "primera parte" es considerada como la clave de partición.

Clave clúster

Cuando alguna de las consultas que vamos a realizar implica obtener los datos ordenados ascendentemente o descendentemente por alguna de las columnas entra en juego la clave clúster. La clave clúster la conforman el resto de las columnas que forman parte de la clave primaria pero no de la clave de partición, es decir, la "segunda parte" de la clave primaria.

4.2.3. Driver de Cassandra para Python

Para este proyecto se ha usado cassandra-driver (3.13.0) para trabajar con la base de datos Cassandra desde Python. Este driver funciona exclusivamente con Cassandra Query Language v3 (CQL3) y el protocolo nativo de Cassandra. Es compatible con las versiones posteriores a 2.1 de Cassandra. Igualmente, este driver soporta las versiones 2.6, 2.7, 3.3, 3.4, 3.5, and 3.6 de Python.

La siguiente tabla muestra la correspondencia entre tipos y objetos de Python y con los de CQL.

Python Type	None	Bool	Float	Int, long	Decimal	Str, unicode	Buffer, Bytearray
CQL Literal Type	Null	boolean	Float double	Int, Bigint, Varint, Smallint, Tinyint, counter	Decimal	Ascii, Varchar, text	blob
Python Type	Date	datetime	Time	List, Tuple, Generator	Set, Frozenset	Dict, orderedDict	Uuid.UUID
CQL Literal Type	Date	timestamp	Time	List	set	Map	Timeuuid Uuid

Tabla 3. Correspondencia de tipos Python y SQL.

La instalación del driver cassandra-driver se ha hecho mediando pip (herramienta para instalar paquetes Python). Pip se encargará de instalar todas las dependencias de Python para el driver al mismo tiempo que instala el propio driver. Para instalarlo se usa la siguiente línea de comando: **`pip install cassandra-driver`**

4.2.4. Preparación del fichero de datos

Para poder utilizar el fichero de datos proporcionado ha sido necesario modificar el formato de los campos “Date” y “Time”. Se ha utilizado el siguiente script escrito en Python:

```python
#!/usr/bin/env python
import os

import pandas as pd

def parse_date(date):
    date_as_string = str(date)
    year_as_string = date_as_string[6:10]
    day_as_string = date_as_string[3:5]
    month_as_string = date_as_string[0:2]
    return str(year_as_string+"-"+month_as_string+"-"+day_as_string)

def parse_time(time):
    time_as_string = str(time)
    return str(time_as_string+":00")

def create_csv(nombrecsv,cols):
    path_name_csv=path %(nombrecsv)
    df.to_csv(path_name_csv,index=False, columns=cols)


path ="C:/Users/Ivan/Desktop/doccassandra/CSVs/%s.csv"

df = pd.read_csv('c:/incidenciasPdId.csv')

df["Date"] = df["Date"].apply(parse_date)
df["Time"] = df["Time"].apply(parse_time)

#Todas las columnas
cols = ["IncidntNum","Category","Descript","DayOfWeek","Date","Time","PdDistrict","Resolution","Address","X","Y","Location","PdId"]
create_csv("completo",cols)
```

Figura 15. Script Python para modificar el formato de los campos 'Date' y 'Time'.

El resultado obtenido es un nuevo csv con las mismas columnas, pero con el formato correcto.
Como se ha comentado anteriormente, para cada consulta es necesario la creación de una tabla con los campos que sean requeridos para dicha consulta. Por este motivo, se ha creado un script Python para facilitar la creación de los distintos csv de donde se importarán los datos para cada tabla.

```python
#!/usr/bin/env python
import os

import pandas as pd

def create_csv(nombrecsv,cols):
    path_name_csv=path %(nombrecsv)
    df.to_csv(path_name_csv,index=False, columns=cols)


path ="C:/Users/Ivan/Desktop/doccassandra/CSVs/%s.csv"

df = pd.read_csv('C:/Users/Ivan/Desktop/doccassandra/CSVs/completo.csv')


#Distintos CSVs

#ver incidencias por zonas de la ciudad
cols = ["Category","Descript","Date","Time","PdDistrict","Resolution","Location","PdId"]
create_csv("incidenciasbyzona",cols)


#ver las categorias de incidentes distintas que hay
cols = ["Category"]
create_csv("categorias",cols)


#ver incidentes por categoria y distrito
cols = ["Category","Descript","Date","Time","PdDistrict","Resolution","Location","PdId"]
create_csv("incidenciasbycategoriazona",cols)
```

Figura 16. Script Python para la creación de los distintos csv para importar datos en cada tabla.

Para importar estos csv a una tabla en Cassandra [1] se utiliza el siguiente comando:

COPY keyspace.name_table(col1,col2…colN) FROM '../archivo.csv' WITH HEADER = TRUE;

Una vez que tenemos todos los csv creados en un formato correcto, pasamos a la creación de las tablas en la base de datos Cassandra. La siguiente imagen muestra el código necesario para la creación de algunas de las tablas de nuestro proyecto:

```
#tabla con todos los campos completo
CREATE TABLE incidencias.completoPdId(
   IncidntNum text,
   Category text,
   Descript text,
   DayOfWeek text,
   Date date,
   Time time,
   PdDistrict text,
   Resolution text,
   Address Text,
   X text,
   Y text,
   Location text,
   PdId text,
   PRIMARY KEY(PdId)
);

#ver actividad criminal por zonas de la ciudad
CREATE TABLE incidencias.incidenciasbyzona(
   Category text,
   Descript text,
   Date date,
   Time time,
   PdDistrict text,
   Resolution text,
   Location text,
   PdId text,
   PRIMARY KEY(PdDistrict,PdId)
)WITH CLUSTERING ORDER BY (PdId ASC);


#ver las categorias de incidentes distintas que hay
CREATE TABLE incidencias.categorias(
   Category text,
   PRIMARY KEY(Category)
);
```

Figura 17. Creación de tablas para Cassandra.

4.2.5. Algunas Consultas

En esta sección se mostrará algunas de las consultas que se han realizados con la base de datos Cassandra.

- Listar todas las categorías

SELECT category FROM categorías

- Cantidad distritos

SELECT count(*) FROM distritos

- Listar incidencias según el distrito y la categoría.

SELECT * FROM incidencias.incidenciasbycategoriazona WHERE Category=“SUICIDE” AND pddistrict=“BAYVIEW”

- Listar incidencias según el distrito, la categoria, fecha inicio y fin.

SELECT * FROM incidencias.incidenciasbycategoriazonafecha WHERE Category= “SUICIDE” AND pddistrict= “BAYVIEW” AND date >= “2017-06-01” AND date <= t “2018-08-01”

- Número de incidencias según el distrito, la categoria, fecha inicio y fin.

SELECT COUNT(*) FROM incidencias.incidenciasbycategoriazonafecha WHERE Category= “SUICIDE” AND pddistrict= “BAYVIEW” AND date >= “2017-06-01”AND date <= “2018-08-01”

Para ejecutar consultas desde Python usando el driver de Cassandra es necesario crear una sesión. Para ello, se escribe el siguiente código:

```
cluster = Cluster(['127.0.0.1'])
session = cluster.connect()
session.set_keyspace('incidencias')
```

Figura 18. Creación de sesión para Cassandra.

Se crean se crean distintas funciones en Python para facilitar las llamadas a la ejecución de las consultas. Para ejecutar una consulta de Cassandra desde Python se usa el objeto “session” y el método “execute()”. Estas funciones tendrán la siguiente estructura:

```
#cantidad distritos
def get_num_districts():
    num_districts = session.execute('SELECT count(*) FROM distritos')
    return num_districts[0].count

"""
num_districts = get_num_districts()
print(num_districts)
"""


#Listar incidencias según el distrito'
def get_incidents_by_district(district):
    rows_incidents = session.execute('SELECT * FROM incidencias.incidenciasbyzona WHERE pddistrict=%s',[district])
    return rows_incidents

"""
rows_incidents = get_incidentsbydistrict('NORTHERN')
for incident in rows_incidents:
    print (incident.pdid)
"""

#Número incidencias según el distrito'
def get_num_incidents_by_district(district):
    num_incidents = session.execute('SELECT COUNT(*) FROM incidencias.incidenciasbyzona WHERE pddistrict=%s',[district])
    return num_incidents[0].count
```

Figura 19. Algunas funciones para Cassandra en Python.

4.3. Neo4j

4.3.1. Conceptos generales

Neo4j es un software libre de base de datos orientada a grafos (el más popular de este tipo de bases de datos según el DB-engines ranking [1]), implementado en Java. En las bases de datos orientadas a grafos, la información se representa como nodos de un grafo y sus relaciones como aristas de dicho grafo. De esta forma se puede usar la teoría de grafos para recorrer la base de datos.

En cuanto a la estructura de los datos en Neo4j, toda la información se almacena en forma de nodo, arista o atributo. Cada nodo o arista puede tener varios atributos. Tanto los nodos como las aristas pueden etiquetarse, es decir, pueden tener un tipo o clase por así decirlo. Estas etiquetas puedes usarse para facilitar las búsquedas dentro de la base de datos orientada a

grafos. Además, a partir de la versión 2.0 de Neo4j se permite la creación de índices para acelerar las consultas.

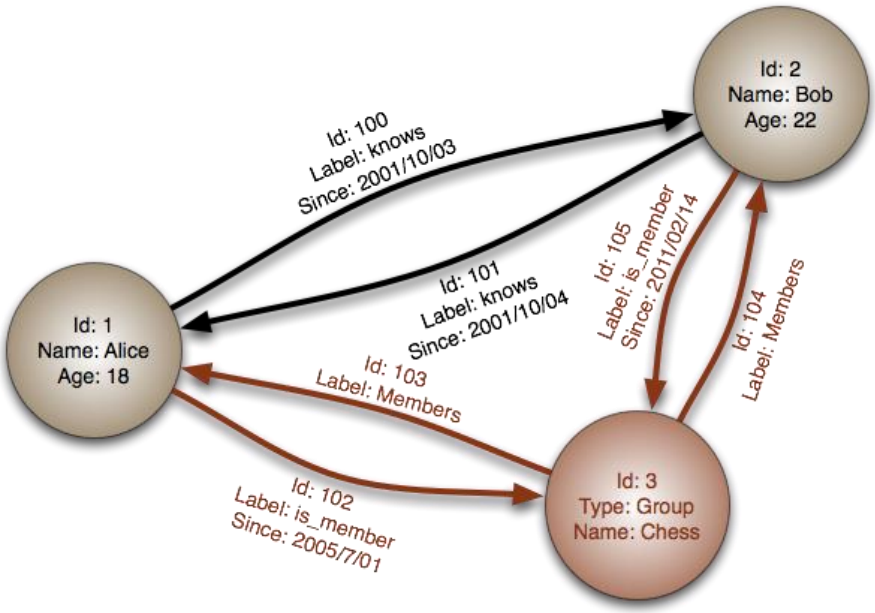


Figura 20. Ejemplo de grafo para las DB orientadas a grafos.

Para realizar dichas consultas y recuperar información de nuestra base de datos Neo4j se utiliza Cypher. Cypher es un lenguaje orientado a consultas que permite realizar consultas, inserciones, actualizaciones y borrados en la base de datos orientado a grafos de una manera eficiente. Consultas muy complicadas pueden ser expresadas de una manera muy sencilla usando este lenguaje. Esto permite al usuario concentrarse en su dominio en lugar de perderse intentando acceder a la base de datos.

Cypher fue creado originalmente por Neo4j, Inc. para su base datos precisamente, pero desde octubre de 2015 abrió su licencia a través del proyecto openCypher.

4.3.2. Driver de Neo4j para Python

No hay una única alternativa para acceder a nuestra base de datos Neo4j con Python. Existe un amplio catálogo de drivers para conectar Python a Neo4j, ya que a parte del driver oficial al que Neo4j da soporte, existen otras alternativas desarrolladas por los miembros de la comunidad de Neo4j.

Antes de seleccionar uno de estos drivers, primero los estudiamos brevemente [2], aunque al final nos decantamos por el oficial. Pese a que algunos de estos drivers facilitan las consultas, evitando usar Cypher, el hecho de que exista un driver con soporte oficial de Neo4j, Inc, y que nosotros ya estuviéramos familiarizados con el lenguaje Cypher de la primera entrega, hizo decantarnos por el oficial. Además, es fácil de utilizar si se conoce previamente la sintaxis básica de Cypher.

Para usarlo primero hay que instalar con pip el módulo **neo4j-driver** de Python. Una vez instalado, debemos seguir los siguientes pasos en nuestro fichero .py:

- **Importar la clase GraphDatabase**: `from neo4j.v1 import GraphDatabase`.
- **Crear una instancia del driver**:
  `driver = GraphDatabase.driver(uri_db, auth("user", "pass"))`, donde uri_db es la uri de la base de datos, y user/pass el usuario y contraseña para acceder a ella.
- **Crear la sesión**: `session = driver.session()`.
- **Crear la transacción**: `tx = session.begin_transaction()`.

Una vez tenemos la transacción, podemos realizar consultas en nuestra base de datos Neo4j. Cuando terminemos de usarla, debemos cerrar la sesión con el la función `session.close()`.

4.3.3. Preprocesado y carga de datos

Tras estudiar el dataset proporcionado se optó por implementar un grafo con tres tipos de nodos (***Incident***, ***Category*** y ***District***) y dos tipos de relaciones entre estos nodos: relación ***WHERE*** de ***Incident*** a ***District*** y relación ***WHAT*** de ***Incident*** a ***Category***.

Entrando en detalle en cada tipo de nodo, ***Incident*** incluye los campos *incnum, dayofweek, resoultion* y *timestamp*, que equivalen al número de incidente (id único), el día de la semana en el que tuvo lugar el incidente, la resolución de este y el instante de tiempo en el que ocurrió el crimen, respectivamente. Por su parte, el nodo ***District*** solo incluye el campo *district*, que es el nombre del distrito en cuestión. Por último, ***Category*** cuenta solo con el campo *category*, el tipo de categoría en la que se incluyen los incidentes.

En la Figura 21, podemos observar un ejemplo de un incidente (nodo ***Incident***) relacionándose con el distrito (relación ***WHERE*** hacia un nodo ***District***) y con la categoría (relación ***WHAT*** hacia un nodo ***Category***).

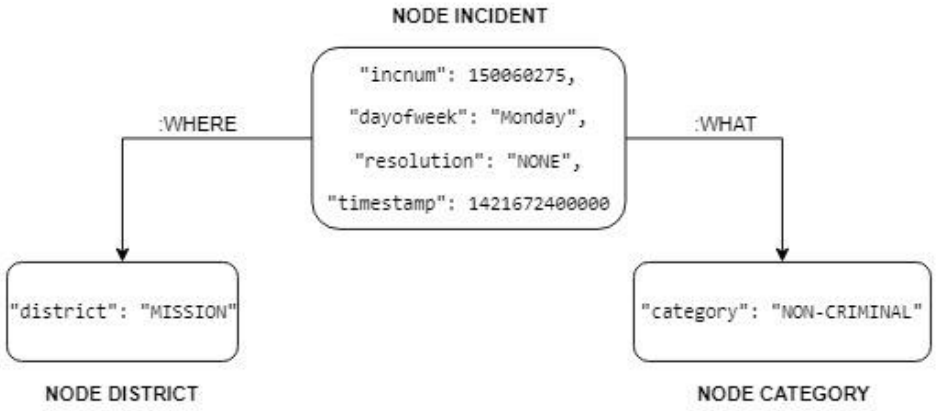


Figura 21. Ejemplo simple del grafo implementado.

El problema que nos encontramos de cara a volcar el contenido del dataset en la base de datos de Neo4j es que algunos números de incidente se repiten, por lo que no se podía utilizar este número como identificador único de los nodos ***Incident***.

Por ello, se llevó a cabo un pre-procesado del fichero de datos proporcionado. Este pre-procesado consistió en obtener otros cinco datasets para crear nuestro grafo. Los ficheros de datos generados fueron los siguientes:

- **Districts.csv**: los distritos de San Francisco que aparecen en el dataset original. Son un total de 10.
- **Categories.csv**: las diferentes categorías de crímenes. Un total de 39.
- **Incidents.csv**: todos los incidentes **ÚNICOS** que han tenido lugar en San Francisco. Más de 1.700.000 incidentes.
- **IncToDistrict.csv**: relaciona los incidentes ÚNICOS con el distrito en el que tuvieron lugar.
- **IncToCategory.csv**: relaciona los incidentes con la categoría en la que se clasifican. En este caso sí que se pueden repetir los números de incidente, ya que un mismo incidente se puede clasificar en varias categorías.

La generación de estos nuevos dataset se realizó en el proyecto Maven ProcessForNeo, en el que se usa la dependencia OpenCSV, que permite leer y parsear un CSV. Gracias a la mencionada dependencia se obtuvieron los campos necesarios para la construcción de cada uno de los nuevos ficheros de datos ya explicados. El proyecto ProcessForNeo también se adjunta para su posible revisión y/o estudio.

Antes de volcar el contenido de estos ficheros en nuestra base de datos orientada a grafos debemos cambiar una serie de parámetros del fichero de configuración de la BD. Dicho archivo se denomina **neo4j.conf** y lo podemos encontrar utilizando el programa Neo4j Desktop. Para ello nos dirigimos al proyecto en el que tengamos nuestra base de datos y clicamos en la opción 'Manage'.

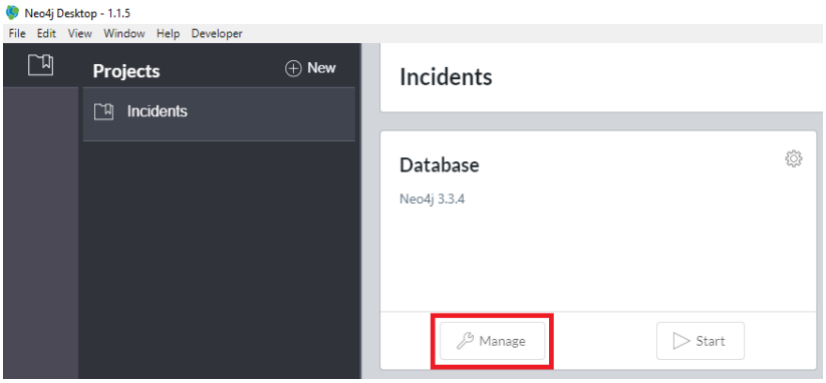


Figura 22. Administración de los grafos con Neo4j Desktop

Una vez pulsamos dicho botón, se nos abre la siguiente ventana. En la pestaña 'Settings' tenemos ante nosotros el fichero de configuración de nuestra base de datos.

Una vez abramos el fichero de configuración debemos comentar cuatro líneas:

- **`dbms.directories.import=import`**: para poder importar CSVs desde cualquier directorio de nuestro equipo.
- **`dbms.memory.heap.initial_size`**, **`dbms.memory.heap.max_size`** y **`dbms.memory.pagecache.size`**: para que la importación de datos, consultas y borrados vayan más rápidos.

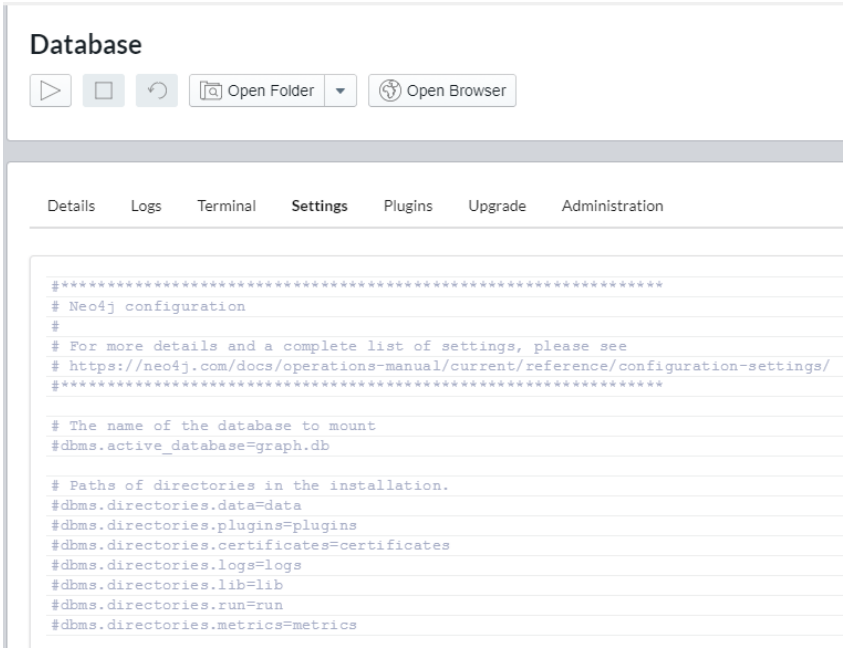


Figura 23. Fichero de configuración neo4j.conf.

Ahora ya podemos crear los nodos y las relaciones de nuestro grafo. Para ello utilizaremos el browser de la aplicación de escritorio de Neo4j y ejecutamos el script **importNeo.txt**.

En primer lugar, creamos índices para los campos category (del nodo Category), district (del nodo District) y incnum (del nodo Incident), lo cual nos permitirá realizar búsquedas con mayor facilidad por el grafo.

```
1 CREATE INDEX ON :Category(category);
2
3 CREATE INDEX ON :District(district);
4
5 CREATE INDEX ON :Incident(incnum);
```

Figura 24. Creación de los índices.

A continuación, procedemos a crear los nodos Category, District e Incident. En la creación de los nodos Incident además usaremos la cláusula **`USING PERIODIC COMMIT 5000`**, para que se haga un commit cada 5000 nodos creados.

```
LOAD CSV WITH HEADERS FROM "file:///C:\\PathTo\\Categories.csv" AS row
CREATE (cat:Category {category: row.Category});

LOAD CSV WITH HEADERS FROM "file:///C:\\PathTo\\Districts.csv" AS row
CREATE (dis:District {district: row.District});
```

Figura 25. Creación de los nodos Category y District.

```
USING PERIODIC COMMIT 5000
LOAD CSV WITH HEADERS FROM
"file:///C:\\Users\\Programar\\Desktop\\Incidents.csv" AS row
CREATE (inc:Incident {incnum: toInt(row.IncNum), dayofweek: row.DayOfWeek,
timestamp: toInt(row.Timestamp), resolution: row.Resolution});
```

Figura 26. Creación de los nodos Incident.

Por último, crearemos las relaciones de nuestro grafo. Al igual que en la creación de los nodos de incidentes, haremos commit cada 5000 relaciones creadas.

```
USING PERIODIC COMMIT 5000
LOAD CSV WITH HEADERS FROM
"file:///C:\\Users\\Programar\\Desktop\\IncToDistrict.csv" AS row
MATCH (inc:Incident { incnum: toInt(row.IncNum)}), (dis:District { district:
row.District})
CREATE (inc)-[:WHERE]->(dis);
```

Figura 27. Creación de las relaciones WHERE de Incident a District.

```
USING PERIODIC COMMIT 5000
LOAD CSV WITH HEADERS FROM
"file:///C:\\Users\\Programar\\Desktop\\IncToCategory.csv" AS row
MATCH (inc:Incident { incnum: toInt(row.IncNum)}), (cat:Category { category:
row.Category})
CREATE (inc)-[:WHAT]->(cat);
```

Figura 28. Creación de las relaciones WHAT de Incident a Category.

4.3.4. Algunas consultas

A continuación, vamos a exponer algunas de las consultas que podemos realizar en el grafo diseñado:

- Obtener todos los distritos
  `MATCH (dis:District) RETURN dis`
- Obtener el distrito 'Tenderloin'
  `MATCH (dis:District {district:'TENDERLOIN'}) RETURN dis`

- Obtener todas las categorías

```
MATCH (cat:Category) RETURN cat
```

- Obtener 100 incidentes de vandalismo

```
MATCH (inc:Incident), (cat:Category {category:'VANDALISM'}) RETURN inc-[:WHAT]->(cat) LIMIT 100
```

- Obtener el número de casos de vandalismo en el distrito 'Tenderloin'

```
MATCH (cat:Category {category:'VANDALISM'}), (dis:District {district:'TENDERLOIN'}) RETURN size((cat)<-[:WHAT]-()-[:WHERE]->(dis)) AS cases
```

- Obtener el porcentaje de casos de vandalismos en el distrito 'Tenderloin'

```
MATCH (cat:Category {category:'VANDALISM'}), (dis:District {district:'TENDERLOIN'}) WITH toFloat(size((cat)<-[:WHAT]-()-[:WHERE]->(dis))) AS cases, size(()-[:WHERE]->(dis)) * 1.0 AS total_cases RETURN cases/total_cases*100 AS percentage
```

5. Herramientas de visualización

5.1. Matplotlib

Matplotlib es una biblioteca de trazado 2D de Python que produce figuras de publicación en una variedad de formatos impresos y entornos interactivos en todas las plataformas. Matplotlib se puede usar en scripts Python, shells Python e IPython, el cuaderno Jupyter, servidores de aplicaciones web y cuatro toolkits gráficos de interfaz de usuario.

En este proyecto hemos usado esta librería para mostrar algunas figuras para obtener una visualización intuitiva sobre los datos.

El siguiente ejemplo muestra gráficamente la consulta de agregación para obtener el total de incidentes por horas.

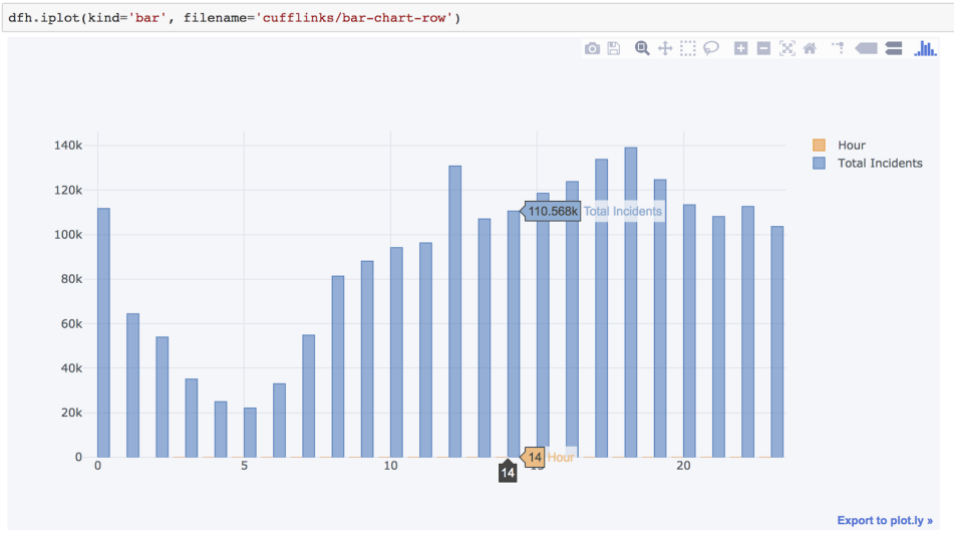


Figura 29. Gráfico de incidentes por hora.

Aquí, por ejemplo, podemos apreciar el número de incidentes totales en cada uno de los distritos de la ciudad de San Francisco.

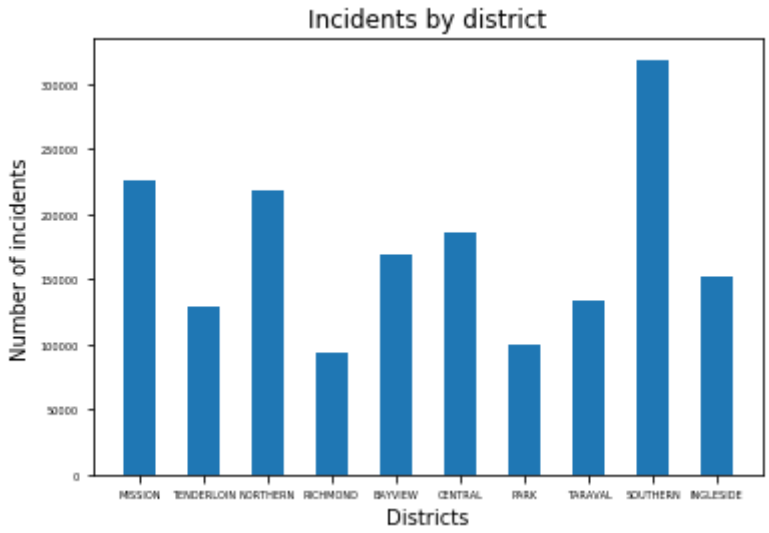


Figura 30. Gráfico de incidentes en cada distrito.

5.2. Folium

Folium facilita la visualización de datos manipulados en Python en un mapa de folleto interactivo. Permite tanto el enlace de datos a un mapa para visualizaciones de coropletas como el paso de visualizaciones ricas de vectores / ráster / HTML como marcadores en el mapa.

La biblioteca tiene una serie de tilesets incorporados de OpenStreetMap, Mapbox y Stamen, y admite tilesets personalizados con Mapbox o las claves API de Cloudmade. folium admite superposiciones de imágenes, videos, GeoJSON y TopoJSON.

En nuestro proyecto hemos usado esta librería para mostrar los resultados de algunas consultas realizadas sobre MongoDB. La siguiente función en Python recibe un dataset con los 1000 primeros incidentes del mes de febrero de 2018.

```
def draw_map(ds):
    """
    Esta función recibe un conjunto de incidentes y los dibuja en un mapa usando el paquete folium,
    el mapa se guarda en un fichero.
    :param ds: dataset de incidentes en formato de DataFrame
    """
    incid_map = folium.Map(location=[37.7617007179518, -122.42158168136999], zoom_start=11,
tiles='Stamen Terrain')
    marker_cluster = plugins.MarkerCluster().add_to(incid_map)
    for name, row in ds.iterrows():
        folium.Marker([row["Y"], row["X"]], popup=row["Descript"]).add_to(marker_cluster)
    incid_map.save('incidents.html')
    return incid_map
```

El resultado es el que se ve en la Figura 31:

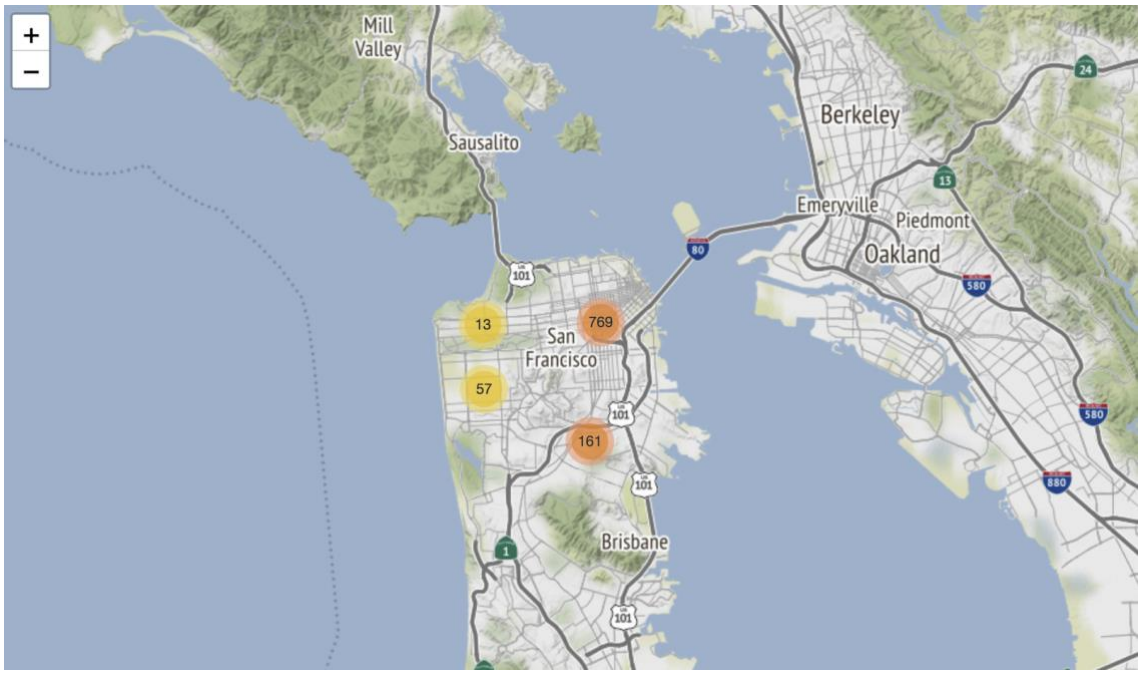


Figura 31. Incidentes en febrero 2018.

Folium también genera mapas de calor. En la siguiente consulta mostramos un mapa de calor con los 1000 primeros incidentes que han ocurrido entre dos fechas.

```
# Fechas para filtro de fechas
fecha1 = datetime(2017, 12, 1)
fecha2 = datetime(2017, 12, 31)
# Buscaremos los incidentes entre dos fechas, B=Between (ver doc de la función)
date_results = generic_date_search(sfdb,'B', fecha1, edate=fecha2)
hm = draw_heatmap(date_results.iloc[:1000])
```

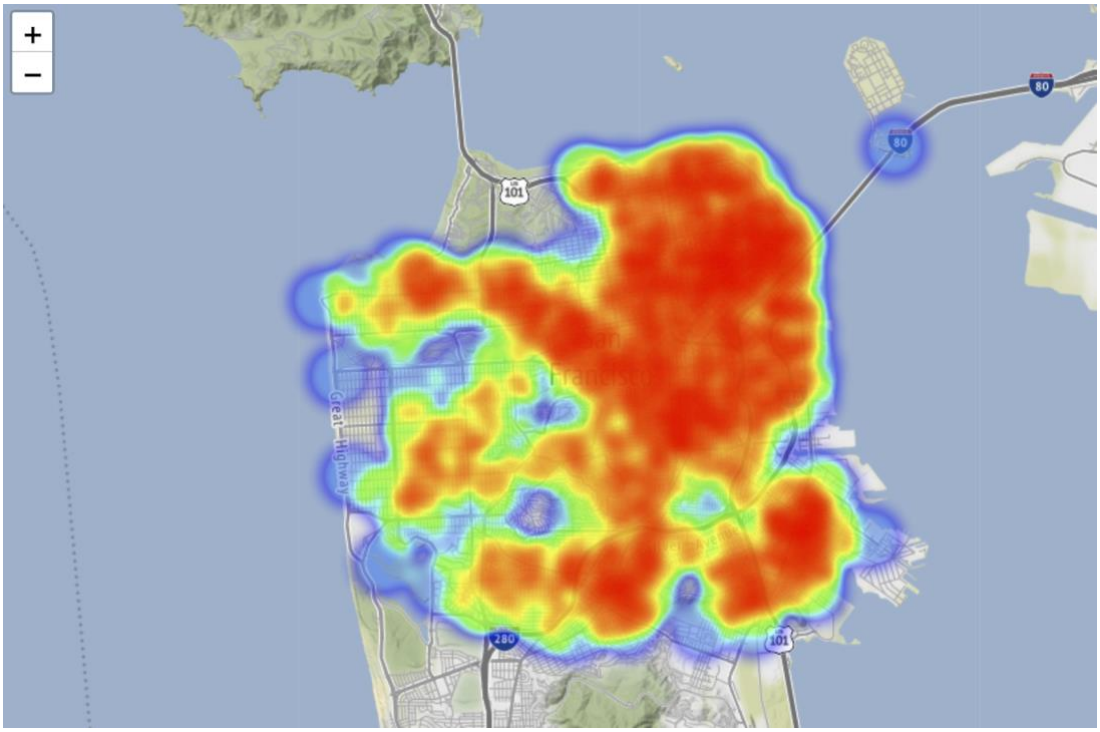


Figura 32. Mapa de calor de incidentes en diciembre de 2017.

5.3. Tableau Desktop con MongoDB

5.3.1. Instalación y conexión

Para usar Tableau junto con MongoDB, se ha usado el conector de BI de MongoDB, es una librería desarrollada por los propios desarrolladores de MongoDB para enlazar la base de datos de documentos con varias soluciones de Business Intelligence, en este caso Tableau.
El siguiente tutorial contiene toda la información necesaria para tener MongoDB y Tableau funcionando en local [7].

5.3.2. Funcionamiento

El conector de mongodb con tableau, realiza dos tareas que hacen que la conexión sea posible. Principalmente recoge consultas desde Tableau en SQL, ya que tableau funciona con esquemas relacionales, y los convierte de forma eficiente a consultas de MongoDB.

También para que Tableau entienda la estructura de los documentos de Mongodb, el conector crea un esquema relacional a base de los documentos. La siguiente figura muestra el proceso.

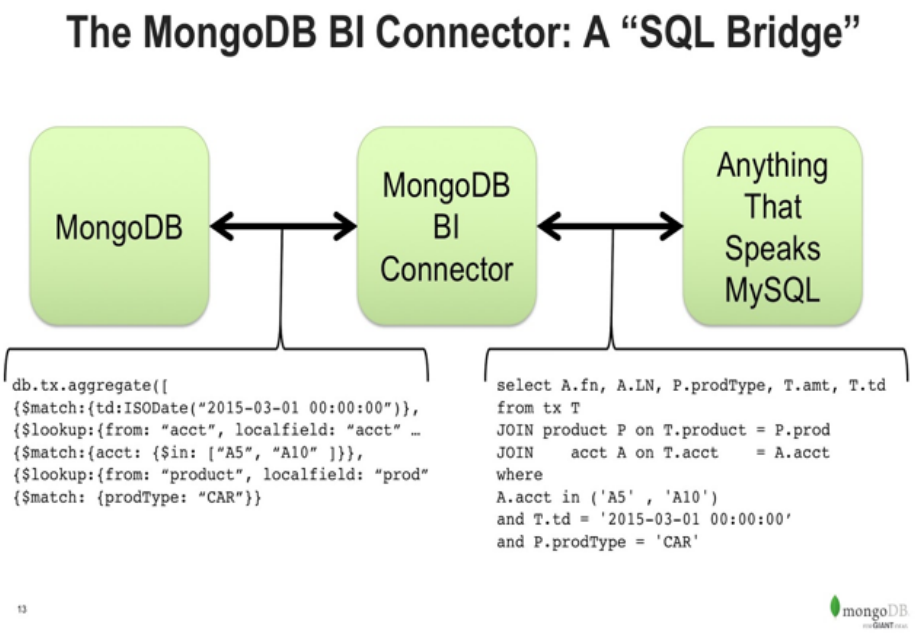


Figura 33. Conexión Tableau Desktop con MongoDB a través del driver Mongo BI.

5.3.3. Ejemplos

A continuación, vemos algunas capturas que muestran esta integración, y el proceso de lanzar todas las herramientas.

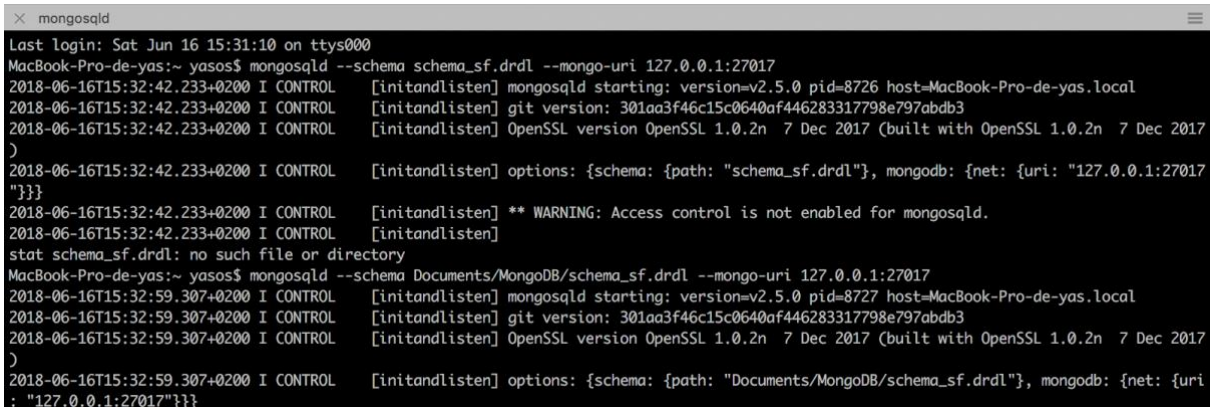


Figura 34. Arranque el driver Mongo BI.

En la captura anterior, tenemos funcionando el proceso mongosqld que es el propio conector BI de MongoDB, usando el fichero schema_sf.drdl que contiene un defición relacional de los documentos de las colecciones de Mongodb.

Teniendo el proceso de mongod corriendo, arrancamos Tableau, y realizamos una consulta en la que mostramos en resultado en un mapa, además se aplican filtros para que solo se obtengan incidentes de traspaso y asalto.

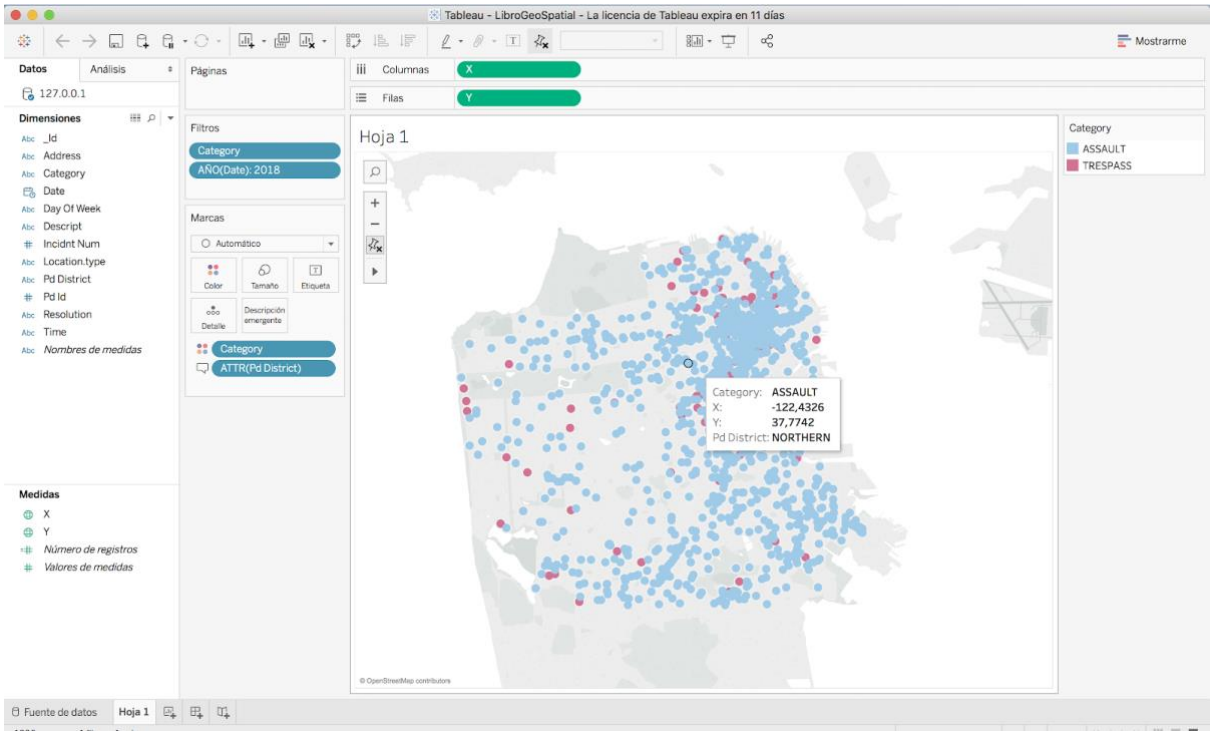


Figura 35. Mapa de incidentes generado con Tableau Desktop.

6. Machine Learning

6.1. Introducción

El aprendizaje automático es una rama de la ciencia que se ocupa de programar los sistemas de tal manera que automáticamente aprenden y mejoran con la experiencia. Aquí, aprender significa reconocer y entender los datos de entrada y tomar decisiones acertadas basadas en los datos suministrados.

En este proyecto hemos puesto en práctica dos técnicas de machine learning, clustering y clasificación.

Antes de realizar ninguna técnica hacemos una consulta para obtener un dataset con 100000 incidentes, en este caso elegimos los últimos desde la base de datos MongoDB, guardamos el resultado de la consulta en un DataFrame.

Como se ve en la captura siguiente, solo seleccionamos los campos que nos interesan, en nuestro caso la categoría, día de semana, distrito y la resolución.

```
# Especificamos los campos que queremos para seleccionar desde MongoDB
target_fields = {
    'Category':1,
    'DayOfWeek':1,
    'PdDistrict':1,
    'Resolution':1
}
# Select lo hacemos usando el primero sort para ordenar según la fecha y luego limitamos sólo a 10000
incidents_ml = sfdb_mon.incidents.find({},target_fields).sort('Date', DESCENDING).limit(100000);
df_ml = pd.DataFrame(list(incidents_ml))
df_ml = df_ml.drop(['_id'], axis=1) # Quitamos el campo _id que siempre viene por defecto
df_ml.__len__() # Vemos el tamaño del df que debe ser 10000
```

```
In [23]: df_ml.head()
```

Out[23]:

	Category	DayOfWeek	PdDistrict	Resolution
0	KIDNAPPING	Saturday	MISSION	ARREST, BOOKED
1	SECONDARY CODES	Saturday	MISSION	ARREST, BOOKED
2	OTHER OFFENSES	Saturday	MISSION	ARREST, BOOKED
3	ASSAULT	Saturday	MISSION	ARREST, BOOKED
4	VANDALISM	Saturday	NORTHERN	NONE

6.2. Clasificación

La clasificación, también conocida como categorización, es una técnica de aprendizaje automático que utiliza datos conocidos para determinar cómo se deben clasificar los nuevos datos en un conjunto de categorías existentes. La clasificación es una forma de aprendizaje supervisado.

En nuestro proyecto, para realizar la clasificación, vamos a limpiar los datos, convirtiendo los campos de tipo string a tipo Categorical de Python, de esta forma obtenemos un tipo numérico que es más adecuado para el tratamiento de datos.
Usaremos un dataframe para la clasificación y otro para el clustering.
Vamos a clasificar según el campo de Categoría, por lo que no lo vamos a convertir.

In [24]:
```
class_df = df_ml.copy() # Dataframe de clasificación
# Convertimos los demás campos a tipo Categoría de pandas
class_df.PdDistrict = pd.Categorical(class_df.PdDistrict)
class_df.DayOfWeek = pd.Categorical(class_df.DayOfWeek)
class_df.Resolution = pd.Categorical(class_df.Resolution)
# En vez de Strings vamos a guardar los códigos en los valores
class_df['PdDistrict'] = class_df.PdDistrict.cat.codes
class_df['DayOfWeek'] = class_df.DayOfWeek.cat.codes
class_df['Resolution'] = class_df.Resolution.cat.codes
class_df.head()
```

Out[24]:

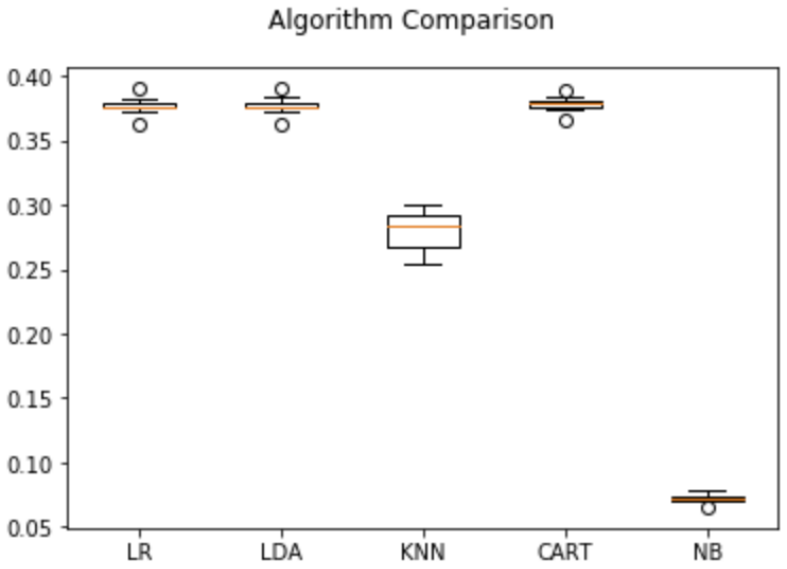
	Category	DayOfWeek	PdDistrict	Resolution
0	KIDNAPPING	2	3	0
1	SECONDARY CODES	2	3	0
2	OTHER OFFENSES	2	3	0
3	ASSAULT	2	3	0
4	VANDALISM	2	4	8

In [25]:
```
# Split-out validation dataset
array = class_df.values
X = array[:,1:4]
Y = array[:,0] # El campo categoría es el primero
validation_size = 0.20 # Usamos 20% para la validación y el resto para el entrenamiento
seed = 7
X_train, X_validation, Y_train, Y_validation =
                    model_selection.train_test_split(X, Y, test_size=validation_size, random_state=seed)
scoring = 'accuracy'

# Spot Check Algorithms
models = []
models.append(('LR', LogisticRegression()))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
# evaluate each model in turn
results = []
names = []
for name, model in models:
    kfold = model_selection.KFold(n_splits=10, random_state=seed)
    cv_results = model_selection.cross_val_score(model, X_train, Y_train, cv=kfold, scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
```

```
LR: 0.376425 (0.006599)
LDA: 0.376650 (0.006621)
KNN: 0.279412 (0.015473)
CART: 0.378088 (0.006084)
NB: 0.072463 (0.003296)
```

In [26]:
```
# Comparamos los algoritmos
fig = plt.figure()
fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
plt.boxplot(results)
ax.set_xticklabels(names)
plt.show()
```



A continuación, realizamos la predicción con el modelo cart.

In [27]:
```
# Viendo el resultado, el CART ha sido el mejor
CART = DecisionTreeClassifier()
CART.fit(X_train, Y_train)
predictions = CART.predict(X_validation)

print(accuracy_score(Y_validation, predictions))
print(confusion_matrix(Y_validation, predictions))
print(classification_report(Y_validation, predictions))
```

```
0.3699
[[ 0  0  0 ...  0  0  0]
 [ 0 14  0 ...  0 10  1]
 [ 0  0  0 ...  0  0  0]
 ...
 [ 0  1  0 ...  5  0  0]
 [ 0 11  0 ...  0 10  0]
 [ 0  3  0 ...  0  0  0]]
```

	precision	recall	f1-score	support
ARSON	0.00	0.00	0.00	41
ASSAULT	0.16	0.01	0.01	1823
BAD CHECKS	0.00	0.00	0.00	2
BRIBERY	0.00	0.00	0.00	5
BURGLARY	0.20	0.00	0.00	808
DISORDERLY CONDUCT	0.00	0.00	0.00	44
DRIVING UNDER THE INFLUENCE	0.00	0.00	0.00	28
DRUG/NARCOTIC	0.19	0.10	0.13	421
DRUNKENNESS	0.00	0.00	0.00	53
EMBEZZLEMENT	0.00	0.00	0.00	18
EXTORTION	0.00	0.00	0.00	6
FAMILY OFFENSES	0.00	0.00	0.00	4
FORGERY/COUNTERFEITING	0.00	0.00	0.00	67
FRAUD	0.00	0.00	0.00	317
GAMBLING	0.00	0.00	0.00	3
KIDNAPPING	0.00	0.00	0.00	36
LARCENY/THEFT	0.39	0.96	0.56	6175
LIQUOR LAWS	0.00	0.00	0.00	8
LOITERING	0.00	0.00	0.00	3
MISSING PERSON	0.42	0.02	0.04	623
NON-CRIMINAL	0.37	0.01	0.02	2291
OTHER OFFENSES	0.31	0.58	0.40	2286
PORNOGRAPHY/OBSCENE MAT	0.00	0.00	0.00	3
PROSTITUTION	0.00	0.00	0.00	62
RECOVERED VEHICLE	0.00	0.00	0.00	78
ROBBERY	0.00	0.00	0.00	454
RUNAWAY	0.00	0.00	0.00	42
SECONDARY CODES	0.05	0.00	0.01	264
SEX OFFENSES, FORCIBLE	0.00	0.00	0.00	128
SEX OFFENSES, NON FORCIBLE	0.00	0.00	0.00	5
STOLEN PROPERTY	0.00	0.00	0.00	99
SUICIDE	0.00	0.00	0.00	17
SUSPICIOUS OCC	0.09	0.00	0.00	722
TREA	0.00	0.00	0.00	1
TRESPASS	0.00	0.00	0.00	186
VANDALISM	0.00	0.00	0.00	1234
VEHICLE THEFT	0.22	0.01	0.01	737
WARRANTS	0.13	0.01	0.03	681
WEAPON LAWS	0.00	0.00	0.00	225
avg / total	0.25	0.37	0.23	20000

Figura 36. Resultado de la clasificación por categorías.

6.3. Clustering

La agrupación en clúster se usa para formar grupos o grupos de datos similares basados en características comunes. La agrupación es una forma de aprendizaje no supervisado.
En nuestro proyecto hemos seguido el guion proporcionado en el Campus Virtual para aplicar clustering al dataset de iris. Principalmente hemos aplicado las mismas técnicas usando Python, pero cambiando el dataset por el de incidentes.
El clustering lo vamos a hacer sobre el campo de Distrito y vamos a intentar crear 10 clusters.

In [33]:
```
# Dataframe del cluster, copiado del dataframe original, y hay que hacer el mismo tratamiento
# y limpiza que se le ha hecho al de clasificación
df_cluster = df_ml.copy()
df_target = df_ml[['PdDistrict']]
df_cluster = df_cluster.drop(['PdDistrict'], axis=1)
df_cluster.head()
```

Out[33]:

	Category	DayOfWeek	Resolution
0	KIDNAPPING	Saturday	ARREST, BOOKED
1	SECONDARY CODES	Saturday	ARREST, BOOKED
2	OTHER OFFENSES	Saturday	ARREST, BOOKED
3	ASSAULT	Saturday	ARREST, BOOKED
4	VANDALISM	Saturday	NONE

In [34]:
```
# Convertimos a tipo Categorical y luego cambiamos los valores a tipo numérico
df_cluster.Category = pd.Categorical(df_cluster.Category)
df_cluster.DayOfWeek = pd.Categorical(df_cluster.DayOfWeek)
df_cluster.Resolution = pd.Categorical(df_cluster.Resolution)
df_target.PdDistrict = pd.Categorical(df_target.PdDistrict)

df_cluster['Category'] = df_cluster.Category.cat.codes
df_cluster['DayOfWeek'] = df_cluster.DayOfWeek.cat.codes
df_cluster['Resolution'] = df_cluster.Resolution.cat.codes
df_target['PdDistrict'] = df_target.PdDistrict.cat.codes
```

In [35]:
```
# Creamos un array de colores, un color por cada distrito
colors = np.array(['red', 'green', 'blue','pink', 'yellow', 'purple', 'black', 'grey', 'brown', 'orange'])
```

In [36]:
```
# Pasamos a ajustar un modelo usando Kmeans con un k=10
model = KMeans(n_clusters=10)
model.fit(df_cluster)
model.labels_
```

Out[36]:
```
array([8, 4, 4, ..., 3, 7, 4], dtype=int32)
```

In [37]:
```
# The fudge to reorder the cluster ids.
predictedY = np.choose(model.labels_, [0,1,2,3,4,5,6,7,8,9]).astype(np.int64)
# Ahora mostramos el resultado después de haber ajustado el modelo a los datos, vamosa  ver cómo quedan
# los datos de categoría con respecto al día de la semana antes y después del ajuste.
plt.subplot(1, 2, 1)
plt.scatter(df_cluster['DayOfWeek'], df_cluster['Category'], c=colors[df_target['PdDistrict']], s=40 )
plt.title('Before classification')

# Plot the classifications according to the model
plt.subplot(1, 2, 2)
plt.scatter(df_cluster['DayOfWeek'], df_cluster['Category'], c=colors[predictedY], s=40)
plt.title("Model's classification")
```

Finalmente, el resultado final de clustering nos da el modelo siguiente, donde podemos apreciar cómo se agrupan los incidentes de color parecido, que se refieren a la misma categoría, en la misma zona del plano que define el día de la semana con la categoría.

```
Out[37]: Text(0.5,1,"Model's classification")
```

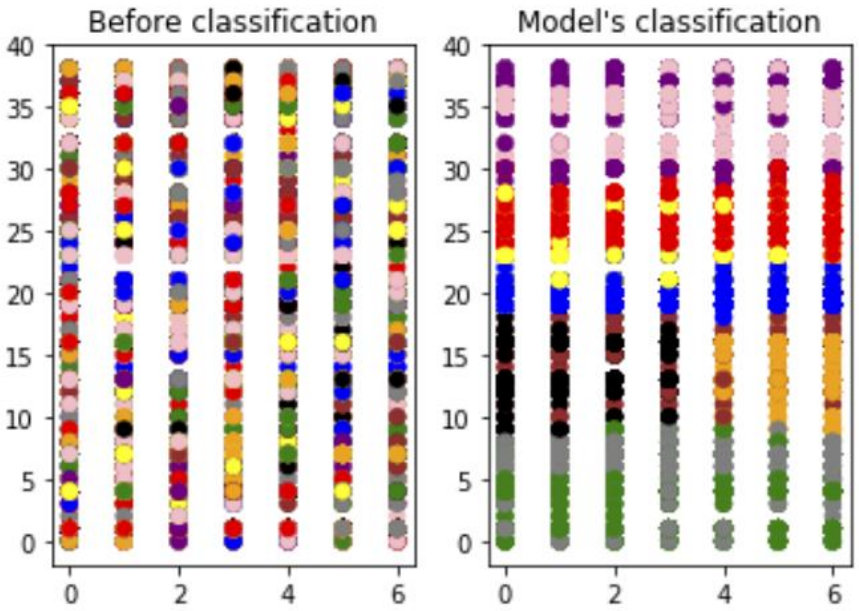


Figura 37. Resultados de la clusterización por categorías.

7. Conclusiones y Trabajos Futuros

En este proyecto nos hemos centrado en conocer tres bases de datos NoSQL, cada una de ella con un paradigma diferente (orientado a documentos, de clave-valor y orientado a grafos). Además, hemos aprendido a acceder a estas tres bases de datos con el lenguaje Python a través de sus diversos drivers. También hemos explorado diversas tecnologías para llevar a cabo la visualización de estos datos almacenados, desde gráficas de barras (Matlibplot) pasando por mapas de calor (Folium) hasta herramientas de BI comerciales (Tableau Desktop).

Por último, hemos intentado adentrarnos en el mundo del Machine Learning, aplicando algoritmos supervisados y no supervisados.

Como trabajos futuros se podrían estudiar otros sistemas de bases de datos NoSQL como Hive y utilizar otras herramientas de visualización de datos que aporten más flexibilidad a nuestros gráficos y mapas.

Referencias y bibliografía

[1] Tutorialspoint (Python), [En línea]. Available: https://www.tutorialspoint.com/python/index.htm.

[2] Tutorialspoint (Pandas), [En línea]. Available: https://www.tutorialspoint.com/python_pandas/index.htm.

[3] Enlace de descarga del dataset de los distritos de San Francisco, [En línea]. Available: https://data.sfgov.org/Geographic-Locations-and-Boundaries/SF-Find-Neighborhoods/pty2-tcw4.

[4] Datastax, [En línea]. Available: https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlshCopy.html.

[5] DB Engines Web Site, [En línea]. Available: https://db-engines.com/en/ranking/graph+dbms.

[6] Neo4j Drivers, [En línea]. Available: https://neo4j.com/developer/python/.

[7] MongoDB (Tableau Desktop connector), [En línea]. Available: https://docs.mongodb.com/bi-connector/master/connect/tableau-no-auth/.

[8] Envatotuts+, [En línea]. Available: https://code.tutsplus.com/es/articles/getting-started-with-cassandra-using-cql-api-and-cqlsh--cms-28026.