

San Francisco crime: deep analysis and prediction model

Álvaro · López · Sánchez

Julio · Sorroche · García

Lupicinio · García · Ortiz

Abstract

En este proyecto utilizaremos la información de la actividades criminales producidas en la ciudad de **San Francisco** organizadas en un volumen masivo de datos, y con el fin de seguir una serie de etapas de pre procesamiento, análisis para finalmente obtener un modelo de predicción. Para ello seguiremos diferentes perspectivas y metodologías a lo largo del almacenamiento, reprocesamiento y representación de la información con ayuda de **MongoDb**, **Cassandra** y **Neo4j**. Para estar mejor preparado para responder a cualquier actividad delictiva y predecirlos, es importante comprender los patrones para que se produzca un delito. Mediante varios modelos de clasificación y predicción basado en aprendizaje profundo supervisado y no supervisado; optaremos por la opción que se ajuste mejor para realizar predicciones.

Introducción

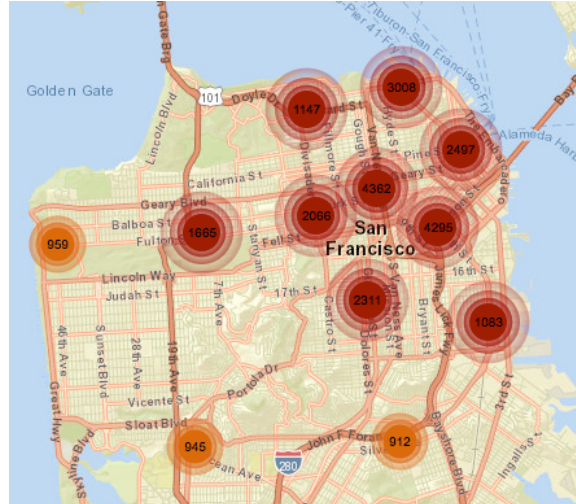
El objetivo principal del proyecto es **clasificar** la información de la actividad criminal de San Francisco dado el **tiempo** y la **localización**. Realizar un análisis y obtener un conjunto de modelos de representación de la información de éstas actividades; y que a partir de dichos modelos podamos consultar información con mejor enfoque y con valores más discretos. Para ello introduciremos las tecnologías *MongoDb*, *Cassandra* y *Neo4j* y obtener diferentes perspectivas de proceso y manejo de la información de forma respectiva.

En el resultado final la información disponible no solo se basa de la información en bruto obtenida a partir de un volumen masivo de datos correspondientes a incidencias desde el 2013 hasta la actualidad; sino que además al ajustar un modelo de predicción se podrá encontrar datos futuros; y la mejor manera es visualizar dichas predicciones sobre la información presente.

I. Dataset (Volumen de datos en bruto)

Nuestro dataset está públicamente disponible desde el portal del departamento Policial de *San Francisco*. Contiene datos de todas las incidencias generadas por

actividades criminales producidas desde el año 2003 hasta la actualidad. De forma concreta, esta información se encuentra expuesta de forma pública desde el sistema de actualización diaria del *SFPD Crime Incident Reporting* (sistema de reportes del departamento policial) a través de su plataforma Socrata.



La información está representada por un conjunto de 2 millones de filas, que contiene información detallada de actividades criminales (o incidencias) representada por: *tipo de delito, resolución, distrito, coordenadas y dirección de la zona, n° de incidencia vinculada*; todos producidos entre el 2003 hasta la actualidad.

- ▶ **Dates** - fecha y hora del incidente del crimen
- ▶ **Category** - categoría del incidente criminal
- ▶ **Descript** - descripción detallada del incidente del crimen
- ▶ **Dayoftheweek** de la semana: el día si la semana
- ▶ **PdDistrict**- nombre del distrito del departamento de policía
- ▶ **Resolution**: cómo se resolvió el incidente del crimen
- ▶ **Address**: dirección aproximada del incidente delictivo
- ▶ X - longitud
- ▶ Y - latitud

En el caso de las **coordenadas** (x,y). La X y la Y dan esencialmente el parámetro de ubicación. Es un dato interesante que veremos aprovechar mediante tecnología que proporciona **MongoDb** con respecto a geolocalizaciones.

La **fecha** se agrupa como una fecha, hora y día de la semana. Esto se divide para su uso posterior en las siguientes secciones.

150060275;NON-CRIMINAL;LOST PROPERTY;Monday;2015-01-19
14:00:00;19;2015;01;14;MISSION;NONE;18TH ST / VALENCIA ST;-
122.42158168137;37.7617007179518;(37.7617007179518, -
122.42158168137);15006027571000

150098210;ROBBERY;ROBBERY, BODILY FORCE;Sunday;2015-02-01
15:45:00;01;2015;02;15;TENDERLOIN;NONE;300 Block of LEAVENWORTH
ST;-122.414406029855;37.7841907151119;(37.7841907151119, -
122.414406029855);15009821003074

II. Fases

A partir de la estructura de datos definida para el dataset se llevará a cabo una serie de procesos y fases, que nos permitan almacenar dicha información desde distintas perspectivas de almacenaje y representación masiva de datos

- ▶ **Mongodb**, como manejo de información a través de documentos
- ▶ **Cassandra**, con potencial residente en la arquitectura para el volumen de datos
- ▶ **Neo4j**, como orientación en la búsqueda de relaciones entre los datos

Fundamentos teóricos/prácticos

En este apartado inicialmente se discutirán las capacidades en las fases de pre procesamiento, análisis y estructura de la información de cada tecnología, conectores usados para realizar la representación de la información.

Estructura y arquitectura de la información

En este apartado se realiza un desglose teórico de cómo cada modelo de base de datos correspondiente a las tecnologías usadas dispone la información, y de cómo la arquitectura se basa.

I. Cassandra

II. Neo4j

III. Mongodb

I. Cassandra

Cassandra es una base de datos dependiente del *caso de uso*. En la mayoría de los casos, una simple instancia de *MySQL* o *PostgreSQL* haría mejor el trabajo. La

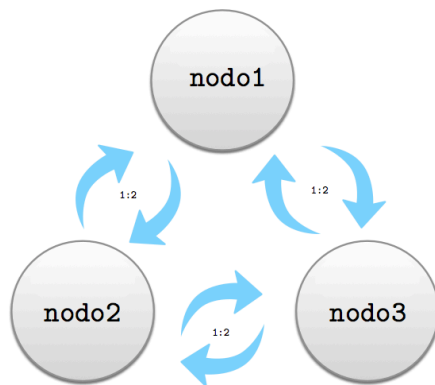
idea principal es que encontremos que Cassandra pueda ofrecer facilidades con respecto a la organización de los **atributos** en nuestro modelo de datos.

Estructura de la información. Unas de las características principales de Cassandra es la separación entre **nivel físico** y **lógico**.

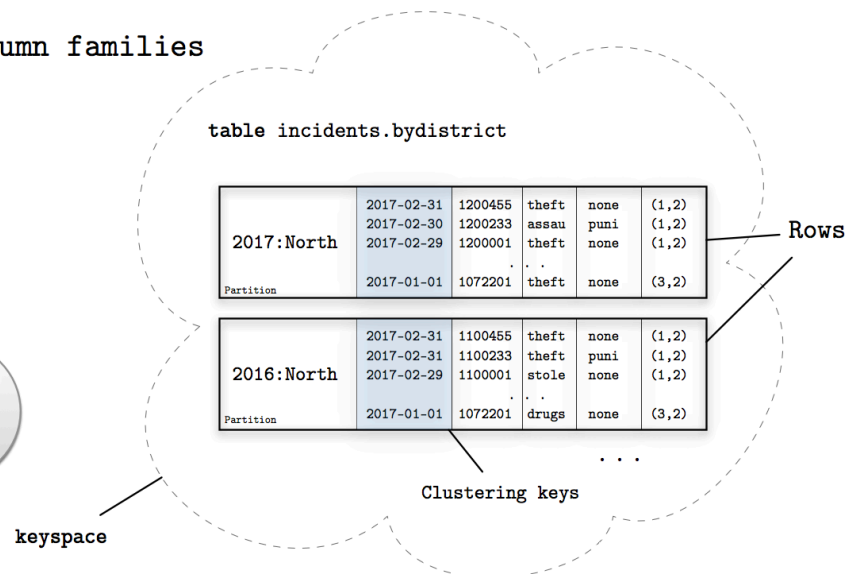
Nivel físico

```
create keyspace if not exists incidents
WITH replication =
{
  'class': 'SimpleStrategy',
  'replication_factor': '2'
};
```

Arquitectura



Column families



- **Column families** son abstracciones lógicas unitarias que conforman las tablas físicas de CQL. Es la forma en la que disponemos la información, el esquema de organización de los campos y valores, mediante distintos tipos de columnas.
- **Particiones.** Define la estructura de compactación que engloban parte de la información de una tabla en concreto, y que las filas que la componen tienen en común el mismo criterio lógico dado por una/s *clave/s de partición*. Las filas que componen cada partición pueden estar ordenadas dependiendo del criterio de ordenación mediante definición de columnas como *claves clustering*.
- **Rows** Son las unidades de forman las column families y son instancias de las particiones. Una partición puede contener una gran cantidad de rows. Por cada partición, se pueden definir la ordenación de las filas que lo

componen.

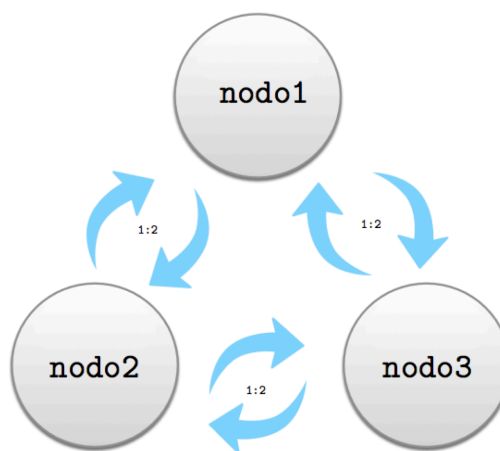
En nuestro caso, las particiones van a ser tipo **multi-row** ya que contendrán una cantidad masiva de filas ordenadas (por timestamp), y tendrán en común un determinado criterio, dado por una *composite key* (clave de partición compuesta): por distrito o por zona a parte del año, conformando claves de distinto tipo dependiendo del modelo de consulta que se requiera por la aplicación:

- Por año: 2017
- Por zona/año: 2017:north
- Por tipo de delito/año: 2017:theft

La definición de las tablas necesarias para las consultas las podemos encontrar bajo el fichero:

Con respecto a las ordenación de las filas las claves de ordenación o clustering han sido fijadas sobre las columnas claves para seleccionar filas y definir cierto criterio en la realización de cualquier consulta. Generalmente la columna `time` ha sido seleccionada en todas las opciones y de forma DESCENDENTE, con el fin de que a nivel físico sean ordenadas de *más reciente a menos reciente*; pues por usabilidad no es cuestionable dicho orden.

Arquitectura. A nivel de arquitectura, solo mencionar que los datos de las tablas son alojadas como **column families** y dicha información es compartida de forma redundante entre los nodos que compongan la red Cassandra y del grado de replicación de cómo se configure el keyspace.



II. Neo4j

Neo4j es una base de datos basada en grafos, utilizan una estructura de los lenguajes de programación conocida como *grafos*, que permite relacionar los datos a través de enlaces que facilitan el recorrido a través de ellos. Se usa para almacenar datos con relaciones complejas (por ejemplo, rutas con coordenadas GPS, relaciones sociales, ...).

Estructura de la información. Al tratarse de una base de datos diseñada a partir del modelo de grafos, Neo4j está robustamente arraigada en un sistema relacional que consta de dos tipos de objetos:

- **Nodos:** son las entidades principales del grafo y pueden contener cualquier número de propiedades en los que se almacena información a modo de tupla. Todo nodo puede ser etiquetado para asignar de esta forma un rol o índice.
- **Relaciones:** Se trata de conexiones directas entre dos nodos. Además de dirección, poseen un sentido, en el que siempre hay un nodo inicial y otro final. Esto no implica que no puedan ser consultados en cualquier sentido. Al igual que los nodos, las relaciones pueden poseer etiquetas y propiedades. Gracias a su implementación, un nodo puede tener un número elevado de relaciones sin sacrificar rendimiento.

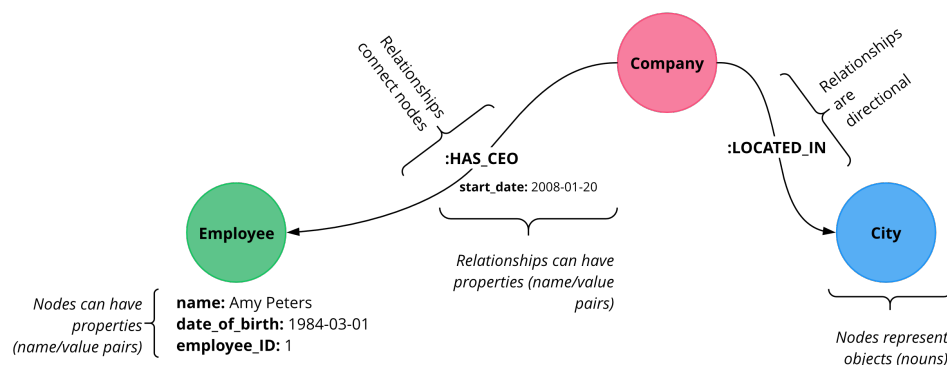


Figure 1 Ejemplo de estructura de datos en Neo4j: nodos (con tuplas como propiedades) y relaciones (cláusulas y tuplas como propiedades).

Cypher es el lenguaje de consulta declarativo que, a pesar de su simplicidad, ofrece un **alto rendimiento** a la hora de recuperar información. Su carácter declarativo centra este en qué recoger frente a cómo recuperarlo, y usa el patrón de grafos con retroceso para emparejar datos de consulta con la información disponible.

Para dichas consultas, se aplican transacciones. Una transacción incluye al menos una orden que tendrá éxito o no, no existe un punto medio de proceso que sea almacenado, por lo que no se realiza el “*commit*” hasta que dicha consulta no haya finalizado. Es especialmente útil para realizar múltiples sentencias Cypher, puesto que fallarán o tendrán éxito todas estas.

Las transacciones no finalizadas se almacenan de forma confinada en un objeto especial de estado de transacción en una hebra. Esto previene que otros hilos vean datos que no han sido escritos aún. En caso de desear retroceder, sólo hay que desechar el objeto de transacción antes de que se produzca el cambio.

Arquitectura. Neo4j almacena la información, que es representada mediante grafos, en disco, de una forma generalmente básica: se tratan de listas enlazadas de tamaño fijo. Las propiedades se guardan como listas enlazadas de registros con tuplas clave-valor, cada nodo y relación referencia su primer registro de propiedad, y los nodos al mismo tiempo también referencian el primer nodo en su cadena de relaciones. Las relaciones, por otro lado, almacenan el nodo inicial y final, además de la anterior y previa relación de cada uno de los nodos previamente mencionados, en orden.

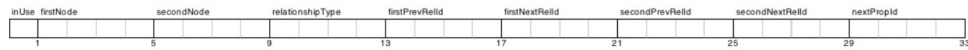
Esta estructura basada en listas y referencias garantiza la robustez de la unión entre ambos tipos de elementos y permite una estructura de datos simple, tal y como se puede apreciar en la figura anterior, basada en identificadores.

Neo4j Storage Record Layout

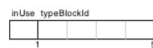
Node (9 bytes)



Relationship (33 bytes)



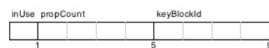
Relationship Type (5 bytes)



Property (33 bytes)



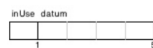
Property Index (9 bytes)



Dynamic Store (125 bytes)



NeoStore (5 bytes)



Gracias a este diseño, el almacenamiento en disco es realmente eficiente y permite la creación de millones de nodos con sus correspondientes relaciones sin sacrificar espacio en disco.

III. MongoDB

Mongodb surgió como una base de datos NoSQL orientada a documentos que es usada para el almacenamiento masivo de información. Su rendimiento se ve mejorado respecto a otras bases de datos debido a sus esquemas dinámicos que resultan similares al formato JSON, respecto a los sistemas tradicionales basados en tablas. Este esquema usado en Mongodb es conocido como BSON.

Estructura de la información. Como se ha explicado, mongo tiene una esquemática dinámica o flexible, donde los documentos almacenados por Mongodb no deben seguir ningún tipo de estructura predeterminada. Esto facilita el mapeo de dichos documentos a una entidad u objeto.

Las decisiones a la hora de organizar los modelos de datos en este tipo de base de datos se toman en base a las relaciones entre estos, para lo cual existen dos herramientas que representan dichas conexiones:

- **Referencias:** Las relaciones son almacenadas entre documentos mediante enlaces o identificadores de un documento a otro. Las aplicaciones diseñadas para trabajar con Mongodb pueden resolver dichas referencias

para encontrar los documentos referidos. En general estos modelos están normalizados.

- **Documentos empotrados:** Estos últimos capturan las relaciones entre datos almacenando dichos registros en un solo documento de mayor tamaño. Esto es posible mediante el uso de estructuras o **listas** dentro del mismo. Estos datos están desnormalizados, pero permiten recuperar toda información relevante en una misma transacción.

Además, MongoDB aplica **atomicidad** en operaciones de escritura, no pudiendo afectar a más de un documento o colección por transacción. Esto es de gran utilidad en caso de los documentos empotrados, ya que todo el modelo de relación está representado en un único documento.

Otras características de este esquema son la capacidad de actualizar los documentos añadiendo **campos** o **listas** adicionales, aumentar la velocidad de lectura integrando **índices** o creando **colecciones limitadas** para casos de documentos recientes que deben de ser recuperados o escritos con más agilidad.

Arquitectura. MongoDB está diseñado con el objetivo de ofrecer a los desarrolladores cuatro características esenciales: Disponibilidad, aislamiento de carga de trabajo, escalabilidad y localidad de datos.

MongoDB mantiene múltiples copias de la información utilizando conjuntos de **réplicas**. Estos “backups” están programados para auto-repararse e iniciar la recuperación de forma automática en caso de fallo. Un conjunto de réplicas consiste en múltiples copias de la base de datos, y para asegurar consistencia, se escoge a una copia “maestra” a la que se le aplican todos los cambios, y a la que el resto imita. Esto otorga disponibilidad al usuario.

Por otro lado, es posible utilizar esta misma estructura para generar modelos de aprendizaje o informes aplicando operaciones sobre distintas partes de un mismo clúster, evitando el impacto operacional de la aplicación generaría sobre un único nodo, lo cual aísla la carga de trabajo.

Escalabilidad. La **fragmentación** es una parte conocida de MongoDB que permite la distribución en múltiples instancias físicas para realizar un escalado horizontal de forma sencilla y así solventar problemas de espacio.

Puede realizarse sobre zonas con una gran precisión, controlando así en que clúster físico, es decir, en qué localización se encuentra la información. Esto

permite ordenar los datos para establecer una prioridad respecto a la latencia o requerimientos gubernamentales.

Comparativa

En la siguiente rubrica realizamos comparaciones del tipo de arquitectura y de la capacidad de la estructuración de la información de las tres tecnologías.

	Ventajas	Desventajas
Cassandra	<ul style="list-style-type: none">• Consistencia estable y replicación de datos. La relación entre la estructura lógica y la física permite que la información quede estructurada y organizada entre los nodos, con el fin de optimizar y almacenar la información entre ellos.• Redundancia de información. La información se encuentra organizada entre los nodos de forma redundante a su vez.• <i>CQL</i> Lenguaje de consulta de Cassandra es una forma bastante familiar para hacer consultas sobre Cassandra. Es un subconjunto de SQL y tiene muchas de las mismas características, haciendo que la transición de un RDBMS basado en SQL a Cassandra sea menos discordante.	<ul style="list-style-type: none">• Sin Consultas Ad-Hoc: La capa de almacenamiento de datos de Cassandra es básicamente un sistema de almacenamiento de clave-valor. Esto significa que debe "modelar" sus datos en torno a las consultas que desea que surjan, en lugar de en torno a la estructura de los datos en sí. Esto puede llevar a almacenar los datos varias veces de diferentes maneras para poder satisfacer los requisitos de su aplicación.• Sin agregaciones, aunque en esta versión si las usaremos (3.11): las versiones más nuevas de Cassandra tendrán soporte limitado para agregaciones con una sola partición. Esto es de uso muy limitado. Debido a que Cassandra es una tienda de valores clave, hacer cosas como SUM, MIN, MAX, AVG y otras agregaciones requieren una gran cantidad de recursos si es posible. Si hacer un análisis ad-hoc es un requisito para su aplicación, entonces Cassandra puede no ser para usted.• <i>CQL:</i> Es fácil para alguien que proviene de SQL confundirse acerca de qué es o no compatible. Esto significa una frustración adicional (costos de lectura) para los desarrolladores que no conocen las limitaciones de Cassandra.

Mongodb	<ul style="list-style-type: none"> • Escalabilidad horizontal eficiente gracias a su capacidad de “sharding” o “fragmentación”, la cual puede dividir despliegues de gran tamaño en múltiples fragmentos manejados mediante un mongos que actúa a modo de interfaz como redirigiendo las peticiones. • Simplicidad, la forma de referenciar datos no da lugar a claves foráneas, joins u otro tipo de relaciones complejas. • No es necesaria la transformación de objetos de aplicación a objetos de base de datos ya que los documentos son flexibles y aceptan cualquier tipo de propiedades. • De código libre. 	<ul style="list-style-type: none"> • El espacio en disco que ocupan los datos suele ser mayor debido a la duplicación de información, por ejemplo, cada documento guarda sus propios nombres de campos. • Sin capacidad de realizar transacciones reales, no existen rollbacks ni commits, todas las operaciones de escritura son atómicas y directas. • La ausencia de estructura en documentos deja en manos del usuario o aplicación el modelado de datos, por lo que pueden existir discrepancias entre documentos que deberían ser similares en estructura.
Neo4j	<ul style="list-style-type: none"> • Tiempos de recorrido constantes tanto en grafos de gran magnitud en profundidad o anchura, gracias a la representación sencilla de nodos y relaciones, lo cual permite escalar hasta billones de nodos en hardware estándar. • El esquema de propiedades de grafos es flexible, haciendo posible la materialización de nuevas relaciones para atajar y agilizar los datos de dominio, pudiendo escalar los cambios de negocio. • Cypher es un lenguaje declarativo y sencillo, muy similar a SQL, especialmente optimizado para grafos. 	<ul style="list-style-type: none"> • Su estructura de datos requiere de un diseño fijo, cualquier cambio en la organización de los datos de una magnitud mayor a añadir/modificar una propiedad suele requerir un rediseño completo. • Especialmente eficiente para recuperar entidades conectadas, no es útil con un número reducido de entidades pobremente relacionadas y numerosas propiedades.

Conexión a la estructura de la información: conectores

Actualmente, toda estructura de almacenamiento de datos posee algún tipo de software o interfaz de usuario oficial para el mantenimiento y gestión de la información que posee.

Esta aplicación, por lo general, está construida bajo uno de los lenguajes de programación comunes **¿Cómo consigue dicho lenguaje conectarse con la base de datos e interactuar con esta?** Mediante el uso de **driver** y **conectores**. Hoy en día, el término “driver”, en este contexto, quiere referirse a ambas funcionalidades de conexión e interacción, mientras que conector es el módulo software que permite interacción. Esto implica que un conector está implícitamente contenido en un driver de bases de datos.

Dichos drivers nos permitirán por tanto recuperar la información almacenada mediante el lenguaje de consulta característico de cada sistema. Tener acceso a las funciones de los conectores nos ofrece la posibilidad de diseñar funciones auxiliares basadas en estas, las cuales estructurarán y recuperarán los datos de manera que resulten útiles para el ámbito de la aplicación, dando lugar a **nuestros conectores personalizados**.

Requerimientos comunes

Para el uso de los conectores (drivers) será necesario tener instaladas las siguientes librerías en nuestro espacio de trabajo:

- Python 3.5
- Librerías de estructuración de datos: *numpy*, *pandas*, ...
- Para la representación gráfica: *matplotlib*, *seaborn*, *tqdm*,

Comparativa

	Ventajas	Desventajas
Cassandra	<ul style="list-style-type: none">• Permite balanceo de carga entre los nodos disponibles.	-
	<ul style="list-style-type: none">• Pool de threads para conexiones.	
	<ul style="list-style-type: none">• Permite añadir una capa de seguridad.	
	<ul style="list-style-type: none">• Conexión única con acceso al clúster de nodos en su totalidad.	

	<ul style="list-style-type: none"> • Capacidad de manejo de datos en formatos de documentos, con capacidad dinámica. • Capacidad de transformación de tipos de datos, sobre todo fechas. 	-
Mongodb	<ul style="list-style-type: none"> • Permite añadir una capa de seguridad. • Añade geo espacialidad para los datos relacionados con las coordenadas. 	
	<ul style="list-style-type: none"> • Permite cambios de configuración de usuario desde el conector. • Pool de conexiones que se sirven a las sesiones. • Posee comunicación mediante TLS y certificados. • Transacciones con simples o auto-commit, y explícitas, con capacidad de auto-reintento. 	<ul style="list-style-type: none"> • Driver inmutable, necesidad de crearlo de nuevo cuando hay cambios en la conexión.
Neo4j		

Análisis y procesamiento de datos

Metodología

Una vez localizada y decidida de cual versión de la fuente de datos partiremos, se procede a su **descarga**, **preprocesamiento** o limpieza de datos, para ello añadiremos fundamentos necesarios para realizar todo el proceso para cada tipo de representación o modelador de datos: Cassandra, Neo4j y/o Mongodb.

- **Análisis de la fuente.** Elección de la versión de la fuente de datos, siguiendo los criterios:
 - Por separación de columnas: .tsv para Cassandra, y .csv para Neo4j y Mongodb
 - Sin filtros añadidos.
- **Descarga.** Por línea de comandos o directa desde el navegador.
- **Preprocesamiento** o limpieza de datos. Mediante *bash scripting* (Cassandra) o mediante scripts de *python* (Mongodb) y *R* (Neo4j).
- **Importación al modelo de datos.** Dependerá del motor que encapsule el modelo de datos en cada caso: Neo4j, Cassandra, Y Mongodb.

Origen y fuente de datos

Análisis de la fuente y descarga. Toda la información que se muestra en el mapa de incidencias, si desactivamos los filtros por defecto, se puede descargar mediante línea de comandos gracias al portal **Socrata**, lo haremos de la siguiente

manera:

```
wget -O incidents.raw.tsv \  
"https://data.sfgov.org/api/views/tmnf-yvry/rows.tsv?accessType=DOWNLOAD&api_foundry=true"
```

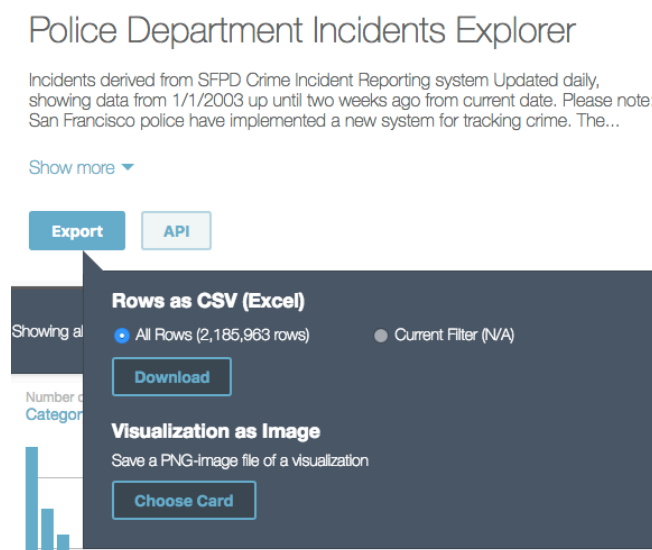
Descarga. La fuente de datos necesaria para realizar nuestro análisis de datos se puede descargar desde una página proporcionada por el sistema de datos del departamento policial, a través de la vista ofrecida por el portal de datos **Socrata Open Data**, el cual proporciona documentación para extraer reportes y las incidencias mencionadas por descargas manuales o servicios de actualizaciones a través de sus portal API. El informe sobre dicha entrada se encuentra en el portal.

Desde la página que contiene la visualización de un mapa de incidencias basada en este dataset, solamente se pueden descargar versiones del año actual. Esto ocurre desde la fecha del **3 de Marzo del 2018**, bajo indicación de la última notificación, ofrecida desde la vista Socrata para dicho reporte:

[Change Notice 03/13/2018]: [Change Notice 03/13/2018]: By the end of this month, this dataset will become historical and a new one will be created starting with incident data in 2018. This one will remain here, but no longer be updated. The new one will have data coming from a new system, will not have a 2 week lag, and have updated districts among other quality improvements. We will attach a guide here with more detailed change updates as soon as we have them.

¿Dónde podemos encontrar la versión adecuada?

El dataset, sin filtros y con información desde el 2003 hasta la actualidad (dos semanas antes de la fecha actual), se puede descargar como documento CSV/TSV a partir de una entrada al portal proporcionado por *Socrata*:



Preprocesamiento e importación al modelo de datos

La **limpieza de datos** se ha hecho de acuerdo al modelo de datos requerido para cada motor de modelado de datos: *Cassandra*, *Neo4j* o *Mongodb*; pero en general se ha seguido el mismo patrón de limpieza y han recuperado la mayoría de las columnas.

Con respecto a la **importación de datos**, para cada tecnología se ha seguido una forma de trabajar distinta en acorde al criterio seleccionado y a la normalización de ciertos atributos en acorde al modelo final.

En la fase de **diseño de la estructura** de datos, se procede con la definición del modelo de datos una vez conocidas las sentencias necesarias en la representación y visualización de la información, algo que puede ser característico para cada tipo de tecnología pero a su vez deben de ceñirse a una estrategia común: la **visualización de actividad criminal por intervalo de tiempo, distrito y por tipo de delito**, cuyo procedimiento lo podemos encontrar en el apartado de *"Resultados: Representación de la información"*.

A continuación se dispone de las fases y etapas realizadas por cada tecnología para llevar a cabo el procesamiento e importación y construir finalmente un modelo de datos.

- I. Cassandra
- II. Mongodb
- III. Neo4j
- IV. Modelo programático (Cpp)

I. Cassandra

Instalación y configuración del entorno. Para la carga de información y el despliegue del servidor de Cassandra se ha optado por ser realizado desde una distribución Linux. En el portal de Cassandra existen numerosas instrucciones de cómo realizar dicha instalación y configuración desde otras perspectivas de sistema operativo y entornos.

Requerimientos técnicos

- Java 1.8
- Cassandra 3.11.2
- DevCenter

Instalación. La descarga del servidor de *Cassandra* lo haremos a partir del comando de descarga `wget` desde una distribución Linux.

```
$ wget http://apache.rediris.es/cassandra/3.11.2/apache-cassandra-3.11.2-bin.tar.gz
$ tar -xvf apache-cassandra-3.11.2-bin.tar.gz
```

- Podemos lanzar una instancia del servidor de Cassandra mediante la ejecución del comando:

```
./apache-cassandra-3.11.2/bin/cassandra
```

- Para poder realizar consultas y realizar manipulaciones necesitamos lanzar el terminal **CQL** de la siguiente manera:

```
./apache-cassandra-3.11.2/bin/cqlsh
```

Descarga del dataset. El mapa de incidencias se puede descargar mediante línea de comandos de la siguiente manera, tal y como pudimos ver en el apartado anterior.

```
wget -O incidents.raw.tsv \
    "https://data.sfgov.org/api/views/tmnf-yvry/rows.tsv?accessType=DOWNLOAD&api_foundry=true"
```

La información en CSV (con tabulador como separador) se encuentra estructurada en bruto de la siguiente manera:

IncidentNum	Category	Descript	DayOfWeek	Date	Time	PdDistrict	Resolution	Address	...
150060275	NON-CRIMINAL	LOST PROPERTY	Monday	01/19/2015	14:00	MISSION	NONE	18TH ST /	...

Limpieza de datos. Una vez creados los esquemas de tablas necesarias para el modelo de datos procedemos al preprocesamiento de los datos descargados desde el portal de información de incidencias para trabajar con datos limpios y en acorde al formato de los atributos que utilizaremos para dichas tablas. Se realizarán los siguientes cambios con respecto a los datos en bruto:

- Formato timestamp para definir el campo 'time', a partir de las columnas 'Date' y 'Time' ('01/19/2015' y '14:00') para obtener '2015-01-19 14:00:00'.
- Campos de agrupación relacionados con periodos de tiempo: year, month, day, hour, basados en los campos anteriores. Los cuáles serán necesarios para las particiones de datos por año y/o búsqueda por horas.

(Ad-hoc) Para **descargar y realizar este limpiado de datos en el mismo lugar** lo podemos realizar desde la línea de comandos de la siguiente manera:


```
wget -O- \
  "https://data.sfgov.org/api/views/tmnf-yvry/rows.tsv?accessType=DOWNLOAD&api_foundry=true" \
  | tail -n +2 | tr '\t' ';' \
  | sed -E 's/([0-9]+)/([0-9]+)/([0-9]+):([0-9]+)/3-1-2 4:5:00;2;3;1;4/g' >
  incidents.raw.<datetime>.csv
```

o una vez descargado el dataset en 'incidents.raw.tsv' con las 2 millones de entradas, realizamos el mismo tipo de procesamiento:

```
$ nice cat incidents.raw.20180601.tsv |wc -l
2185964

$ nice cat incidents.raw.20180601.tsv |tail -n +2 | tr '\t' ';' \
  | sed -E 's/([0-9]+)/([0-9]+)/([0-9]+):([0-9]+)/3-1-2
  4:5:00;2;3;1;4/g' \
  > incidents.dataset.csv
```

A partir de aquí se procede al volcado de datos a una tabla de índole general

Importación de datos. Para realizar el volcado e importación de datos es imprescindible que se haya creado: el keyspace de trabajo y los esquemas y tablas físicas sobre la consola CSQL o desde el DevCenter. El comando utilizado para realizar el volcado se denomina COPY. Es un comando bastante simple de usar, poco flexible y dúctil para poder realizar transformaciones o selecciones específicas.

```
COPY incidents.overall(incidentId, category, description, dayoftheweek, time,
day,year,month, hour, district, resolution, address,x,y,location, subid)
FROM 'incidents.dataset.csv'
WITH DELIMITER=';'
and HEADER=false and DATETIMEFORMAT='%Y-%m-%d %H:%M:%S';
```

Para la importación de datos a los otros esquemas se sigue un proceso de importación en cadena, de manera que entre las tablas con la información volcada son tomadas como base para realizar una exportación-importación de campos selectivos.

Tabla 1 -> Export -> solo los campos necesarios para la Tabla2 (script cql temporal) <- Import <- Tabla 2

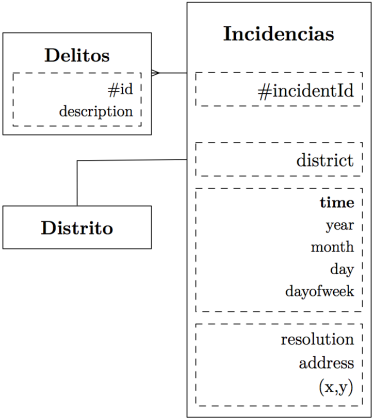
Diseño y estructura de datos. En *Cassandra*, el diseño y definición de un modelo de datos se procede una vez conocidas las metas y sentencias necesarias para la visualización final de la información a analizar. Se sugiere seguir una series de pautas para conseguir un modelado de datos idóneo para el análisis y el procesamiento masivo de datos.

Hay dos **metas importantes** a tener en cuenta para el modelado de datos en *Cassandra*:

- Definir un procedimiento para repartir la información de forma equitativa y horizontal en los *clústeres* definidos.
- Minimizar el número de lectura de particiones.

Análisis de requisitos. Se requiere obtener y visualizar la actividad criminal para *periodos de tiempo* de ciertas *zonas* de la ciudad.

Identificar entidades y relaciones. Cada línea del documento contiene información de la relación incidencia-delito. Cada incidencia se puede desglosar en varios causas delictivas o delitos. La relación natural entre las entidades mencionadas queda representada en el siguiente diagrama:



Dado que el modelo de datos se hace en acorde a las sentencias y no sobre las entidades o relaciones, no se requiere de un diseño alejado de la representación natural. En tal caso, se tendrá en cuenta la representación actual de las incidencias pero se hará para distintos escenarios (esquemas) en acorde a cada sentencia requerida.

incidentId	subid	category	time	district	address
150098373	15009837307021	VEHICLE THEFT	Sunday, 2015-02-01 12:45:00	NORTHERN NONE	FRANKLIN ST / TURK ST
150098373	15009837372000	NON-CRIMINAL	Sunday, 2015-02-01 12:45:00	NORTHERN NONE	FRANKLIN ST / TURK ST

II. Mongoddb

Instalación y configuración del entorno. Para la carga de información y el despliegue desde Mongo se necesita de la instalación del motor **MongoDB**, por vía **Compass** como suite o vía independiente.

Requerimientos técnicos

- Mongoddb
- Compass

Limpieza e importación de datos. Descargamos csv desde la web:

- **Forma 1:** Mediante la herramienta de importación de Mongoddb.

```
mongoimport -d datascience -c incidents --type csv \
--file incidents.dataset.csv --headerline
```

- **Forma 2:** Hemos escrito un script en python2, `load_data.py` que realiza la misma función.

```
#!/usr/bin/env python
import os
from pymongo import MongoClient
import pandas as pd
import json
import progressbar

STEP = 200

def import_content(filepath):
    # cdir = os.path.dirname(__file__)
    # file_res = os.path.join(cdir, filepath)
    client = MongoClient()

    # Creates the database for the San Francisco city incidents data
    db = client['datascience']

    # Creates the collection of the documents of that will represent the incidents
    incid = db.incidents
    incid.remove()

    # Reads the csv file into python's dataframe type
    csv = pd.read_csv('Incidents.csv')

    bar = progressbar.ProgressBar()
    for start in bar(range(0, len(csv.index), STEP)):
        partial_csv = csv.ix[start:(start + STEP), :]

        # Reads the dataframe as json using orient=records,
        csv_to_json = json.loads(partial_csv.to_json(orient='records'))
```

```

# We bulk all data in
incid.insert(csv_to_json)

# we free some memory as csv is not needed anymore ;)
del partial_csv
del csv_to_json

if __name__ == "__main__":
    filepath = 'incidents.dataset.csv'
    import_content(filepath)

```

III. Neo4j

Instalación y configuración del entorno. Para la carga de información y el despliegue desde Neo4j, es necesario realizar la descarga y ejecución de las suites Neo4j Desktop y RStudio.

- ▶ Neo4j Desktop (<https://neo4j.com/download>)
- ▶ R + RStudio (para la limpieza de datos)

Requerimientos técnicos

- ▶ Neo4j Desktop/Server
- ▶ Compass

Preparación de la información y limpieza de datos. El preprocesamiento se realiza con ayuda del lenguaje R. Aunque en primer lugar no era necesario, tras causar cierto error en Cypher relacionado con la ausencia de una propiedad en una operación MERGE, se procedió a buscar el motivo de este fallo en el archivo **dataset.raw.csv** descargado directamente de la fuente.



El **proceso** llevado a cabo fue el siguiente:

- ▶ En **RStudio** se importó el dataset al completo, reemplazando los valores nulos con NA. Sin embargo, al iterar sobre el dataframe, el número de filas que **complete.cases** devolvía era el mismo que el número de líneas del dataset original.

```
# R
SF_Crime_Heat_Map <- read.csv("dataset.raw.csv")
View(SF_Crime_Heat_Map)
missing <- SF_Crime_Heat_Map[complete.cases(SF_Crime_Heat_Map),]
nrow(SF_Crime_Heat_Map) == nrow(missing)
# [1] TRUE
```

- Por tanto se decidió exportar estos mismos datos pero aprovechando el tipado que se había producido en la importación para ordenar el dataframe mediante el **incidentNum**. Este sería exportado a un nuevo archivo **dataset.data.csv**.

```
# R
SF_data <- SF_Crime_Heat_Map[order(SF_Crime_Heat_Map$IncidentNum),]
View(SF_data)
write.csv(SF_data,"datasets/dataset.data.csv")
```

Este **fichero resultante** sería el utilizado para realizar el volcado al modelo de datos.

- **Nueva mejora y actualización.** Sin embargo, encontramos un problema con el formato actual. Este consiste en que el campo Date no nos proporciona acceso individual a cada variable (Día, mes y año). Obtener estas dividiendo la columna en una sentencia cipher aumenta de forma exponencial el tiempo de carga del CSV a la base de datos, por lo tanto es recomendable realizar esta operación anteriormente y modificar el archivo existente:

```
# Transformamos las columnas del data.frame para poder operar sobre ellas.
SF_Crime_Ordered_Map_2 <- data.frame(lapply(SF_Crime_Ordered_Map, as.character),
stringsAsFactors=FALSE)

# Usamos la librería stringr para mayor facilidad a la hora de separar la columna
install.packages("stringr")
library("stringr")

# Almacenamos los tres vectores resultantes
fechaSeparada <- str_split_fixed(SF_Crime_Ordered_Map_2$Date, '/',3)

# Añadimos al data.frame original las nuevas columnas y lo exportamos
SF_Crime_Ordered_Map$Day = fechaSeparada[,2]
SF_Crime_Ordered_Map$Month <- fechaSeparada[,1]
SF_Crime_Ordered_Map$Year <- fechaSeparada[,3]

write.csv(SF_Crime_Ordered_Map, "dataset.ordered.data.csv", row.names = FALSE)
```

Importación de datos. Una vez obtenido el fichero preformateado **dataset.ordered.data.csv** se procede a la importación desde el motor de Neo4j. Es necesario utilizar la sentencia LOAD CSV. Pero a continuación describimos el proceso que llevamos a cabo para realizar la importación de datos y la cadena de problemas con los que nos encontramos hasta

la solución final. Importamos con la sentencia **LOAD CSV básica** en su defecto.

```
LOAD CSV WITH HEADERS FROM 'dataset.ordered.data.csv' as line return line
```

Para importar los datos de CSV a la base de datos, sin embargo esto no genera nodos ni relaciones.

Para realizar la importación de datos tuvimos que seguir una serie de pasos hasta llegar con la solución final, a partir del data set **dataset.ordered.data.csv**:

Primera solución

Esta fue la primera sentencia **Cypher** utilizada para cargar los datos.

- El tag **WITH HEADERS** de la instrucción **LOAD CSV** nos permite referirnos a las columnas por los nombres originales situados al comienzo del csv.
- Donde **MATCH** es similar a **SELECT** y **CREATE** a **INSERT**, respectivamente, **MERGE** es una mezcla entre ambos que busca el elemento especificado, y si no lo encuentra, lo crea, lo que conlleva una búsqueda previa a la potencial inserción.
- Los elementos como *nodos* y *relaciones* se especifican con **LABELS** y **Properties**, donde los primeros actúan como tipo principal del elemento y sirven para filtrarlos más fácilmente, y los segundos almacenan datos más específicos por lo que requieren sentencias con **WHERE** para su filtrado.

```
LOAD CSV WITH HEADERS FROM 'dataset.ordered.data.csv' AS line
MERGE (i:INCIDENT { incidentNum:toInt(line.IncidentNum)})
SET ON CREATE i.description = line.Description
MERGE (c:CATEGORY { name: line.Category})
MERGE (f:DATE { date: line.Date, diaSemana: line.DayOfWeek})
MERGE (r:RESOLUTION { name: line.Resolution})
MERGE (d:DISTRICT { name:line.PdDistrict})
CREATE (i)-[t:TYPE]->(c),
(i)-[p:PLACE { address:line.Address, x:line.X, y:line.Y}]->(d),
(i)-[ti:TIME {time: line.Time}]->(f),
(i)-[s:STATUS]->(r);
```

Problema de esta sentencia

- Las sentencias **MERGE** buscan nodos con las mismas propiedades y labels antes de insertar lo que ralentiza la carga.

- La sentencia completa intenta realizarse con la memoria RAM actual, con 2M de datos por procesar.

Esto resultaba en errores de memoria o tiempos demasiado largos para la importación (una hora y media).

Solución

- Se añadió la instrucción *USING PERIODIC COMMIT* la cual almacena los cambios realizados cada 1000 líneas, por defecto.
- Se añadió la instrucción *ON CREATE SET* que condiciona una propiedad de un nodo a su creación en *Merge*, mejorando la eficiencia.
- Se incluyeron índices antes de la carga para cada nodo, mejorando la velocidad de búsqueda:

```
CREATE INDEX ON :INCIDENT(incidentNum);
CREATE INDEX ON :CATEGORY(name);
CREATE INDEX ON :DATE(day);
CREATE INDEX ON :DATE(month);
CREATE INDEX ON :DATE(year);
CREATE INDEX ON :RESOLUTION(name);
CREATE INDEX ON :DISTRICT(name);
```

- Se aumentó la memoria modificando el archivo *neo4j.conf* con estos valores:

```
dbms.memory.heap.initial_size=2G
dbms.memory.heap.max_size=4G
dbms.memory.pagecache.size=4G
```

Lo cual aumentaba la memoria máxima utilizable por Neo4j hasta 8Gb de RAM.

Solución final. Por tanto la sentencia final fue esta:

```
USING PERIODIC COMMIT 5000
LOAD CSV WITH HEADERS FROM 'datasets/dataset.ordered.data.csv' AS line
MERGE (i:INCIDENT { incidentNum:toInt(line.IncidentNum)})
SET ON CREATE i.description = line.Description
MERGE (c:CATEGORY { name: line.Category})
MERGE (f:DATE { date: line.Date})
ON CREATE SET f.dayofweek = line.DayOfWeek
MERGE (r:RESOLUTION { name: line.Resolution})
MERGE (d:DISTRICT { name:line.PdDistrict})
CREATE (i)-[t:TYPE]->(c),
(i)-[p:PLACE { address:line.Address, x:line.X, y:line.Y}]->(d),
(i)-[ti:TIME {time: line.Time}]->(f),
(i)-[s:STATUS]->(r);
```

Que importaba los datos, creaba los nodos y sus relaciones en un tiempo de **3m20s**.

Mejora definitiva

Para cargar los nuevos datos de día, mes y año, la query tuvo que cambiar a:

```
USING PERIODIC COMMIT 5000
LOAD CSV WITH HEADERS FROM 'dataset.ordered.data.csv' AS line
MERGE (i:INCIDENT { incidentNum:toInt(line.IncidentNum)})
SET ON CREATE i.description=line.Description
MERGE (c:CATEGORY { name: line.Category})
MERGE (f:DATE { day:toInteger(line.Day),
month:toInteger(line.Month),year:toInteger(line.Year)})
ON CREATE SET f.dayofweek = line.DayOfWeek
MERGE (r:RESOLUTION { name: line.Resolution})
MERGE (d:DISTRICT { name:line.PdDistrict})
CREATE (i)-[t:TYPE]->(c),
(i)-[p:PLACE { address:line.Address, x:line.X, y:line.Y}]->(d),
(i)-[ti:TIME {time: line.Time}]->(f),
(i)-[s:STATUS]->(r);
```

La cual es ciertamente más lenta, debido a que se han añadido tipados en **:INCIDENT** y **:DATE**, además de que el **MERGE** realizado en este segundo tipo de nodo debe comparar dos variables adicionales para no causar duplicidad, aumentando la duración de la carga a **12m49s**.

¿Motivo principal? La subida de datos a servidor es algo que sólo se debe realizar una vez, por tanto es absolutamente más eficiente que tarde más este paso, que ralentizar las búsquedas teniendo que filtrar la fecha con comparadores de cadenas de caracteres en cada una.

Configuración necesaria antes del import. Fue necesario tener en cuenta una serie de requisitos de configuración, por lo que también es importante recordar que el fichero ha de estar en la carpeta:

```
Application\neo4jDatabases\database-(numero)\installation-3.3.3\import
```

Pero se puede cambiar modificando el archivo neo4j.conf en la línea:

```
dbms.directories.import=import
```

Siendo "import" el directorio por defecto.

También se puede acceder a esta configuración desde la aplicación Desktop mediante:

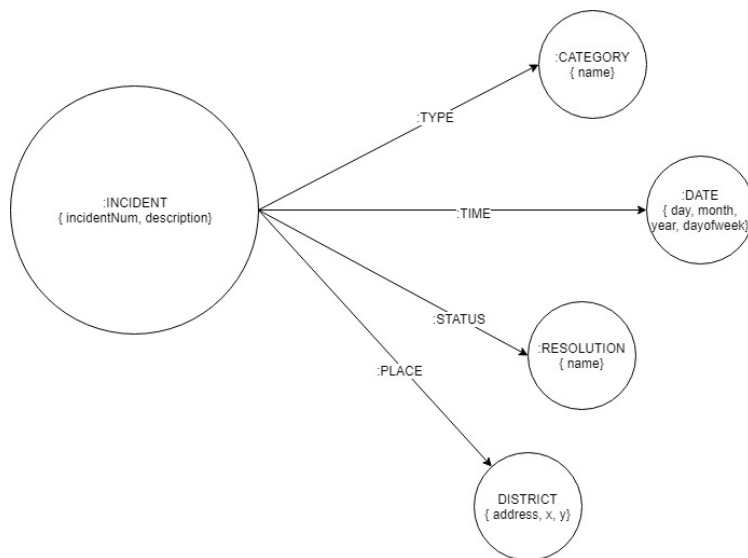
My project->(Nombre de BD)->Settings

Que provee de un editor propio.

Diseño y estructura de datos. Dada la estructura del archivo .CSV proporcionado, titulado con los siguientes **headers**:

IncidntNum, Category, Descript, DayOfWeek, Date, Time, PdDistrict, Resolution, Address, X, Y, Location.

Se ha decidido realizar este diseño de nodos y relaciones



Dado que, a excepción de los incidentes, un gran número de elementos estaban duplicados, se ha procedido a crear sus propios nodos y a evitar su inserción múltiple. Los datos que podían causar la aparición doble de un mismo valor, como un distrito con mismo nombre pero coordenadas X e Y diferentes a las de otro mismo distrito, se han añadido a las relaciones entre los nodos de incidentes y el resto. Esto aumenta la eficiencia y rendimiento final tanto de la inserción como de las búsquedas futuras.

Formato de fechas. Mientras que en el archivo sólo se guarda la fecha como cadena de caracteres en la columna *Date*, en el esquema aparecen tres campos: *Day*, *Month* y *Year*, los cuales serán obtenidos durante el apartado de preprocesamiento

IV. Modelo programático (Cpp)

La aplicación `processor` exporta la información contenida en varios ficheros con la información detallada basada en el fichero inicial TSV recién preprocesado.

Para la instalación de la aplicación solo es necesario realizar la compilación del proyecto y de la aplicación C++.

```
$ g++ -g -o processor app.cpp
$ ./processor incidents.dataset.tsv
```

Requerimientos técnicos

- ▶ G++
- ▶ Visual Studio Code (IDE)

Preprocesamiento e importación de datos. La aplicación preprocesa los datos y realiza la importación de forma automática, mediante la clase `Incidents::import(filename)`.

```
$ ./processor incidents.dataset.tsv
First Incident:GRAND THEFT FROM LOCKED AUTO
First occurrence of Filtered Incident by dayoftheweek:GRAND THEFT
FROM LOCKED AUTO,Monday
```

```
Exporting incidents count by district into filename:
incidentsByDistrict.tsv ...
Incidents groups:6
```

```
Exporting incidents count by category into filename:
incidentsByCategory.tsv ...
Incidents groups:1
```

Estructura de datos. La información se encuentra desglosada en estructuras basada en listas de estructura tipo `struct ``incident```

```
struct incident
{
    std::string id;
    std::string category;
    std::string description;
    std::string dayoftheweek;
    std::string time;
    std::string place;
    std::string resolution;
```

```
std::string address;
std::string x;
std::string y;
std::string location;
};
```

Mediante el método `Incidents::import(inputname)` el fichero CSV es procesado y transformado en un vector de estructuras tipo `incident`.

Representación de la información

Mediante las consultas y otras estructuras de representación llamadas vistas se puede obtener una visión pragmática de la información procesada. También se incluyen representaciones gráficas por lo que añadiremos todos los procedimientos llevados a cabo para realizar las presentaciones de cualquier índole. En este apartado desglosaremos este conjunto de procedimiento y pasos por tecnología, tal y como hemos ido haciendo con los apartados anteriores en acorde a las siguientes fases.

- ▶ Diseño de consultas
- ▶ Uso de conectores y construcción de consultas (python)
- ▶ Representaciones gráficas

Consultas. Hemos generalizado los tipos de consultas dependiendo a las expectativas sobre la aplicación, en acorde a los siguientes usos:

- Obtener toda las incidencias para un periodo de tiempo (rango).
- Actividad criminal por zona
- Actividad criminal por tipo de delito
- N° incidencias agrupadas por zona / *año*
- N° incidencias agrupadas por delito / *año*

Conectores. Con ayuda de los conectores (drivers) y mediante Python, se pueden realizar peticiones de las consultas requeridas en la fase de diseño.

Diseño de consultas (*Identificar sentencias*). La mejor manera de particionar las lecturas es modelar los datos en acorde a las sentencias, que son dadas por los requisitos. De primera instancia hay que considerar los siguientes puntos antes de realizar la definición de las sentencias:

- Agrupación por un atributo: por distrito, por tipo de actividad criminal

- Ordenación por un atributo: por tiempo
- Filtro basado en un conjunto de condiciones.

A partir de aquí, desde otra perspectiva de alto nivel, nuestro objetivo consistirá en clasificar las requeridas incidencias y recuperar la información de la actividad criminal (cantidad/frecuencia), en *periodos de tiempo*, de forma *ordenada*, y dada ciertas condiciones definidas: por *distrito* o *tipo de delito*.

La **visualización de la actividad criminal** se consigue mediante la representación de las incidencias producidas para cada atributo: periodo de tiempo, por cada zona ó tipo de delito. Para una perspectiva de análisis un poco más precisa, se necesitaría representar la cuantificación mediante uso de proporciones o frecuencias para un determinado grupo de atributos o condiciones.

- Frecuencia criminal por periodos de tiempo: horas o día de la semana para cierto año.
- Frecuencia criminal por cada zona.
- Proporción de la incidencia de diferentes actividades delictivas.

Todo el procedimiento de diseño y construcción mediante python lo encontraremos explicado de forma detallada para cada tecnología:

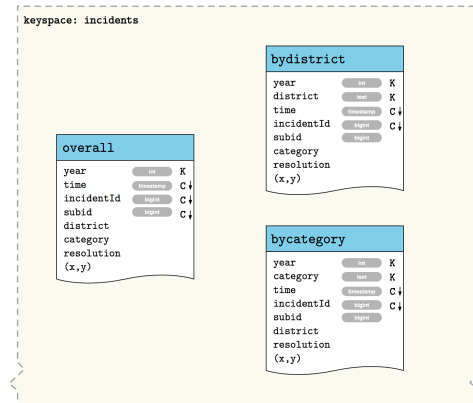
- I. Cassandra
- II. Neo4j
- III. MongoDB
- IV. Processor (aplicación C++)

I. Cassandra

Nuestro modelo lógico contiene tres tablas des normalizadas para admitir consultas de incidencias por año, zona, tipo de delito y fecha. A medida que trabajamos para implementar estos diseños por consultas, querremos considerar si debemos realizar la administración con la des normalización de forma manual o usar la capacidad de extrapolar las consultas dentro de vistas en Cassandra, con ayuda de las consultas de agregación.

El diseño que se muestra para el espacio de incidencias en la siguiente figura utiliza ambos enfoques. Elegimos implementar **incidents.overall** y **incidents.bydistrict** y **incidents.bycategor** y como **tablas regulares**.

El razonamiento detrás de esta elección de diseño momentáneamente, es que gracias a las claves de partición compuestas entre zona o tipo de delito con el año, podemos dividir y dimensionar mejor las consultas sobre todo cuando existe una base de información masiva.



Conexión con la base de datos. *El connector o driver para Cassandra se denomina `cassandra.cluster`. Contiene una clase principal denominada `Cluster` que se conecta a un cluster de Cassandra estableciendo una conexión encapsulada en el objeto `Session`.*

Se pueden añadir ciertas configuraciones en acorde a la arquitectura definida o a la forma de conexión. Toda la información del proceso de conexión con la base de datos para la obtención de consultas lo podemos encontrar bajo el notebook:

[Análisis-Cassandra.ipynb](#).

Instalación y configuración. Una vez instalado un entorno amigable a Python 3.5 necesitamos instalar las librerías necesarias:

```
pip install cassandra-driver
pip install django-cassandra-engine
```

Conexión y sesión con la base de datos. Para realizar la conexión necesitamos utilizar las clases pertenecientes de la librería `cassandra-driver`.

```
from cassandra.cluster import Cluster
```

Una vez definida se puede permitir realizar conexiones y consultas de forma genérica hemos implementado un conjunto de funciones auxiliares.

- La función **get_session** encapsula la conexión sobre nuestro "keyspace" generando el objeto `Session` característico del conector de `cassandra`.

```
def get_session(keyspace):
    """Obtiene el conector con la sesión actual al keyspace indicado."""
    return (Cluster(['cassandra.vrandkode.net']).connect(keyspace))
```

- La función **q** es una abreviatura de query y permite acondicionar la realización de consultas y parametrizar valores sobre ellas. El resultado es un objeto tipo **data frame**, por lo cual estaría totalmente adaptado al entorno y sin necesidad de ser característico para el tipo de conector o base de datos.

```
def q(session, query, **kwargs):
    """Función para encapsular las queries producidas en fmt dataframe."""
    return pd.DataFrame([row for row in
        session.execute(query.format(**kwargs))])
```

Existen otras **funciones auxiliares**, que son necesarias para agilizar el tratamiento de las consultas sobre todo en la parte de parametrización.

- Función auxiliar para permitir un valor por defecto en el timestamp de las queries si no se introduce un valor válido. Por defecto devuelve la fecha actual como timestamp

```
def nowOrdate(date=None)
```

- Función que obtiene el operador = o in junto con los valores dependiendo si el objeto es una lista o no.

```
def eqOrInIntegers(obj=None)
```

- Función que obtiene el operador = o in junto con los valores dependiendo si el objeto es una lista o no.

```
def eqOrIn(obj=None)
```

Representación e implementación de consultas y vistas

Todas las consultas se pueden encontrar en el script CQL [cql/queries.cql](#), pero para realizarlas es necesario crear el esquema de la estructura en la base de datos de *Cassandra* desde la consola a partir de los esquemas en [cql/schema.cql](#).

1. Crear keyspace
2. Crear tablas e volcar los dataset mediante el comando COPY

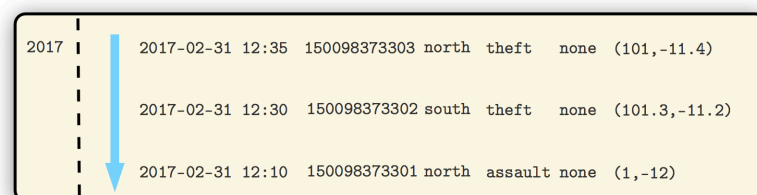
Existen distintos grupos de consultas a realizar, en acorde a los atributos de tiempo, distrito y tipo de delito.

1. Periodo de tiempo
2. Por zona

1. Actividad criminal para un *periodo de tiempo*

Diseño. La actividad es ofrecida por la tabla: *incidents.overall*, con la estructura:

Partition keys	<i>year</i>
Clustering keys	<i>time,incidentid</i>



2017	2017-02-31 12:35	150098373303	north	theft	none	(101,-11.4)
	2017-02-31 12:30	150098373302	south	theft	none	(101.3,-11.2)
	2017-02-31 12:10	150098373301	north	assault	none	(1,-12)

La **información** se encuentra particionada por una clave primaria compuesta de las claves: de partición *año*; pero al añadir el campo *time* como clave de clusterización podemos realizar una búsqueda por periodo. No se podría considerar una consulta muy eficiente ya que no se aprovecha las ventajas de particionamiento con respecto a la condición de búsqueda.

```
CREATE TABLE incidents.overall (  
    year int,  
    time timestamp,  
    ...  
PRIMARY KEY ((year), time, incidentId, subid)  
) WITH CLUSTERING ORDER BY (time DESC, incident ASC, subId ASC);
```

Obtener toda las incidencias para un periodo de tiempo.

```
select * from incidents.overall  
where time >= '2014-01-01 00:00:00' and time <= dateof(now())  
and yearh = 2017  
allow filtering;
```

```
cqlsh:incidents> select * from incidents.overall where time >= '2014-01-01 00:00:00' and time <= dateof(now()) limit 10 allow filtering;
```

incidentid	subid	time	address	category	day	dayoftheweek	description
150098919	15009891907041	2015-02-01 19:53:00.000000+0000	MCALLISTER ST / LEAVENWORTH ST	RECOVERED VEHICLE	1	Sunday	
150098947	15009894727170	2015-02-01 20:00:00.000000+0000	3400 Block of 16TH ST	OTHER OFFENSES	1	Sunday	
150098618	15009861806153	2015-02-01 18:00:00.000000+0000	GEARY ST / STOCKTON ST	LARCENY/THEFT	1	Sunday	
150098743	15009874307021	2015-01-28 16:00:00.000000+0000	LINCOLN WY / 38TH AV	VEHICLE THEFT	28	Wednesday	
150098602	15009860227130	2015-02-01 18:09:00.000000+0000	FRANKLIN ST / SUTTER ST	OTHER OFFENSES	1	Sunday	
150098599	15009859965016	2015-02-01 19:15:00.000000+0000	14TH ST / FOLSOM ST	OTHER OFFENSES	1	Sunday	DRIVER
150098414	15009841404136	2015-02-01 17:05:00.000000+0000	PACIFIC AV / GRANT AV	ASSAULT	1	Sunday	
150098947	15009894726210	2015-02-01 20:00:00.000000+0000	3400 Block of 16TH ST	ASSAULT	1	Sunday	THREAT
150098630	15009863064010	2015-02-01 18:38:00.000000+0000	1000 Block of FOLSOM ST	NON-CRIMINAL	1	Sunday	
150098862	15009886204134	2015-02-01 18:05:00.000000+0000	400 Block of RHODEISLAND ST	ASSAULT	1	Sunday	

(10 rows)

Implementación. Se han implementado funciones necesarias para obtener cada una de las vistas de las consultas diseñadas:

- Obtener incidencias (general)
- Obtener total de actividad criminal (getIncidents)
- Búsqueda de incidencia (getIncident)
- En un rango/periodo de tiempo (getIncidentsByDate)

Ejemplos de llamadas a consultas por funciones mediante Python.

Gracias a las funciones mencionadas se pueden realizar consultas que previamente fueron diseñadas y ejecutadas directas a la base de datos. Mostraremos alguna de ellas:

- Obtener actividad criminal dado un identificador de incidencia

```
getIncident(conn, incidentId = 140009459)
```

	year	time	incidentid	subid	address	category	day	dayoftheweek	description	district
0	2014	2014-01-04 03:52:00	140009459	14000945926030	SACRAMENTO ST / POLK ST	ARSON	4	Saturday	ARSON	NORTHERN

- Obtener la actividad criminal en general (límite de 5 filas)

```
getIncidents(conn, limit=5)
```


	year	time	incidentid	subid	address	category	day	dayoftheweek	description	dist
0	2014	2014-01-04 03:52:00	140009459	14000945926030	SACRAMENTO ST / POLK ST	ARSON	4	Saturday	ARSON	NOF
1	2015	2015-02-04 20:27:00	150098577	15009857763010	800 Block of BRYANT ST	WARRANTS	4	Wednesday	WARRANT ARREST	SOL
2	2015	2015-02-01 20:35:00	150098997	15009899716010	700 Block of MARKET ST	DRUG/NARCOTIC	1	Sunday	POSSESSION OF MARIJUANA	SOL
3	2015	2015-02-01 20:35:00	150098997	15009899703414	700 Block of MARKET ST	ROBBERY	1	Sunday	ATTEMPTED ROBBERY ON THE STREET WITH BODILY FORCE	SOL
4	2015	2015-02-01 20:30:00	150098969	15009896903074	PALOU AV / QUINT ST	ROBBERY	1	Sunday	ROBBERY, BODILY FORCE	BAY

- Obtener la actividad criminal desde el 2 Enero del 2015 (límite de 5 filas)

```
getIncidentsByDate(conn, since="2015-01-02", limit = 5)
```

	year	time	incidentid	subid	address	category	day	dayoftheweek	description
0	2015	2015-02-04 20:27:00	150098577	15009857763010	800 Block of BRYANT ST	WARRANTS	4	Wednesday	WARRANT ARREST
1	2015	2015-02-01 20:35:00	150098997	15009899716010	700 Block of MARKET ST	DRUG/NARCOTIC	1	Sunday	POSSESSION OF MARIJUANA
2	2015	2015-02-01 20:35:00	150098997	15009899703414	700 Block of MARKET ST	ROBBERY	1	Sunday	ATTEMPTED ROBBERY ON THE STREET WITH BODILY FORCE
3	2015	2015-02-01 20:30:00	150098969	15009896903074	PALOU AV / QUINT ST	ROBBERY	1	Sunday	ROBBERY, BODILY FORCE
4	2015	2015-02-01 20:26:00	150098975	15009897564020	1300 Block of REVERE AV	NON-CRIMINAL	1	Sunday	AIDED CASE, MENTAL DISTURBED
5	2015	2015-02-01 20:15:00	150098953	15009895304134	1000 Block of GOETTINGEN ST	ASSAULT	1	Sunday	BATTERY
6	2015	2015-02-01 20:00:00	150098947	15009894728160	3400 Block of 16TH ST	VANDALISM	1	Sunday	MALICIOUS MISCHIEF, VANDALISM OF VEHICLES

2. Actividad criminal por zona

Para esta sentencia si se realiza una partición de datos adecuada, con respecto a la zona y año: *district:year*. La actividad es ofrecida por la tabla: *incidents.overall*, con la estructura:

Partition keys	<i>year, district</i>
Clustering keys	<i>time,...</i>

- Obtener información de incidencias por zonas (para un determinado año)

```
select district, year, incidentid, category, time, location
from incidents.bydistrict
where year = ?
```

```
cqlsh:incidents> select district, year, incidentid, category, time, location from incidents.bydistrict where year=2015 allow filtering;
```

district	year	incidentid	category	time	location
CENTRAL	2015	150098618	LARCENY/THEFT	2015-02-01 18:00:00.000000+0000	(37.7875673013055, -122.406592269796)
CENTRAL	2015	150098414	ROBBERY	2015-02-01 17:05:00.000000+0000	(37.7969028838908, -122.406831986427)
CENTRAL	2015	150098492	MISSING PERSON	2015-02-01 15:30:00.000000+0000	(37.7951450868486, -122.406478762416)
CENTRAL	2015	150098884	LARCENY/THEFT	2015-02-01 09:30:00.000000+0000	(37.803594011382, -122.416489414289)
CENTRAL	2015	150098200	LARCENY/THEFT	2015-01-31 17:00:00.000000+0000	(37.786257854865, -122.417289322526)
CENTRAL	2015	150098254	BURGLARY	2015-01-31 16:09:00.000000+0000	(37.7878092959561, -122.40656817787)
CENTRAL	2015	150098583	NON-CRIMINAL	2015-01-31 01:15:00.000000+0000	(37.7969934198176, -122.40354422656)
CENTRAL	2015	150098931	MISSING PERSON	2015-01-29 12:00:00.000000+0000	(37.794611817992, -122.403994862674)
CENTRAL	2015	150098652	OTHER OFFENSES	2015-01-05 00:01:00.000000+0000	(37.7915661006349, -122.409144009353)
MISSION	2015	150098947	DRUNKENNESS	2015-02-01 20:00:00.000000+0000	(37.7643418581632, -122.430494795393)

Si queremos añadir condicion de periodo de tiempo, necesitamos añadir **filtering**:

```
select district, year, incidentid, category, time, location
from incidents.bydistrict
where year = 2015 and time >= '2015-02-01 00:00:00' and time <=
dateof(now())
allow filtering;
```

Implementación. Se han implementado funciones necesarias para obtener cada una de las vistas de las consultas diseñadas por zona:

- Obtener total de actividad criminal por distrito (*getCountByDistrict*)
- Obtener actividad criminal agrupado por distrito/año (*getByDistrict*)

Ejemplos de llamadas a consultas por funciones mediante Python.

Gracias a las funciones mencionadas se pueden realizar consultas que previamente fueron diseñadas y ejecutadas directas a la base de datos. Mostraremos alguna de ellas:

- Obtener numero de incidencias por distrito (en un año determinado)

```
getCountByDistrict(conn,
limit=10)
```

	district	count
0	PARK	2
1	MISSION	12
2	NORTHERN	7
3	TARAVAL	5
4	INGLESIDE	2
5	TENDERLOIN	7
6	CENTRAL	9
7	SOUTHERN	8
8	BAYVIEW	9
9	RICHMOND	6

```
getCountByDistrict(conn,
year=[2014], limit=10)
```

	district	count
0	NORTHERN	1

- Obtener incidencias agrupadas por distrito (en un año determinado)

```
getByDistrict(conn, limit=10)
```

	district	year	incidentid	category	time	location
0	PARK	2015	150098395	LARCENY/THEFT	2015-02-01 14:30:00	(37.7671999403456, -122.458638758608)
1	PARK	2015	150098470	LARCENY/THEFT	2015-01-30 22:00:00	(37.7501301863303, -122.446483988175)
2	MISSION	2015	150098947	VANDALISM	2015-02-01 20:00:00	(37.7643418581632, -122.430494795393)
3	MISSION	2015	150098599	OTHER OFFENSES	2015-02-01 19:15:00	(37.7685360123583, -122.41561633832)
4	MISSION	2015	150098981	LARCENY/THEFT	2015-02-01 19:00:00	(37.7650244301204, -122.41920245941)
5	MISSION	2015	150098527	WEAPON LAWS	2015-02-01 17:02:00	(37.7666737551835, -122.419827929961)
6	MISSION	2015	150098458	OTHER OFFENSES	2015-02-01 16:56:00	(37.764228935718, -122.419520367886)
7	MISSION	2015	150098367	ROBBERY	2015-02-01 16:20:00	(37.7651107322703, -122.432198022433)
8	MISSION	2015	150098856	NON-CRIMINAL	2015-02-01 15:30:00	(37.7564864109309, -122.406539115148)
9	MISSION	2015	150098345	WARRANTS	2015-02-01 14:00:00	(37.7690748003847, -122.413354187018)

```
getCountByDistrict(conn, limit=10)
```

	district	count
0	PARK	2
1	MISSION	12
2	NORTHERN	7
3	TARAVAL	5
4	INGLESIDE	2
5	TENDERLOIN	7
6	CENTRAL	9
7	SOUTHERN	8
8	BAYVIEW	9
9	RICHMOND	6

3. Actividad criminal por tipo de delito

Para esta sentencia si se realiza una partición de datos adecuada, con respecto al tipo de incidencia y año: *category:year*.

Partition keys	<i>year, category</i>
Clustering keys	<i>time,...</i>

- Obtener información de incidencias por categorias (para un determinado año)

```
select category, year, incidentid, category, time, location
from incidents.bycategory
where year = ?
```

```
cqlsh:incidents> select district, year, incidentid, category, time, location from incidents.bydistrict where year=2015 allow filtering;
```

district	year	incidentid	category	time	location
CENTRAL	2015	150098618	LARCENY/THEFT	2015-02-01 18:00:00.000000+0000	(37.7875673013055, -122.406592269796)
CENTRAL	2015	150098414	ROBBERY	2015-02-01 17:05:00.000000+0000	(37.7969028838908, -122.406831986427)
CENTRAL	2015	150098492	MISSING PERSON	2015-02-01 15:30:00.000000+0000	(37.7951450868486, -122.406478762416)
CENTRAL	2015	150098884	LARCENY/THEFT	2015-02-01 09:30:00.000000+0000	(37.803594011382, -122.416489414289)
CENTRAL	2015	150098260	LARCENY/THEFT	2015-01-31 17:00:00.000000+0000	(37.7862578545865, -122.417295322526)
CENTRAL	2015	150098254	BURGLARY	2015-01-31 16:09:00.000000+0000	(37.7878092959561, -122.40656817787)
CENTRAL	2015	150098583	NON-CRIMINAL	2015-01-31 01:15:00.000000+0000	(37.7969934198176, -122.40354422656)
CENTRAL	2015	150098931	MISSING PERSON	2015-01-29 12:00:00.000000+0000	(37.794611817992, -122.403994862674)
CENTRAL	2015	150098652	OTHER OFFENSES	2015-01-05 00:01:00.000000+0000	(37.7915661006349, -122.409144009353)
MISSION	2015	150098947	DRUNKENNESS	2015-02-01 20:00:00.000000+0000	(37.7643418581632, -122.430494795393)

- Nº incidencias agrupadas por año

Para obtener número de incidencias producidas de cierta categoria (por un determinado año). Sin añadir filtro, más eficiente.

```
select category, year, count(*)
from incidents.bycategory
where year = ?
group by category;
```

```
cqlsh:incidents> select category, year, count(*) from incidents.bycategory where year = 2015 group by category;
```

category	year	count
WEAPON LAWS	2015	2
BURGLARY	2015	2
DRUG/NARCOTIC	2015	4
DRUNKENNESS	2015	1
SUSPICIOUS REC	2015	3
LARCENY/THEFT	2015	14
WARRANTS	2015	4
SECONDARY CODES	2015	3
DRIVING UNDER THE INFLUENCE	2015	1
MISSING PERSON	2015	2
ASSAULT	2015	10
VANDALISM	2015	3
ROBBERY	2015	8
OTHER OFFENSES	2015	14
NON-CRIMINAL	2015	13
RECOVERED VEHICLE	2015	1
VEHICLE THEFT	2015	4

- Nº incidencias agrupadas por año

```
select category, year, count(*)
from incidents.bycategory
where year = 2015 and month = 1
group by category
allow filtering;
```

```
cqlsh:incidents> select category, year, count(*) from incidents.bycategory where year = 2015 and month = 1 group by category allow filtering;
```

category	year	count
BURGLARY	2015	2
LARCENY/THEFT	2015	4
SECONDARY CODES	2015	1
MISSING PERSON	2015	1
ASSAULT	2015	1
VANDALISM	2015	2
ROBBERY	2015	1
OTHER OFFENSES	2015	2
NON-CRIMINAL	2015	2
VEHICLE THEFT	2015	1

Implementación. Se han implementado funciones necesarias para obtener cada una de las vistas de las consultas diseñadas por tipo de delito:

- ▶ Obtener total de actividad criminal por categoria (getCountByCategory)
- ▶ Obtener actividad criminal agrupado por category/año (getRangeByCategory) permitiendo consultar un rango de fechas.

Ejemplos de llamadas a consultas por funciones mediante Python.

Gracias a las funciones mencionadas se pueden realizar consultas que previamente fueron diseñadas y ejecutadas directas a la base de datos. Mostraremos alguna de ellas:

- Obtener total de actividad criminal agrupado por categoria

```
getCountByCategory(conn, limit = 10)
```

	category	year	count
0	WEAPON LAWS	2015	2
1	BURGLARY	2015	2
2	DRUG/NARCOTIC	2015	4
3	DRUNKENNESS	2015	1
4	SUSPICIOUS OCC	2015	3
5	LARCENY/THEFT	2015	14
6	WARRANTS	2015	4
7	SECONDARY CODES	2015	3
8	DRIVING UNDER THE INFLUENCE	2015	1
9	MISSING PERSON	2015	2

- Obtener total de actividad criminal agrupado por categoria a partir del 1 Enero de 2015

```
getRangeByCategory(conn, limit = 10, since= "2015-02-01")
```

	category	year	incidentid	category_	time	location
0	WEAPON LAWS	2015	150098420	WEAPON LAWS	2015-02-01 17:10:00	(37.784696907904, -122.413609328985)
1	WEAPON LAWS	2015	150098527	WEAPON LAWS	2015-02-01 17:02:00	(37.7666737551835, -122.419827929961)
2	DRUG/NARCOTIC	2015	150098997	DRUG/NARCOTIC	2015-02-01 20:35:00	(37.7871160984672, -122.403919148357)
3	DRUG/NARCOTIC	2015	150098527	DRUG/NARCOTIC	2015-02-01 17:02:00	(37.7666737551835, -122.419827929961)
4	DRUG/NARCOTIC	2015	150098458	DRUG/NARCOTIC	2015-02-01 16:56:00	(37.764228935718, -122.419520367886)
5	DRUG/NARCOTIC	2015	150098345	DRUG/NARCOTIC	2015-02-01 14:00:00	(37.7690748003847, -122.413354187018)
6	DRUNKENNESS	2015	150098947	DRUNKENNESS	2015-02-01 20:00:00	(37.7643418581632, -122.430494795393)
7	SUSPICIOUS OCC	2015	150098765	SUSPICIOUS OCC	2015-02-01 19:06:00	(37.7701099298175, -122.420010175609)
8	SUSPICIOUS OCC	2015	150098680	SUSPICIOUS OCC	2015-02-01 18:37:00	(37.7724556440219, -122.416305723264)
9	SUSPICIOUS OCC	2015	150098787	SUSPICIOUS OCC	2015-02-01 18:17:00	(37.7260849566228, -122.409528258798)

```
getRangeByCategory(conn, limit = 100, since= "2015-02-01", category=["WEAPON LAWS", "DRUNKENNESS"])
```

	category	year	incidentid	category_	time	location
0	DRUNKENNESS	2015	150098947	DRUNKENNESS	2015-02-01 20:00:00	(37.7643418581632, -122.430494795393)
1	WEAPON LAWS	2015	150098420	WEAPON LAWS	2015-02-01 17:10:00	(37.784696907904, -122.413609328985)
2	WEAPON LAWS	2015	150098527	WEAPON LAWS	2015-02-01 17:02:00	(37.7666737551835, -122.419827929961)

II. Neo4j

Conexión con la base de datos. El conector **neo4j-driver** está basado en el driver oficial de **Neo4J** para **Python**. Basándose en este último, se ha creado un nuevo conector más avanzado preparado para la recolección específica de datos con el objetivo de ofrecer una mayor transparencia acerca de los objetos con los que se trabaja y mejorar la facilidad de acceso a la información consultada.

neo4jConnector - GraphDatabase – La clase **neo4JConnector** permite la creación de un objeto que encapsula todas las funciones que ofrece el driver de **GraphDatabase**, el cual procede de **neo4j.v1**, la librería que contiene las funciones para realizar la conexión. Se ha elegido encapsular dicho conector para abstraer los detalles sobre el procedimiento de autenticación y así simplificar el proceso. De esta forma se reduce el ámbito a la sección de real importancia: La recuperación de datos en un formato útil.

Se pueden añadir ciertas configuraciones en acorde a la arquitectura definida o a la forma de conexión. Toda la información del proceso de conexión con la base de datos para la obtención de consultas lo podemos encontrar bajo el notebook:

[Análisis-Neo4j](#).

Instalación y configuración. Una vez instalado un entorno amigable a Python 3.5 necesitamos instalar las librerías necesarias:

```
pip install neo4j-driver
```

Conexión y sesión con la base de datos mediante el driver. En Neo4j existen dos tipos de entidades sobre las que se basa el sistema de almacenamiento de datos: *nodos* y *relaciones*. Lo que significa que dentro del propio driver se han creado clases específicas para estos elementos.

```
from neo4jConnector import Neo4JConnector
```

Una vez definida se puede permitir realizar conexiones y consultas de forma genérica hemos implementado un conjunto de funciones auxiliares. Gracias a la nueva clase, no es necesario especificar los parámetros, y con la URI específica de neo4j se reducen a tres argumentos por llamada de conexión.

En cada función de *Neo4JConnector* se utiliza el driver para obtener una sesión y ejecutar las llamadas a la base de datos. La existencia de este objeto nos libera de la necesidad de re-declarar funciones estáticas para las distintas consultas que se deseen hacer, y pone énfasis en qué datos se han de adquirir.

```
session = Neo4JConnector('bolt://neo4j.vrandkode.net:7687', 'test', '4321')
```

Caso de uso de sesión: ejemplo de consulta

Debido al wrapper, las consultas pueden no requerir de conocer el lenguaje Cypher en el que están basadas estas últimas para Neo4j. Sin embargo, tal y como se especificó en la documentación, es posible realizar consultas especializadas también.

Para este caso, se seleccionan incidentes con una limitación en cantidad sin ningún tipo de filtro, lo que nos devuelve los primeros cinco nodos que poseen dicha etiqueta.

```
# Un ejemplo de llamada: Queremos 5 nodos de incidentes
record = session.select_limit('INCIDENT',5)
pd.DataFrame([row for row in record])
```

	N.description	N.incidentNum	N.labels	id
0	WARRANT ARREST	3979	{INCIDENT}	0
1	WARRANT ARREST	10128	{INCIDENT}	5
2	WARRANT ARREST	10736	{INCIDENT}	8
3	WARRANT ARREST	38261	{INCIDENT}	11
4	WARRANT ARREST	52205	{INCIDENT}	14

Representación e implementación de consultas y vistas

Existen distintos grupos de consultas a realizar, en acorde a los atributos de tiempo, distrito y tipo de delito.

1. Periodo de tiempo (periodo, día, y por año)
2. Por zona
3. Por tipo de delito

1. Actividad criminal para periodos de tiempo

En este caso se obtendría el número de incidencias por día. **Count** es una función agregativa que reúne todas las entradas de un mismo tipo.

```
MATCH (n:INCIDENT)-[r:TIME]->(d:DATE) return d,count(n)
```

- N° incidencias agrupadas por *día*

```
MATCH (n:INCIDENT)-->(s:DATE) return s,count(n); // Incidentes por día
```

- N° incidencias agrupadas por *año*

```
MATCH (n:INCIDENT)-->(s:DATE) return s.year,count(n)
```

```
records = s.select_custom('MATCH (n:INCIDENT)-->(s:DATE) return s.year as Year ,count(n) as Ocurrances ORDER BY Year')
```

```
year = list()
ocur = list()
```

```
for dic in iter(records):
    year.append(dic['Year'])
    ocur.append(dic['Ocurrances'])
```

- Actividad criminal por día de la semana. Totales de los incidentes por días de la semana.


```
MATCH (n:INCIDENT)-->(s:DATE) return s.dayofweek,s.year,count(n) order by s.dayofweek,s.year
```

2. Actividad criminal por zona

```
MATCH (n:INCIDENT)-[r:PLACE]->(d:DISTRICT) return distinct d,count(n)
```

3. Actividad criminal por tipo de delito

- N° incidencias agrupadas por *año*

```
MATCH (c:CATEGORY)<--(n:INCIDENT)-->(s:DATE) return s.year,c.name,count(n) order by c.name
```

- N° incidencias agrupadas por *día*

```
MATCH (c:CATEGORY)<--(n:INCIDENT)-->(s:DATE) return s,c.name,count(n) order by c.name
```

Representaciones gráficas. Un ejemplo de representación gráfica se consigue por la frecuencia de actividad criminal por tipo o año, por ejemplo, por año lo haríamos con respecto a la siguiente consulta:

Consulta: `MATCH (n:INCIDENT)-->(s:DATE) return s.year,count(n)`

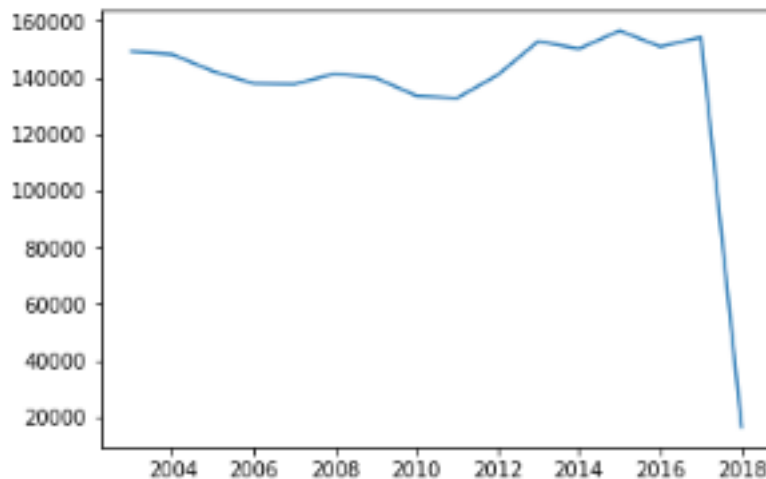
Implementación.

```
records = s.select_custom('MATCH (n:INCIDENT)-->(s:DATE) return s.year as Year ,count(n) as Ocurrances ORDER BY Year')
```

```
year = list()
ocur = list()
```

```
for dic in iter(records):
    year.append(dic['Year'])
    ocur.append(dic['Ocurrances'])
```

```
plt.plot(year, ocur)
```



III. Mongodb

Conexión con la base de datos. El paquete pymongo contiene herramientas para trabajar desde Python con MongoDB. La estructura de los datos en documentos tipo JSON con un esquema dinámico llamado BSON, lo que implica que no existe un esquema predefinido. Los elementos de los datos se denominan documentos y se guardan en colecciones.

pymongo – Mediante su clase principal, MongoClient, da acceso a una instancia de MongoDB, un conjunto de replicasiones o un set de “mongoses”. Dicho objeto es seguro sobre hilo y da acceso a un pool de conexiones. Soporta múltiples enlaces de MongoDB y provee de las configuraciones de cliente necesarias para establecer conexiones persistentes en segundo plano, además de poder recuperar datos en una clase de documento preestablecida.

Instalación y configuración.

```
pip install pymongo
```

Configuración y creación de la sesión. Una vez se realiza la conexión con el cliente de Mongo mediante la URL y el puerto, o el enlace específico de MongoDB, es posible acceder a la base de datos y a continuación, recuperar la información.

Las bases de datos tienen nombres únicos, por lo que la librería permite, mediante el cliente, acceder a estas como **propiedades** de un objeto. Dicha propiedad ofrece la posibilidad de adquirir los datos, que están almacenados en forma de documentos. Para ello vuelve a utilizarse una propiedad, en este caso del **objeto de base de datos**, que retorna una colección de dichos documentos.

```
client = MongoClient("mongodb.vrandkode.net", 27017)
db = client.datascience # Base de datos llamada datascience
collection = db.incidents # Colección de documentos
```

Consultas. Cuando accedemos a los datos de los campos de un documento en **MongoDB** podemos encontrarnos la necesidad de realizar consultas MongoDB like. Es decir, realizar consultas por cadenas similares de texto. Por ejemplo, nombres que empiecen por una letra o letras, palabras que contengan una cierta cadena de caracteres,...

Las consultas **MongoDB** like se resuelven mediante expresiones regulares. Lo que realizaremos mediante la siguiente sintaxis:

```
db.coleccion.find({campo:expresión_regular});
```

Para los patrones de las expresiones regulares MongoDB utiliza “Perl Compatible Regular Expressions” (PCRE).

Caso de uso de sesión: ejemplo de consulta

Mediante la función **aggregation**, se pueden aplicar ciertos comandos sobre la colección que retuvimos en la variable. En este caso, queremos los incidentes cuya categoría sea “robo”, e imprimimos la primera.

```
pipeline = [
    {"$match": {"Category": "ROBBERY"}},
]

aggResult = collection.aggregate(pipeline)
robbery = pd.DataFrame(list(aggResult))
robbery.head(1)
```

	Address	Category	Date	DayOfWeek	Descript	IncidentNum	Location	PdDistrict	PdId
0	300 Block of LEAVENWORTH ST	ROBBERY	02/01/2015	Sunday	ROBBERY, BODILY FORCE	150098210	(37.7841907151119, -122.414406029855)	TENDERLOIN	1500

Representación e implementación de consultas y vistas

Existen distintos grupos de consultas a realizar, en acorde a los atributos de tiempo, distrito y tipo de delito.

1. Periodo de tiempo
2. Por zona
3. Por tipo de delito

1. Actividad criminal para un *periodo de tiempo*

Actividad criminal para un periodo de tiempo En este caso se obtendría el *número de incidencias* por día. **Count** es una función agregativa que reúne todas las entradas de un mismo tipo.

- Incidencias en dia Sunday

```
db.find({DayOfWeek:"Sunday"}).pretty()
```

```
_id: ObjectId("5ac48f27a2eb3a9495192d44")
IncidntNum: 150060275
Category: "NON-CRIMINAL"
Descript: "LOST PROPERTY"
DayOfWeek: "Monday"
Date: "01/19/2015"
Time: "14:00"
PdDistrict: "MISSION"
Resolution: "NONE"
Address: "18TH ST / VALENCIA ST"
X: -122.42158168137
Y: 37.7617007179518
Location: "(37.7617007179518, -122.42158168137)"
PdId: 15006027571000
```

```
_id: ObjectId("5ac48f27a2eb3a9495192d45")
IncidntNum: 150098210
Category: "ROBBERY"
Descript: "ROBBERY, BODILY FORCE"
DayOfWeek: "Sunday"
Date: "02/01/2015"
Time: "15:45"
PdDistrict: "TENDERLOIN"
Resolution: "NONE"
Address: "300 Block of LEAVENWORTH ST"
X: -122.414406029855
Y: 37.7841907151119
Location: "(37.7841907151119, -122.414406029855)"
PdId: 15009821003074
```

- Número de incidencias en Sunday

```
db.incidents.count({DayOfWeek:"Sunday"})
290936
```

- Incidencias en el dia 02/01/2015

```
db.find({Date:"02/01/2015"}).pretty()
```

```
_id: ObjectId("5ac48f27a2eb3a9495192d45")
IncidentNum: 150098210
Category: "ROBBERY"
Descript: "ROBBERY, BODILY FORCE"
DayOfWeek: "Sunday"
Date: "02/01/2015"
Time: "15:45"
PdDistrict: "TENDERLOIN"
Resolution: "NONE"
Address: "300 Block of LEAVENWORTH ST"
X: -122.414406029855
Y: 37.7841907151119
Location: "(37.7841907151119, -122.414406029855)"
PdId: 15009821003074
```

```
_id: ObjectId("5ac48f27a2eb3a9495192d46")
IncidentNum: 150098210
Category: "ASSAULT"
Descript: "AGGRAVATED ASSAULT WITH BODILY FORCE"
DayOfWeek: "Sunday"
Date: "02/01/2015"
Time: "15:45"
PdDistrict: "TENDERLOIN"
Resolution: "NONE"
Address: "300 Block of LEAVENWORTH ST"
X: -122.414406029855
Y: 37.7841907151119
Location: "(37.7841907151119, -122.414406029855)"
PdId: 15009821004014
```

- Número de incidencias el 02/01/2015

```
db.incidents.count({Date:"02/01/2015"})
466
```

2. Actividad criminal por zona

- Incidencias en la Zona de LEAVENWORTH

```
db.incidents.find({Address:/LEAVENWORTH/}).pretty()
```

- Número de incidencias en LEAVENWORTH

```
db.incidents.count({Address:/LEAVENWORTH/})
28163
```

3. Actividad criminal por tipo de delito. por tipo de delito, por año/día (por tipo de delito)

- Incidencias en el año 2013 para los delitos ROBBERY

```
db.incidents.find({Date:/2013/,Category:"ROBBERY"}).pretty()
```

```
_id: ObjectId("5ac48f2fa2eb3a94951ce496")
IncidentNum: 130117084
Category: "ROBBERY"
Descript: "ROBBERY, ARMED WITH A KNIFE"
DayOfWeek: "Sunday"
Date: "02/10/2013"
Time: "01:46"
PdDistrict: "BAYVIEW"
Resolution: "NONE"
Address: "1600 Block of KIRKWOOD AV"
X: -122.390952930587
Y: 37.7386625599684
Location: "(37.7386625599684, -122.390952930587)"
PdId: 13011708403072
```

```
> _id: ObjectId("5ac48f2fa2eb3a94951ce73c")
IncidentNum: 130790232
Category: "ROBBERY"
Descript: "ROBBERY, BODILY FORCE"
DayOfWeek: "Wednesday"
Date: "09/18/2013"
Time: "23:00"
PdDistrict: "PARK"
Resolution: "NONE"
Address: "2200 Block of GEARY BL"
X: -122.440341074545
Y: 37.7833242481047
Location: "(37.7833242481047, -122.440341074545)"
PdId: 13079023203074
```

Número de incidencias de la búsqueda anterior

```
db.incidents.count({Date:/2013/,Category:"ROBBERY"})
4196
```

Distribución por tipos de delitos. Obtenemos por cada tipo de delito una sección de datos por lo que luego podemos sacar una tabla de distribución.

```
db.incidents.distinct( " Category " )
[ 'ROBBERY',
  'ASSAULT'...
```

```
pipeline = [
    {"$match": {"Category": "ROBBERY"}},
]

aggResult = collection.aggregate(pipeline)
robbery = pd.DataFrame(list(aggResult))
robbery.head()
```

...

```
pipeline = [
    {"$match": {"Category": "ASSAULT"}},
]

aggResult = collection.aggregate(pipeline)
assault = pd.DataFrame(list(aggResult))
assault.head()
```

```
print("{} robberies {:.1%}, {} assaults {:.1%}, {} drugs {:.1%})"
      .format(
        len(robbery), len(robbery)/d,
        len(assault), len(assault)/d,
        len(drug), len(drug)/d))
```

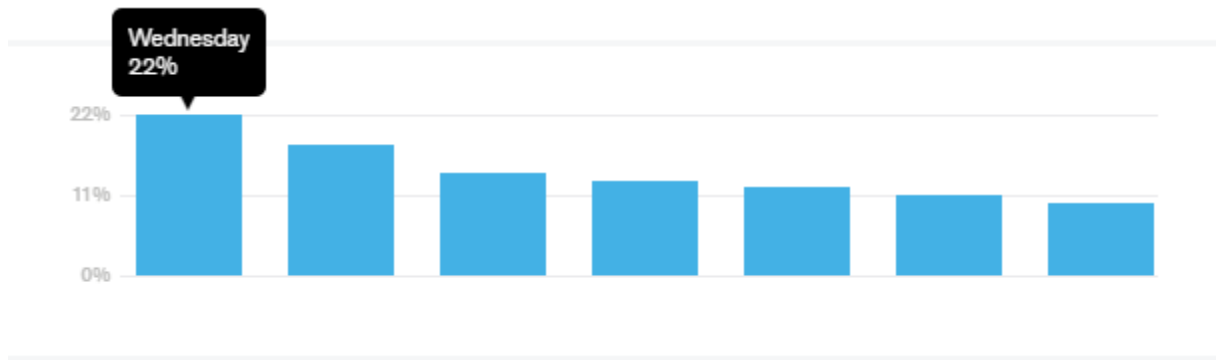
Obtenemos como resultado la distribución por porcentajes:

55242 robberies (2.5%), 191952 assaults (8.8%), 118739 drugs (5.4%)

Representaciones gráficas. Se puede obtener una representación de la distribución de forma agregada por tipo de día de la semana en formato de barras:

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

```
{ "_id" : "Wednesday", "num" : 319726 }
{ "_id" : "Friday", "num" : 333562 }
{ "_id" : "Saturday", "num" : 316451 }
{ "_id" : "Tuesday", "num" : 311399 }
{ "_id" : "Thursday", "num" : 312052 }
{ "_id" : "Monday", "num" : 302862 }
{ "_id" : "Sunday", "num" : 290936 }
```



IV. Processor (aplicación C++)

Con ayuda de los **métodos** de la clase *Incidents*: `getByDistrict` y `getByCategory`, en base al privado genérico `getByAttribute`; se pueden construir consultas de filtrado y de agregación de forma rápida:

```
/* Filtrar y agrupar por distrito */
std::map<std::string, vector<incident> > getByDistrict();

/* Filtrar y agrupar por categoria */
std::map<std::string, vector<incident> > getByCategory();
std::map<std::string, vector<incident> > getByAttribute(const char * attribute);
```

Otros métodos de agrupación de índole genéricas son usados para realizar consultas de tipo *agregado*:

- Por zona void `exportDistrictsCount(Incidents incidents, const char* filename)`
- Por distrito void `exportCategoryCount(Incidents incidents, const char* filename)`

Modelos de aprendizaje

Introducción

El objetivo de este proyecto es predecir la categoría de crímenes que ocurrieron en la ciudad de San Francisco. Desde la ingeniería de características, diferente se realizarán pruebas y técnicas para mejorar la precisión, así como las visualizaciones que nos ayudarán a aprender sobre la ciudad y los crímenes.

El desafío de este proyecto implica intentar predecir la categoría, frecuencia y otros atributos para un crimen comprometido dentro de la ciudad, dadas las diferentes características tales como descripción, tiempo, ubicación, etc. Uso de aprendizaje automático enfoques, uno puede obtener una mejor comprensión de qué crímenes ocurren dónde y cuándo en una ciudad.

El conjunto de datos utilizado en este trabajo es un subconjunto de los datasets normalizados los cuales se han dividido en:

- entrenamiento (train): primeros 1.830,000 ejemplos.
- validación (test): últimos 120,000 ejemplos.

Además de la primera asignación, el entrenamiento inicial se dividió en 80% de entrenamiento y 20% de validación. Esto fue hecho para evitar sobreajuste y falta de generalización. El conjunto de prueba original fue utilizado al final como un análisis final.

Modelos. El modelo de predicción puede ser supervisado y/o no supervisado si consideramos utilizar elementos existentes y un tercero para confirmar la validación del modelo. Hemos elegido tres modelos diferentes, dos supervisados y uno no supervisado, con el fin de encontrar o aproximarnos a un ajuste que nos pudiera permitir realizar predicciones correctas u obtener una base para continuar desarrollando un modelo final y alcanzar el objetivo.

- ▶ El aprendizaje **supervisado** es la tarea de aprendizaje automático de inferir una función de datos de entrenamiento supervisados. Para ello hemos tratado dos modelos, **Bayes** y **Random forests** como una agrupación de árboles múltiples de decisión.
- ▶ Mediante aprendizaje **no supervisado** con **Neural Networks**, exploraremos una nueva estrategia para obtener un modelo de predicción

basado en una red de perceptrones sin necesidad de contrastar la información.

I. Bayes

II. Random forests

III. Redes neuronales

Metodología

Inicialmente recuperamos el dataset procesado en los apartados anteriores de formato CSV y subido a un repositorio de ficheros en Google Cloud Platform bajo la url:

<https://storage.googleapis.com/grupospark/incidents.ml.csv>

Los datos se encuentran encapsulados en objetos pandas.

```
names = ['category', 'type', 'dayofweek',
         'date', 'time', 'day', 'year',
         'month', 'hour', 'district',
         'resolution', 'latitude', 'longitude']
incidents = pd.read_csv("incidents.dataset.csv", names=names, sep=';')
```

	category	type	dayofweek	date	time	day	year	month	hour	district	resolution	latitude	longitude
0	NON-CRIMINAL	LOST PROPERTY	Monday	2015-01-19	14:00:00	19	2015	1	14	MISSION	NONE	-122.42158168137	37.761701
1	ROBBERY	ROBBERY, BODILY FORCE	Sunday	2015-02-01	15:45:00	01	2015	2	15	TENDERLOIN	NONE	-122.414406029855	37.784191
2	ASSAULT	AGGRAVATED ASSAULT WITH BODILY FORCE	Sunday	2015-02-01	15:45:00	01	2015	2	15	TENDERLOIN	NONE	-122.414406029855	37.784191
3	SECONDARY CODES	DOMESTIC VIOLENCE	Sunday	2015-02-01	15:45:00	01	2015	2	15	TENDERLOIN	NONE	-122.414406029855	37.784191

Para el análisis de los modelos utilizaremos las librerías **pandas** y **sklearn** en la manipulación de dataframes.

```
from sklearn import preprocessing
```

Normalización de atributos. Las columnas con valores con nombres o etiquetas deberían ser normalizados para permitir una clasificación más discreta tal y como se ha realizado en la clasificación por el modelo determinado. Dado que estamos haciendo uso de las librerías de **sklearn**, éstas nos facilitan herramientas para convertir estos valores y obtener finalmente un dataset listo para ser tratado.

```
from sklearn import preprocessing
normalized_x = preprocessing.normalize(x)
```

También se necesitan que los atributos pasen a ser de tipo numéricos y así poder aplicar modelos estadísticos de manera fácil. A parte se necesitan realizar desarte de columnas para una mejor discretación de los atributos.

```
# seleccionamos las columnas para nuestro ajuste
incidents.drop(incidents.columns[[1,3,4,5,6,7]], axis=1, inplace=True)

# Normalización: discretización de columnas con valores de etiqueta
incidents['district'] = pd.Categorical(incidents.district).cat.codes
incidents['dayofweek'] = pd.Categorical(incidents.district).cat.codes
incidents['resolution'] = pd.Categorical(incidents.resolution).cat.codes
incidents['category'] = pd.Categorical(incidents.category).cat.codes

# eliminamos los NA
incidents.convert_objects(convert_numeric=True).dropna()
```

	category	dayofweek	hour	district	resolution	latitude	longitude
0	35	4	14	32	15	-122.466758	37.729185
1	21	8	21	33	0	-122.412224	37.782073
2	35	2	19	28	15	-122.422253	37.790863
3	21	11	16	26	0	-122.447125	37.721031

Con la librería **sklearn**, mediante algunas clases y funciones específicas para cada tipo de modelo permite realizar clasificaciones y operaciones necesarias.

- ▶ Bayes `from sklearn.naive_bayes import BernoulliNB`
- ▶ Random forests `from sklearn.ensemble import RandomForestClassifier`
- ▶ Neural Networks `from sklearn.neural_network import MLPClassifier`

Una vez cargado y normalizados los datos del dataset seguiremos el siguiente guión de **fases** para cada uno de los modelos, para realizar una evaluación completa y detallada:

1. Clasificación (Bayes / Random Forests / Neural networks)

2. Entrenamiento del modelo (80% vs 20%)
3. Ajuste
4. Predicciones

I. Bayes

Los ingredientes esenciales de los cuales incluyen combinar el teorema de Bayes con una suposición de independencia sobre las características (esta es la parte "ingenua"). Aunque es simple, sigue siendo un método popular para la categorización de texto. Por ejemplo, usando frecuencias de palabras como características, este enfoque puede clasificar con precisión correos electrónicos como correo no deseado, o si un texto en particular fue escrito por un autor específico. De hecho, con un preprocesamiento cuidadoso, el algoritmo a menudo es competitivo con métodos más avanzados, incluyendo máquinas de vectores de soporte.

En Python existen múltiples formas de aplicar machine learning para el reconocimiento de patrones. Una de estas consiste en los clasificadores probabilísticos, entre los cuales encontramos el **teorema de Bayes**.

El modelo Bayes se caracteriza por asumir una gran independencia entre las características que se encuentran en los datos, lo cual ofrece una perspectiva distinta de el problema que se va a resolver: Obtener la categoría del crimen dada su localización en distrito, hora, día de la semana y mes.

Preparación de datos. Para el caso del modelo de Bayes vamos a necesitar un modelo de datos ordenados por fechas, dado que se necesitaran ciertos atributos y campos para la clasificación, lo haremos de la siguiente manera:

```
incidents = pd.read_csv("incidents.dataset.csv", parse_dates=['Date'],
names=names, sep=';')

# Normalización de la columna de tiempos
incidents['Time'] = pd.to_datetime(incidents['Time'],format='%H:%M')
```

Se van a necesitar ciertas funciones auxiliares para añadir un extra de normalización sobre los datos antes de realizar el entrenamiento del modelo:

- Normalización (normalize)

```
def normalize(data): # Normalización de características.
    data = (data - data.mean()) / (data.max() - data.min())
```

```
return data
```

- **Preparación de datos.** Consiste en convertir los datos a formatos adecuados para la discretización con los que se pueda trabajar.

```
def prep_data(data, test):  
    if (test == 0):  
        # Se realiza una codificación de las etiquetas  
        crimen_labels = preprocessing.LabelEncoder()  
        crimen_encode = crimen_labels.fit_transform(data.Category)  
  
        days = pd.get_dummies(data.DayOfWeek)  
        district = pd.get_dummies(data.PdDistrict)  
        month = pd.get_dummies(data.Date.dt.month, prefix="m")  
        hour = data.Time.dt.hour  
        hour = pd.get_dummies(hour)  
  
        # Construimos el array a partir de los datos obtenidos  
        prepared_data = pd.concat([hour, month, days, district], axis=1)  
        prepared_data['X'] = normalize(data.X)  
        prepared_data['Y'] = normalize(data.Y)  
  
        if (test == 0):  
            prepared_data['crime'] = crimen_encode  
  
    return prepared_data
```

Como resultado, los valores se discretan y se normalizan entre valores máximos y mínimos definidos (normalize), las columnas con valores de etiquetas se codifican y los valores de geolocalización se acotan.

2. Entrenamiento y validación. A partir de ahora podemos llevar a cabo la fase de preprocesamiento del modelo para realizar entrenamiento y tests:

```
train_proc = prep_data(train, 0)  
test_proc = prep_data(test, 1)
```

3. Ajuste.

Con el modelo de Bernoulli, que equivale al modelo binario de independencia, realizamos nuestro modelo a partir del conjunto de entrenamiento ya preparado.

```
model = BernoulliNB()  
model.fit(train_proc[features], train_proc['crime'])
```

4. Predicciones

Tras esto, con el mismo modelo intentamos predecir los casos de test.

```
predicted = model.predict_proba(test_proc[features])
```

Resultados y análisis

Por último, exponemos el resultado de nuestro proceso. Dado que estamos en el repositorio, comentado el código para almacenar este en un CSV. En su lugar convertimos este a un listado de diccionarios sobre el que podemos iterar para obtener los tres primeros casos de test.

```
# Write results
crimen_labels = preprocessing.LabelEncoder()
crimen = crimen_labels.fit_transform(train.Category)
result = pd.DataFrame(predicted, columns=crimen_labels.classes_)
#result.to_csv('results.csv', index=True, index_label='Id')

result_dict = result.to_dict(orient='records')

# Primer caso de test, para visualizar la estructura
pprint.pprint(result_dict[0])
print('\n')
# Iteramos sobre los tres primeros tests, para visualizar cuál es el valor máximo de cada resultado
for i, row in enumerate(result_dict):
    maximo = max(row, key=lambda key: row[key])
    print('Caso', i, ': Categoría-> ', maximo, ' | Resultado-> ', row[maximo])
    if i == 2:
        break;
```

```
{'ARSON': 0.034332225834823565,
'ASSAULT': 0.047551791152962736,
'BAD CHECKS': 0.0028829952318701405,
'BRIBERY': 0.013526986958289225,
'BURGLARY': 0.02731243879623048,
'DISORDERLY CONDUCT': 0.004351904024434736,
'DRIVING UNDER THE INFLUENCE': 0.005169667320699265,
'DRUG/NARCOTIC': 0.0993090885750867,
'DRUNKENNESS': 0.0019527956268837066,
'EMBEZZLEMENT': 0.012016693387333568,
'EXTORTION': 0.0017078669536744534,
'FAMILY OFFENSES': 0.0036255612759995286,
'FORGERY/COUNTERFEITING': 0.026312768999884172,
'FRAUD': 0.01038061964305988,
'GAMBLING': 0.0070507821226408985,
'KIDNAPPING': 0.012860688585998665,
'LARCENY/THEFT': 0.022770216117140567,
'LIQUOR LAWS': 0.005433796665775091,
'LOITERING': 0.014457488665450862,
'MISSING PERSON': 0.021583838781292233,
'NON-CRIMINAL': 0.028694929727619296,
'OTHER OFFENSES': 0.12787248134197488,
'PORNOGRAPHY/OBSCENE MAT': 4.1608133663723245e-07,
'PROSTITUTION': 5.20058065245541e-05,
'ROBBERY': 0.012147180452482009,
'RUNAWAY': 0.004559475780457875,
'SECONDARY CODES': 0.02734717537657905,
'SEX OFFENSES, FORCIBLE': 0.020067755958067337,
'SEX OFFENSES, NON FORCIBLE': 0.0018692613871580982,
'STOLEN PROPERTY': 0.005564584733291485,
'SUICIDE': 0.005869059564852941,
```

```
'SUSPICIOUS OCC': 0.011698214336841362,  
'TRESPASS': 0.004828874844694987,  
'VANDALISM': 0.018721245613633475,  
'VEHICLE THEFT': 0.011754759272107455,  
'WARRANTS': 0.3044903283572655,  
'WEAPON LAWS': 0.03987203664558263}
```

```
Caso 0 : Categoría->  WARRANTS | Resultado->  0.3044903283572655  
Caso 1 : Categoría->  WARRANTS | Resultado->  0.17823174563083816  
Caso 2 : Categoría->  LARCENY/THEFT | Resultado->  0.19658437070412  
366
```

Random forests

```
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',  
                        max_depth=11, max_features='log2', max_leaf_nodes=None,  
                        min_impurity_decrease=0.0, min_impurity_split=None,  
                        min_samples_leaf=1, min_samples_split=1000,  
                        min_weight_fraction_leaf=0.0, n_estimators=24, n_jobs=1,  
                        oob_score=True, random_state=None, verbose=0, warm_start=False)
```

Los **bosques aleatorios** combinan las predicciones de árboles de decisión múltiple. Al construir un árbol de decisión, el conjunto de datos se divide repetidamente en subárboles, guiados por la mejor combinación de variables. Sin embargo, encontrar la combinación correcta de variables puede ser difícil. Por ejemplo, un árbol de decisión construido en base a una muestra pequeña podría no ser generalizable a futuras muestras grandes. Para superar esto, se podrían construir árboles de decisión múltiples, aleatorizando la combinación y el orden de las variables utilizadas. El resultado agregado de estos bosques de árboles formaría un conjunto, conocido como bosque aleatorio.

1. Clasificación. Creamos un clasificador *Random Forest*, con un máximo de profundidad de `depth=11`.

```
clf = RandomForestClassifier(max_features="log2", max_depth=11, n_estimators=24,  
                           min_samples_split=1000, oob_score=True)
```

2. Entrenamiento y validación. A partir de ahora podemos llevar a cabo la fase de preprocesamiento del modelo para realizar entrenamiento y tests. Para ello necesitamos dividir el dataset en dos partes, dijimos 80% y 20%. Con ayuda de la librería `sklearn` con ayuda de `StandardScaler`, `preprocessing` y `train_test_split` podemos particionar y normalizar los datos antes de realizar los ajustes.

```
from sklearn.preprocessing import StandardScaler  
from sklearn import preprocessing  
from sklearn.model_selection import train_test_split
```

a) El atributo de clasificación para nuestro modelo es el tipo de delito, lo que corresponde con la columna de Categoría. Por lo que antes de realizar la clasificación sobre el dataset debemos desvincularlo por ahora.

```
x = dataset.drop('category',axis=1)
y = dataset['category']
```

Normalizamos la columna x:

```
normalized_x = preprocessing.normalize(x)
```

b) Particionamos 80% y 20% respectivamente:

```
x_training, x_test, y_training, y_test = train_test_split(x, y, test_size=0.2)
```

3. Ajuste. Capacitamos al clasificador para que tome las funciones de clasificador y aprenda cómo se relacionan los datos del dataset para el entrenamiento y (la especie). Obtenemos los valores del ajuste de los datos de **entrenamiento** y **validación**.

```
scaler = StandardScaler()
scaler.fit(x_training)
```

```
clf.score(x_training, y_training)
clf.score(x_test, y_test))
```

Entrenamiento	Validación
0.362146	0.359434

4. Predicciones. Ahora que tenemos un modelo, es hora de usarlo para obtener predicciones. Podemos usar las métricas integradas de SciKit-Learn, como un informe de clasificación y una matriz de confusión para evaluar el rendimiento de nuestro modelo. La breve explicación de cómo interpretar una **matriz de confusión** es: cualquier cosa en la diagonal se clasificó correctamente y cualquier cosa fuera de la diagonal se clasificó incorrectamente.

```
from sklearn.metrics import classification_report, confusion_matrix
predictions = clf.predict(x_test)
```

► Matrix de confusión (confusion_matrix)

```
confusion_matrix(y_test,predictions)
```


- Puntuación de eficacia (`accuracy_score`)

```
accuracy_score(y_test, predictions)
```

Accuracy score	0.3594
----------------	--------

Neural networks

Una red neuronal está formada por capas de pequeños elementos informáticos que procesan datos de una manera que recuerda a las neuronas del cerebro. Una forma de aprendizaje automático, mejora en función de los comentarios, si sus juicios eran correctos. En este caso, los investigadores entrenaron su algoritmo utilizando datos del Departamento de Policía de Los Ángeles (LAPD) en California desde 2014 hasta 2016 en más de 50,000 homicidios relacionados con pandillas y no relacionados con pandillas, asaltos agravados y robos.

1. Clasificación. Ahora es el momento de entrenar a nuestro modelo. SciKit Learn lo hace increíblemente fácil, mediante el uso de objetos **estimadores**. En este caso, importaremos nuestro estimador (el modelo clasificador de perceptrón multicapa) de la biblioteca `neural_network` de SciKit-Learn. A continuación, creamos una instancia del modelo, hay muchos parámetros que puede elegir definir y personalizar aquí, solo definiremos `hidden_layer_sizes`.

- Para este parámetro, se pasa una tupla que consiste en el número de neuronas que se desea en cada capa, donde la *n*-ésima entrada de la tupla representa el número de neuronas en la *n*-ésima capa del modelo MLP.
- Hay muchas maneras de elegir estos números, pero para simplificar, elegiremos 3 capas con el mismo número de neuronas que funciones en nuestro conjunto de datos junto con 100 iteraciones máximas, como ejemplo.

```
clf = MLPClassifier(hidden_layer_sizes=(15,15,15),  
                    max_iter=500,verbose=10, solver="sgd")
```

2. Entrenamiento y validación. A partir de ahora podemos llevar a cabo la fase de preprocesamiento del modelo para realizar entrenamiento y tests. Para ello necesitamos dividir el dataset en dos partes, dijimos 80% y 20%. Con ayuda de la librería `sklearn` con ayuda de `StandardScaler`, `preprocessing` y `train_test_split` podemos particionar y normalizar los datos antes de realizar los ajustes.

```
from sklearn.preprocessing import StandardScaler
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
```

a) El atributo de clasificación para nuestro modelo es el tipo de delito, lo que corresponde con la columna de Categoría. Por lo que antes de realizar la clasificación sobre el dataset debemos desvincularlo por ahora.

```
x = dataset.drop('category',axis=1)
y = dataset['category']
```

Normalizamos la columna x:

```
normalized_x = preprocessing.normalize(x)
```

b) Particionamos 80% y 20% respectivamente:

```
x_training, x_test, y_training, y_test = train_test_split(x, y, test_size=0.2)
```

3. Ajuste. Capacitamos al clasificador para que tome las funciones de clasificador y aprenda cómo se relacionan los datos del dataset para el entrenamiento y (la especie). Obtenemos los valores del ajuste de los datos de **entrenamiento** y **validación**.

```
scaler = StandardScaler()
scaler.fit(x_training)
```

```
clf.score(x_training, y_training)
clf.score(x_test, y_test))
```

Entrenamiento	Validación
0.341236	0.341874

4. Predicciones. Ahora que tenemos un modelo, es hora de usarlo para obtener predicciones. Podemos usar las métricas integradas de SciKit-Learn, como un informe de clasificación y una matriz de confusión para evaluar el rendimiento de nuestro modelo. La breve explicación de cómo interpretar una **matriz de confusión** es: cualquier cosa en la diagonal se clasificó correctamente y cualquier cosa fuera de la diagonal se clasificó incorrectamente.

```
from sklearn.metrics import classification_report, confusion_matrix
predictions = clf.predict(x_test)
```

- ▶ Matrix de confusión (`confusion_matrix`)

```
confusion_matrix(y_test, predictions)
```

- ▶ Puntuación de eficacia (`accuracy_score`)

```
accuracy_score(y_test, predictions)
```

Resultado

La desventaja de utilizar un modelo Perceptron multicapa es lo difícil que es interpretar el modelo en sí. Los pesos y los sesgos no serán fácilmente interpretables en relación con qué características son importantes para el modelo en sí.

Sin embargo, si desea extraer los pesos y sesgos MLP después de entrenar su modelo, use sus atributos públicos `coefs` e `intercepts`.

- ▶ **`coefs_`** es una lista de matrices de peso, donde la matriz de ponderación en el índice `i` representa los pesos entre la capa `i` y la capa `i + 1`.
- ▶ **`intercepts_`** es una lista de vectores de sesgo, donde el vector en el índice `i` representa los valores de sesgo agregados a la capa `i + 1`.

Bibliografía

- S. T. Ang, W. Wang, and S. Chyou, “San Francisco Crime Classification,” University of California San Diego, 2015
- Python Cassandra Driver. 3.13.0 (2017). Datastax.
<https://datastax.github.io/python-driver/>
- PyMongo. 3.6.1 (2018). MongoDB.
<https://api.mongodb.com/python/current/index.html>
- Neo4j Bolt Driver for Python.1.6 (2018). Neo4J.
<https://neo4j.com/docs/api/python-driver/current/>
- Graph-database. (2018). Neo4J <https://neo4j.com/developer/graph-database/>

- An overview of Neo4j Internals. (2012). Tobias Lindaaker.
<https://www.slideshare.net/thobe/an-overview-of-neo4j-internals>
- Data modeling introduction. (2018). MongoDB.
<https://docs.mongodb.com/manual/core/data-modeling-introduction/>
- Introducción a la base de datos NoSQL MongoDB. (2017). Pico.dev.
<https://picodotdev.github.io/blog-bitix/2017/05/introduccion-a-la-base-de-datos-nosql-mongodb/>
- Mongo Architecture Guide (PDF). (2017). MongoDB.
<https://resources.mongodb.com/mongodb-architects/mongodb-architecture-guide>
- Data Science for Business. Provost & Fawcett, O'Reilly.