Escuela Técnica Superior de Ingeniería Agronómica y del

Medio Natural (ETSIAMN)

Trabajo Fin de Carrera

Licenciatura en Biotecnología

Desarrollo de una aplicación interactiva para el análisis gráfico de datos biológicos

Autor: Carlos D. Martínez Hinarejos

Director: José Miguel Blanca Postigo

Tutor: Joaquín Cañizares Sales

Septiembre 2012

Desarrollo de una aplicación interactiva para el análisis gráfico de datos biológicos

Autor: Carlos D. Martínez Hinarejos

Director: José Miguel Blanca Postigo

Tutor: Joaquín Cañizares Sales

Septiembre 2012

Índice general

A mis padres, por el tiempo ausente

A Ana, por darme la felicidad

Agradecimientos

Hace siete años me embarqué en la aventura de estudiar una nueva carrera, supongo que añorando mis años de Ingeniería Informática y con el objetivo de conseguir formación que me permitieran encauzar una nueva línea de investigación. La verdad es que el periplo ha sido fructfero no sólo a nivel académico, sino también a nivel personal, y muchas personas han tenido un papel relevante en ello.

En primer lugar, por su cercanía temporal y material a este trabajo final, gracias a José Blanca por ofrecerme la idea y dirigirme el trabajo y a Joaquín Cañizares por tutorizarlo. También a Peio Ziarsolo por participar activamente en las muchas charlas que hemos tenido (varias de ellas ajenas a este trabajo como tal).

Luego, por haber sido mis acompañantes directos en estos años, a mis compañeras y compañeros de clase. Un recuerdo muy especial va para Ana Estellés por ayudarme en esos primeros días de llegada tras mi estancia británica, y luego para toda la gente que me fue acompañando y que aún a veces me convocan a vernos: David, Alicia, Eva, Esther, Leo, Ximo, Paula, Cris, Leire, Lidia, Almu, María, Patri, ... y muchos más nombres que sería excesivo citar. También a mis profesoras y profesores, que conociendo mi si-

tuación de trabajador y estudiante me facilitaron mucho la asistencia a clases y laboratorio, en especial a Miguel Ángel por sus clases de química orgánica.

A nivel extra-biotecnológico, a mis compañeros y compañeras de docencia e investigación, en especial al PRHLT, que ha tenido que sufrir mis ausencias durante esta temporada, y a mis alumnos y alumnas por su comprensión. En lo personal, a mis padres y mi familia, que han tenido que tenido que mirar con perplejidad cómo me he vuelto a liar la vida con estas cosas; a la pandilla de Imágenes, por todas las cenas, juergas, estrenos de cine y su incondicional amistad, y en especial a Jorge por llevar ya tantos años ahí. A mis amigos y amigas del pueblo, a pesar de la distancia.

Y ante todo, y sobre todo, a Ana, por haber aparecido en mi vida de esa forma tan extraordinaria y hacer que cada día merezca más la pena.

Capítulo 1

Introducción y objetivos

En los últimos años, la obtención de datos biológicos, y más específicamente de datos genéticos, ha incrementado notablemente su capacidad. Esto ha sido especialmente drástico en las tecnologías de secuenciación, donde se ha pasado en apenas una década a tecnologías de escaso rendimiento, como Sanger [Sanger and Coulson, 1975], a otras de altísimo rendimiento [Hall, 2007]. Esto ha desplazado el cuello de botella del proceso de interpretación de datos biológicos: si antes lo costoso solía ser la adquisición de los datos (por ejemplo, por secuenciación), hoy día lo habitual es que el mayor esfuerzo se centre en el análisis de la cantidad masiva de datos de la que se dispone.

1.1. Análisis de datos

Los datos en bruto obtenidos del análisis biológico se suelen reflejar en series de datos de gran dimensión (series de vectores). Así, de cada individuo de la muestra se obtienen una serie de características (nivel de expresión de un

cierto conjunto de genes, alelos presentes, coordenadas geográficas en las que habita, relaciones filogenéticas, etc.), las cuales se pueden codificar de forma numérica, lo que facilita la representación de cada individuo en un espacio d-dimensional (siendo d el número de características escogidas). También es posible mantener datos de categorías, representables igualmente en espacios vectoriales con una asociación adecuada entre categoría y coordenada.

El tratamiento automático de estas codificaciones no es problemático. Sin embargo, la representación d-dimensional de los datos resulta inviable a nivel gráfico para $d > 3$, e incluso difícil para $d = 3$. Esta representación gráfica es necesaria para la supervisión de los datos por parte de expertos humanos. La intervención humana es necesaria en gran parte de las ocasiones debido a que los tratamientos automáticos pueden llegar a conclusiones geométricamente correctas pero sin coherencia biológica. Es la intervención del experto humano la que garantiza que se escogen soluciones apropiadas a la biología de los datos analizados. Una apropiada representación gráfica de los resultados facilita enormemente la interpretación de los resultados por parte del experto humano, de ahí que sea muy importante conseguir esa representación.

Para ello es imprescindible realizar una reducción de la dimensionalidad de los datos, de forma que sólo se consideren aquellas características más relevantes para el conjunto de datos en estudio (es decir, aquellas con mayor variabilidad). Las técnicas de análisis multivariate, como por ejemplo el análisis de componentes principales (*Principal Component Analysis, PCA*) [Abdi and Williams, 2010] permite esta obtención de características relevantes. Aun así, es frecuente que el número de dimensiones siga siendo superior a 2, y en ese caso es necesario la representación de las distintas ca-

racterísticas una frente a otra (a las que llamaremos *vistas*), a fin de poder identificar las distintas relaciones.

A modo de ejemplo, de una cierta población podrían obtenerse datos de expresión génica de varios genes de interés, datos morfológicos (tamaños, masas) y datos geográficos (coordenadas en las que se obtuvo la muestra). Cada uno de los datos individuales obtenidos puede codificarse numéricamente o con una categoría (representable por ejemplo con un color particular). A la hora de representarse gráficamente, la vista se constituye eligiendo el par de datos a representar uno frente a otro y un posible dato categórico. Por ejemplo, una vista podría representar la expresión de un gen A frente a un gen B para diversas especies (marcada cada una por un color distinto) de una familia, otra vista la longitud del fruto frente a su peso para distintos climas (cada uno codificado en un color distinto) y otra vista la posición geográfica en la que se obtuvo cada muestra.

Con el *software* actual de representación de datos, lo habitual es que se haga la selección de conjuntos individuos con ciertas características interesantes (desde el punto de vista biológico) en una vista, para luego examinar cómo se reflejan estas relaciones en otras vistas. Si las relaciones encontradas en las distintas vistas no satisfacen las restricciones biológicas es necesario volver a seleccionar en una de las vistas y repetir la verificación. Este proceso, al que podemos llamar secuencial o *batch*, implica un coste temporal muy elevado para el análisis, que se reduciría ostensiblemente si se puede realizar con todas las vistas a la vez y permitiendo que las selecciones en cada vista se reflejen de forma instantánea en otras vistas. A esta forma de trabajo la llamaremos, por contraposición con la anterior, *interactiva*.

1.2. Representación interactiva

La representación interactiva de datos puede llevar a un incremento de productividad en el análisis de datos biológicos. Como se ha explicado en la Sección 1.1, la aproximación *batch* requiere volver a realizar selecciones en una vista sin saber el efecto que tiene dicha selección en otras vistas. Como el efecto en otras vistas es fundamental de cara a confirmar o no la relevancia biológica de la selección realizada, es imposible obviar este proceso, pero sí que es posible hacerlo de forma más eficiente.

La representación interactiva permitiría al experto humano, como su propio nombre indica, *interactuar* con las distintas vistas de datos y ver, a medida que hace selecciones en una de las vistas, el efecto en las otras vistas. Como no debe haber restricciones sobre la vista que se pueda realizar la selección, la inclusión o exclusión de datos en la selección puede hacerse sobre aquella vista que sea más conveniente, visualizando de forma casi inmediata el sentido biológico en el resto de vistas. Esto agiliza drásticamente el proceso de selección de conjuntos de datos con significado biológico y, de paso, permite encontrar de forma visual relaciones que pueden quedar ocultas en un análisis *batch* debido a su carácter secuencial.

1.3. Objetivos del trabajo

Con los argumentos expuestos a lo largo de este capítulo, el objetivo general del trabajo será *implementar una herramienta de representación gráfica de datos con capacidades interactivas para ser empleada en el análisis de datos biológicos.*

Para cubrir dicho objetivo, será necesario cubrir una serie de objetivos secundarios, que serían los siguientes:

1. Elegir un lenguaje de programación apropiado para la implementación.
2. Estudiar y escoger las bibliotecas de representación gráfica más apropiadas para la aplicación.
3. Estudiar y escoger las bibliotecas de representación interna de datos.
4. Implementar y probar la herramienta inicial como prueba de concepto.
5. Dejar a disposición pública la herramienta inicial para prueba y crítica por parte de la potencial comunidad usuaria.

Capítulo 2

Materiales y métodos

En este capítulo se describen las diversas alternativas estudiadas para la implementación de la aplicación interactiva objeto de este trabajo. Se describen las diversas alternativas a nivel de lenguaje de programación (Sección 2.1) y bibliotecas disponibles para la implementación de las funcionalidades de la aplicación (Secciones 2.2 y 2.3). Tras la toma de estas decisiones, se describe la arquitectura de la aplicación (Sección 2.4).

2.1. Lenguaje de programación: Python

A la hora de realizar esta aplicación una primera elección a tomar es la del lenguaje de programación en el cual se implementará. Las diversas alternativas que se tuvieron en cuenta fueron las siguientes:

- *C/C++*: C es una de los lenguajes más clásicos y tradicionales de programación; fue diseñado fundamentalmente para el desarrollo de sistemas operativos, pero su aplicabilidad (así como la de su extensión

C++, con capacidades para orientación a objetos) se ha extendido a todos los campos de aplicación de un *software*.

Ventajas Muy eficiente, muy orientado a la algorítmica, muy bien conocido por el desarrollador del trabajo.

Inconvenientes Limitado en el aspecto gráfico e interactivo.

- *Java*: Java es un lenguaje derivado de C++ que apuesta por una orientación a objetos más fuerte (no existe el concepto de *programa* como tal, y todo son clases); se desarrolló como vehículo multiplataforma de aplicaciones en Internet, pero se ha convertido en un lenguaje de propósito general de amplia extensión.

  Ventajas Excelentemente documentado (en general), multiplataforma, abundancia de bibliotecas disponibles, buenas capacidades de interfaz gráfica.

  Inconvenientes Poco flexible por su obligada arquitectura orientada a objetos, las actualizaciones de versión pueden ser problemáticas.

- *JavaScript*: JavaScript es un lenguaje de *scripting* puramente orientado a Internet; es un lenguaje interpretado que funciona en una arquitectura cliente/servidor y que permite distribuir una aplicación en esos dos términos.

  Ventajas Capacidad inmediata de hacer aplicaciones vía web, con la consecuente facilidad de instalación y utilización.

  Inconvenientes Lenguaje muy complicado en sintaxis y semántica.

- *Python*: Python es un lenguaje interpretado multiplataforma; está especialmente capacitado para el tratamiento de listas de datos; permite aproximaciones tanto funcional como orientada a objetos y es muy utilizado para el prototipado rápido por la facilidad de manejo de datos que ofrece.

  Ventajas Rapidez de implementación de prototipos básicos, abundancia de bibliotecas disponibles, facilidad para crear interfaz gráfica, flexibilidad en el estilo de programación, lenguaje habitual de aplicaciones bioinformáticas.

  Inconvenientes Los cambios de versiones suelen ser muy problemáticos, las compatibilidades entre distintas versiones de bibliotecas no están bien garantizadas en todos los casos.

A la vista de todas estas alternativas y de las bibliotecas de representación gráfica disponibles, una primera opción considerada fue combinar C con la biblioteca *cairo* [Cairo, 2012]. Sin embargo, esta opción fue descartada debido a que implicaba realizar todas las capacidades gráficas e interactivas desde cero, lo cual resultaba demasiado costoso de implementar. Además, uno de los objetivos es que la implementación de la aplicación sea fácilmente modificable y mantenible por bioinformáticos, algo que no es asumible en una implementación en C.

La falta de flexibilidad de Java y la dificultad de JavaScript hicieron que se descartaran como alternativas. Python presenta también varias desventajas, en especial la compatibilidad entre versiones de bibliotecas, pero el resto de ventajas y, en particular, su uso habitual en la comunidad bioinformática,

fueron los factores decisivos para tomarlo como lenguaje de la aplicación.

2.2. Biblioteca de representación gráfica interactiva: *chaco*

Una vez escogido Python como lenguaje de programación, la siguiente decisión era escoger la biblioteca que permitiera implementar la representación gráfica de los datos así como la interactividad (básicamente, capacidades de selección de conjuntos de datos).

Una de las bibliotecas más populares para la representación gráfica en Python es *matplotlib* [Hunter, 2007]. *matplotlib* es una biblioteca que capacita a las aplicaciones Python para la representación de gráficos bidimensionales de alta calidad. Proporciona también una gran cantidad de módulos basado en el mismo, así como paquetes para la realización de interfaces gráficas de usuario (GUI).

Sin embargo, aunque *matplotlib* proporciona ciertas capacidades interactivas, estas son únicamente alcanzables a nivel gráfico, es decir, a la representación en píxeles de los datos internos. Esto dificulta ampliamente la programación de la interacción, ya que requeriría mantener una doble representación de los datos (dato interno y dato gráfico) para poder llevar las selecciones gráficas a la representación interna. Además, cualquier cambio en el gráfico (resolución, relación de aspecto,...) implicaría tener que recalcular esa asociación interna, con una merma de eficiencia y una complicación del código evidente.

La otra alternativa considerada fue la biblioteca *chaco* [Enthought, 2008].

chaco es una biblioteca desarrollada por Enthought [Enthought, 2012] dentro del marco de la Enthought Tool Suite (ETS) y con el objetivo de permitir el desarrollo de aplicaciones con representaciones gráficas e interactivas. *chaco* depende a su vez de *traits*, biblioteca de ETS que proporciona capacidades de tipado y de GUI a las aplicaciones Python. Enthought es una compañía muy involucrada en la comunidad Python, lo que hace presuponer que sus bibliotecas son o serán en un futuro cercano de uso masivo.

La ventaja principal de *chaco* es que la implementación y el manejo de la interactividad es relativamente sencillo, además de trabajar directamente sobre el conjunto de datos internos, de manera independiente a la representación gráfica. Su principal problema es que la documentación no es tan abundante ni bien organizada como la de *matplotlib*, pero en principio la facilidad de programar la interacción la hicieron la opción escogida. El desarrollo posterior mostró otros problemas de *chaco* (por ejemplo, el pésimo mantenimiento de la compatibilidad entre distintas versiones de *chaco* y otras bibliotecas de las que depende), pero a priori se escogió como la mejor alternativa para implementar la parte gráfica e interactiva de la aplicación.

2.3. Biblioteca de representación de datos: *pandas*

Otro elemento a tener en cuenta dentro de la implementación de la aplicación era la representación de los datos. En este caso, el problema fundamental está en la capacidad de manejo de metadatos, en general etiquetas de los datos. Por tanto, es deseable utilizar una biblioteca que permita el

adecuado manejo de metadatos (nombres de filas y columnas de tablas de datos), así como las operaciones apropiadas de selección, lectura y escritura de ficheros, etc.

Haciendo una exploración de las bibliotecas disponibles, se encontraron dos bibliotecas de Python que parecían cumplir con estos requisitos: *larry* y *pandas*.

larry [Python-Software-Foundation, 2012] es una clase Python que extiende los *arrays* de Numpy asociándoles etiquetas. De hecho, los *arrays* Numpy son la base de *larry*, que viene a ser una capa intermedia de manejo de esos *arrays* a los que puede asociar etiquetas. Esto por un lado permite que las tablas de datos tengan más de dos dimensiones, pero por otro lado limita los contenidos de las tablas a contenidos homogéneos (del mismo tipo de datos). Debido a que los datos que queremos analizar y representar pueden tener una naturaleza heterogénea (numérica, cadenas, etc.), la opción de *larry* acabó por ser descartada.

En cambio, *pandas* [Lambda Foundry, 2012] es una biblioteca que, aunque sólo ofrece la capacidad de manejar tablas (*arrays*) bidimensionales, se ajusta más a nuestro propósito. *pandas* define sus propias estructuras de datos, llamadas *Series* y *DataFrame*; la primera de ellas representa estructuras lineales, mientras que la segunda (que puede definirse a partir de combinaciones de *Series* entre sí, o de *DataFrame* con *Series*) representa estructuras matriciales (tablas). Ambas presentan la capacidad de etiquetado, y *DataFrame* puede ser heterogénea respecto a los datos. Presentan además capacidades de selección de datos con múltiples operaciones, así como un interfaz de entrada/salida bastante flexible.

Por todo ello, la biblioteca *pandas* fue la escogida a la hora de implementar la aplicación.

2.4. Arquitectura del sistema

A la hora de diseñar la aplicación se debe decidir el conjunto de clases que la integran, la funcionalidad de cada una de ellas y la forma de coordinarse que presentan, es decir, la arquitectura del sistema.

En este trabajo hemos optado por realizar una arquitectura basada en modelar cada tipo de vista en una clase distinta. Además de ello, se requerirá una clase para la interfaz de acceso inicial (que permitirá, además de acceder a las vistas, la carga de datos). En el desarrollo de la aplicación puede aparecer la necesidad de implementar clases auxiliares, las cuales se detallarán (por no ser fundamentales para la arquitectura) en el Capítulo 3.

El objetivo del trabajo es realizar una aplicación como prueba de concepto, con lo cual el tipo de vista se va a limitar inicialmente a dos: diagrama de puntos bidimensional (*scatter*) y diagrama geográfico. Ambos son bastante semejantes, siendo la diferencia los rangos de representación (números reales generales en los *scatter* y longitudes/latitudes en los geográficos) y el fondo de la vista (debe ser un mapa en el caso del geográfico).

Por tanto, la arquitectura del sistema se define por un módulo (fichero `.py`) para las vistas *scatter*, un módulo para las vistas geográficas y un módulo para la interfaz inicial.

En todos estos módulos tendremos las siguientes clases:

- Interfaz: todas las interfaces son herederas de la clase `HasTraits` y

definen una serie de atributos internos; de estos atributos, el que define el aspecto gráfico como tal es un objeto de la clase `View`; a este objeto se le asocian el resto de elementos de la interfaz (entre otros, menúes, desplegables, textos y cuadros gráficos); estos elementos de la interfaz también pueden definirse como atributos y asociarlos al objecto `View` en los métodos correspondientes; también es necesario un objeto que permita el acceso a la clase de estructura de datos.

- Gestor de eventos: toda interfaz que quiera gestionar eventos necesita de la implementación de una clase gestora de eventos, heredera de la clase `Handler`; en esta clase gestora se debe definir un atributo que permita acceder al objeto interfaz correspondiente, además de todos los métodos que gestionen los eventos procedentes de esa interfaz; la asociación evento-gestor se hace al instanciar el objeto `View` de la interfaz y por la definición de objetos de la clase `Action`.

- Estructura de datos: es una clase heredera de `HasTraits` que da la representación interna de los datos necesaria para cada vista; en el caso de la interfaz inicial, guardaría los datos internos en un objeto `DataFrame` de *pandas*.

La comunicación de estas clases entre sí se hace mediante la definición de atributos dentro de la clase interfaz, declarando los atributos necesarios para el acceso a cada una de las interfaces creadas. Así, en la clase de la interfaz inicial se definen tantos objetos como vistas se vayan a crear, y a medida que se crean las vistas se asocian a dichos objetos. De la misma manera, cada vista al crearse inicializa un atributo que es un objeto de la clase interfaz inicial

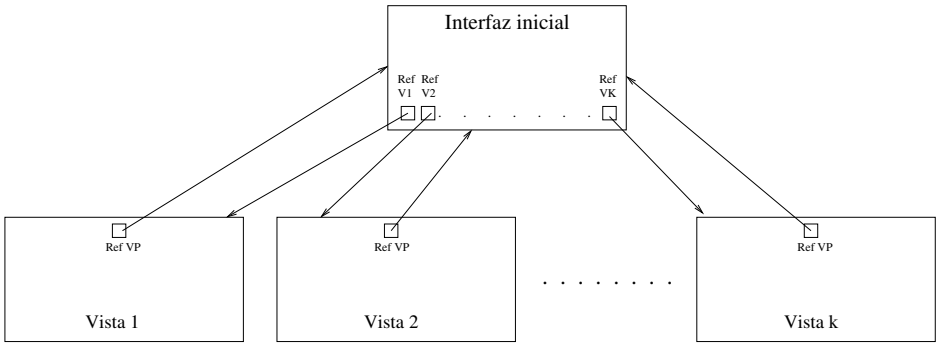

Figura 2.1: Esquema de la arquitectura de la aplicación.

al valor del objeto correspondiente. A la hora de actualizar datos (por nueva selección, por carga de un nuevo fichero de datos, etc.), esta actualización pasa siempre por la clase estructura de datos de la interfaz inicial, la cual se encarga entonces de la actualización de las distintas vistas. El esquema presentado en la Figura 2.1 aclara las relaciones entre los distintos objetos que componen la aplicación.

En el Capítulo 3 se describe la implementación de la arquitectura descrita con las bibliotecas comentadas a lo largo de este capítulo.

Capítulo 3

Desarrollo del trabajo

A lo largo de este capítulo daremos los detalles de implementación de la aplicación. Se empezará en la Sección 3.1 por la definición de la interfaz básica (desarrollo de las clases interfaz y sus gestores de eventos) para luego describir en la Sección 3.2 la representación de datos de cada una de las vistas requeridas; tras ello, en la Sección 3.3 se describirá en profundidad el desarrollo de cada una de las vistas requeridas. Por último, en la Sección 3.4 se describirá el resultado de las pruebas, así como las restricciones de desarrollo (a nivel de versión de bibliotecas, tipos de datos, etc.) que se han encontrado.

3.1. Implementación de la interfaz básica

Antes de ponerse a desarrollar la interfaz básica hay que fijar los objetivos mínimos, a nivel de vistas y funcionalidad, de la aplicación. Para este desarrollo inicial, con potencialidad de ampliaciones futuras, estableceremos lo siguiente:

Manejo de datos externos La relación con datos externos se fija, de forma básica, en la lectura de ficheros de datos en formato CSV, al ser la forma habitual de presentar los datos en la mayor parte de las aplicaciones bioinformáticas. Se describirá con detalle en la Sección 3.2

Vistas a implementar Reduciremos las posibles representaciones a dos tipos básicos: diagramas bidimensionales de puntos (*scatter*) y diagramas geográficos; en realidad, ambos comparten representación primaria (dibujan una serie de puntos en un plano de coordenadas), pero el diagrama geográfico se situará sobre una imagen de una representación geográfica de la zona de interés (aquella definida por los datos a representar); supondremos que puede haber varias vistas *scatter* para un cierto conjunto de datos, pero sólo una vista geográfica para dicho conjunto. Más detalles sobre las vistas se especifican en la Sección 3.3.

Capacidades interactivas Las capacidades interactivas son, básicamente, la selección gráfica de subconjuntos de los datos representados en las vistas; como tal, será necesario proveer de herramientas que permitan la selección individual y múltiple de puntos, así como anular una selección en curso, combinar ambos tipos de selección, o seleccionar todos los datos disponibles. También será necesario definir la forma de diferenciar datos seleccionados de no seleccionados, así como diferenciar los objetos de la selección en curso. El manejo interactivo se describe en detalle en la Sección 3.3.

Aprovechamos también este momento para bautizar la herramienta a implementar, a la cuál llamaremos **HINT**[1].

A la vista de estos objetivos funcionales, la interfaz básica se construye basándose en tres módulos Python, a los que llamaremos `hint.py`, `scatter.py`, y `map.py`. Cada uno de ellos implementará, respectivamente, la interfaz inicial, las vistas *scatter* y la vista geográfica.

A continuación, describiremos en profundidad los elementos que forman parte de la interfaz básica de cada uno de los módulos.

3.1.1. `hint.py`

El módulo `hint.py`, a nivel de interfaz básica, implementa dos clases: la interfaz en sí (clase `HINTWindow`) y su gestor de eventos (clase `handlerHINTWindow`). Cada una de ellas es heredera, despectivamente, de las clases `HasTraits` y `Handler`. Además, `hint.py`, al ser el módulo central de la aplicación, implementa el `main` que activa la ventana de la interfaz inicial (es decir, crea el objeto de `HINTWindow` y lo visualiza en pantalla).

Los atributos principales de la interfaz (clase `HINTWindow`) son:

- `viewHINTWindow`: define el objeto de la clase `View`, es decir, la interfaz como tal.

- `viewScatterWindow`, `viewMapWindow`: definen las referencias a las vistas (de tipo *scatter* las primeras y de tipo geográfico la segunda); de hecho, el primer atributo es un vector de referencias a vistas *scatter*, ya

[1] De *H*erramienta *Int*eractiva, y jugando también con el significado del término en inglés (pista, sugerencia).

que puede haber múltiples vistas *scatter*.

- `activeScatterWindow`: indica qué ventanas de tipo *scatter* están activas; se emplea para saber cuál es la primera posición vacía dentro de las de `viewScatterWindow`, para que cuando se active una nueva vista *scatter* se asocie a dicha posición.

- `dataMenu`: define el menú de datos, con las opciones de cargar datos y salir de la aplicación.

- `actionLoadDataMenu`, `actionExitDataMenu`: definen las acciones asociadas a las opciones de `dataMenu`.

- `viewLoadData`: da acceso a un tipo predefinido de ventana que permite la carga de datos desde el disco.

- `viewMenu`: define el menú de vistas, que da acceso a las vistas *scatter* y geográfica; las opciones de este menú se activan o desactivan en función de que existan datos cargados o no y de que las vistas ya estén activas o no.

- `actionPlotViewMenu`, `actionMapViewMenu`: definen las acciones asociadas a las opciones del `viewMenu`.

- `menubarHINTWindow`: define la barra de menús completa, con los menús `dataMenu` y `viewMenu`.

Los métodos principales de la interfaz (clase `HINTWindow`) son:

- `__init__`: método constructor; además de llamar al constructor de la superclase (`HasTraits`), se encarga de inicializar las referencias a las vistas (se crean las ventanas correspondientes pero no se visualizan), se indica que todas están inactivas y se inactivan las opciones de `viewMenu`, ya que en el estado inicial no hay datos cargados.

- `default_traits_view`: es el método fundamental para la visualización de la interfaz; este método instancia a valores específicos un objeto `View` y lo retorna, haciéndolo visible; en esta instanciación se define, entre otros parámetros, el título de la ventana, su tamaño, su barra de menús y, sobre todo, la clase gestora de eventos asociada.

La interfaz como tal no requiere más métodos, pues es el gestor de eventos de esta clase la que debe procesar las diversas acciones. Para ello, la clase gestora (`handlerHINTWindow`) emplea como atributo `viewHINTWindowHandler`, una referencia a la clase `HINTWindow` que permite el acceso a la interfaz (y con ello su modificación). Los métodos que se definen en el gestor (clase `handlerHINTWindow`) son los siguientes:

- `init`: procesa el evento que se da al crear la ventana; en nuestro caso, lo empleamos para inicializar el valor del atributo `viewHINTWindowHandler` y asociarlo a la interfaz.

- `exitDataMenu`: procesa el evento que se da al tomar la opción `Exit` del menú `dataMenu`; en este caso, sólo finaliza el `main` del módulo.

- `close`: procesa el evento que se da al cerrar la ventana (con el aspa de cierre); en este caso, consiste únicamente en llamar a `exitDataMenu`.

- `scatterViewMenu`: procesa el evento que se da al seleccionar una vista *scatter* en el `viewMenu`; se encarga de encontrar la primera referencia libre en el vector de vistas *scatter*, modifica su estado, crea la ventana e inactiva (en el caso de que se haya alcanzado el máximo de vistas *scatter*) la opción del menú.

- `mapViewMenu`: procesa el evento que se da al seleccionar una vista geográfica en el `viewMenu`; se encarga de crear la ventana y de inactiva la opción del menú.

3.1.2. `scatter.py`

Al igual que pasa en `hint.py`, a nivel de interfaz básica las vistas *scatter* requieren la interfaz en sí (clase `ScatterWindow`) y su gestor de eventos (clase `handlerScatterWindow`), siendo herederas también de `HasTraits` y de `Handler`. Los atributos principales para la clase interfaz (clase `ScatterWindow`) para esta gestión básica son los siguientes:

- `viewScatterWindow`: define el objeto de la clase `View`, es decir, la ventana *scatter* en sí.

- `mainHINTWindow`: es la referencia a la interfaz inicial, es decir, a la definida por una clase `HINTWindow`.

- `windowNum`: identificador de la ventana, es decir, posición que ocupa en el vector de referencias `viewScatterWindow` de la clase `HINTWindow`.

- `actionScatterSelectionShowall`, `actionScatterSelectionReset`, `actionScatterSelectionConfirm`, `actionScatterSelectionNone`,

`actionScatterSelectionIndividual`, `actionScatterSelectionGroup`: acciones asociadas a las opciones del único menú que presenta la clase.

- `menubarScatterWindow`: barra de menús de la ventana; en este caso, el menú no se instancia e incluye en la declaración (pues de esa forma daba problemas), sino que se hará en la creación de la ventana.

Por otro lado, los métodos fundamentales para la gestión básica de la interfaz son los siguientes:

- `__init__`: además de llamar al constructor de la superclase, inactiva todas las opciones del menú (al crear la ventana no tiene datos asociados).

- `default_traits_view`: crea el menú de la ventana (llamando a otro método auxiliar, `createScatterMenu`) y la ventana correspondiente, la cual se retorna; al crear la ventana se define el gestor de eventos correspondiente (clase `handlerScatterWindow`).

- `createScatterMenu`: crea el menú de opciones de la vista; incluye seis opciones en dicho menú:

  1. *None*: sin selección; inactiva cuando no se está en proceso de selección.
  2. *Individual*: selección punto a punto; inactiva si ya se está en selección punto a punto.
  3. *Group*: selección por área; inactiva si ya se está en selección por área.

4. *Confirm selection*: aceptar la selección de datos actual; inactiva cuando no se está en proceso de selección.
5. *Reset selection*: vuelve al estado previo al proceso de selección actual; inactiva cuando no se está en proceso de selección.
6. *Show all data*: seleccionar todos los datos y aceptar dicha selección; esta opción está siempre disponible.

- `reactivateAllMenuOptions`: pone todas las opciones del menú de la vista asociadas al tipo de selección como activas (por tanto, activa *Individual* y *Group*), e inactiva aquellas que requieren estar en proceso de selección (*Confirm selection*, *Reset selection* y *None*).
- `inactivateIndividualMenuOptions`: desactiva la opcion *Individual* del menú y activa el resto.
- `inactivateGroupMenuOptions`: desactiva la opción *Group* y activa el resto.

Con esta serie de métodos queda implementada la funcionalidad principal de la interfaz, pero esto debe complementarse con el gestor de eventos de esta clase. La clase gestora (en este caso, `handlerScatterWindow`) emplea como atributo una referencia a su correspondiente ventana llamada `viewScatterWindowHandler` (del tipo `ScatterWindow`). Los métodos que se implementan en esta clase gestora son:

- `init`: como en `hint.py`, asocia el atributo referencia a la ventana (en este caso `viewScatterWindowHandler`) a la interfaz (ventana) correspondiente.

- `close`: este método gestor tiene un efecto sobre la interfaz inicial, pues al cerrar una ventana *scatter* se debe reactivar la opción de menú de la interfaz inicial que da acceso a los *scatter*, así como marcar como inactiva el índice correspondiente de las vistas *scatter* (atributo `activeScatterWindow` de la ventana principal asociada).

- `scatterShowallSelection`, `scatterConfirmSelection`, `scatterSelectionNone`: a nivel de interfaz llaman al método `reactivateAllMenuOptions` de `viewScatterWindowHandler`.

- `scatterResetSelection`: no tiene efecto a nivel de interfaz.

- `scatterSelectionIndividual`: a nivel de interfaz llama al método `inactivateIndividualMenuOptions` de `viewScatterWindowHandler`.

- `scatterSelectionGroup`: a nivel de interfaz llama al método `inactivateGroupMenuOptions` de `viewScatterWindowHandler`.

Todos estos métodos excepto `init` retornan un valor `True`. De igual forma, los métodos verán enriquecida su funcionalidad cuando se tengan en cuenta tanto la representación de datos como la implementación final de las vistas (capacidades de dibujo de diagramas y de selección gráfica) que se describen en las Secciones 3.2 y 3.3.

3.1.3. `map.py`

Este módulo es prácticamente idéntico a `scatter.py` a nivel de interfaz. La interfaz en sí se implementa en la clase `MapWindow` (heredera de `HasTraits`). Los atributos básicos de la gestión de interfaz son:

- `viewMapWindow`: define el objeto de la clase `View`, es decir, la ventana de vista geográfica.
- `mainHINTWindow`: referencia a la interfaz inicial (a la definida por `HINTWindow`).
- `actionMapSelectionShowall`, `actionMapSelectionReset`, `actionMapSelectionConfirm`, `actionMapSelectionNone`, `actionMapSelectionIndividual`, `actionMapSelectionGroup`: acciones asociadas al único menú que presenta la clase.
- `menubarMapWindow`: barra de menús de la ventana, instanciada en la misma forma que para el menú equivalente de `viewScatterWindow`.

Los métodos implementados para esta clase y su funcionamiento a nivel de interfaz son:

- `__init__`: actúa igual que para `ScatterWindow`.
- `default_traits_view`: actúa de manera semejante al análogo de `ScatterWindow`, sólo que el menú se crea con el método `createMapMenu`.
- `createMapMenu`: crea un menú con las mismas opciones y comportamiento que el creado por `createScatterMenu` de `ScatterWindow`.
- `reactivateAllMenuOptions`, `inactivateIndividualMenuOptions`, `inactivateGroupMenuOptions`: con las mismas funcionalidades que los equivalentes de `ScatterWindow`.

Al igual que para las vistas *scatter*, se debe implementar una clase gestora, a la que llamaremos `handlerMapWindow` y que tendrá como atributo

una referencia a la ventana (objeto de `MapWindow` correspondiente) llamada `viewMapWindowHandler` y como métodos los siguientes:

- `init`: como en `scatter.py`, asocia el atributo referencia a la ventana (en este caso `viewMapWindowHandler`) a la interfaz (ventana) correspondiente.

- `close`: este método gestor tiene un efecto sobre la interfaz inicial, pues al cerrar una ventana geográfica se debe reactivar la opción de menú de la interfaz inicial que da acceso a la vista geográfica.

- `mapShowallSelection`, `mapConfirmSelection`, `mapSelectionNone`: a nivel de interfaz llaman al método `reactivateAllMenuOptions` de `viewMapWindowHandler`.

- `mapResetSelection`: no tiene efecto a nivel de interfaz.

- `mapSelectionIndividual`: a nivel de interfaz llama al método `inactivateIndividualMenuOptions` de `viewMapWindowHandler`.

- `mapSelectionGroup`: a nivel de interfaz llama al método `inactivateGroupMenuOptions` de `viewMapWindowHandler`.

Como en el módulo `scatter.py`, todos estos métodos retornan `True` (excepto `init`) y adquirirán una mayor funcionalidad cuando se le añada a la implementación la representación de datos y las funcionalidades esperadas de las vistas (Secciones 3.2 y 3.3).

3.2. Implementación de la representación interna de datos

Como hemos presentado en el Capítulo 2, la representación de datos se va a basar en la estructura `DataFrame` de la biblioteca *pandas*. Por tanto, se definirá una clase que albergue un atributo `DataFrame` y con una serie de métodos que permita acceder a esos datos almacenados de una forma ordenada. Posteriormente, para cada una de las vistas habrá que definir una representación interna de los datos (en general, son vectores numéricos), con lo que para cada módulo definiremos una clase para almacenar los datos.

Además, uno de los requerimientos de la aplicación es que al activar cualquier vista se le pida al usuario qué datos quiere representar. Si pensamos en un `DataFrame` como un conjunto de filas, donde cada fila representa un individuo, cada columna representa un dato específico del conjunto de individuos. Aprovechando la capacidad de los `DataFrame` de asociar etiquetas a filas y columnas, se debe ofrecer al usuario la selección de estas columnas de datos por las etiquetas correspondientes. Para ello se implementará un módulo auxiliar (`datachoose.py`) y se añadirá a las interfaces de las vistas unas listas desplegables (*droplists*) que permitirán cambiar la selección de las columnas de datos representadas en la vista.

3.2.1. `hint.py`

En este módulo se implementa una nueva clase llamada `SelectedData`; hereda de `HasTraits` y tiene dos atributos:

- `selected`: es el objeto `DataFrame` en el que se almacenarán los datos.

- `indexes`: es un objeto `Series` que permitirá almacenar los índices de cada objeto a nivel global de la aplicación; esto es necesario porque las herramientas de selección utilizan índices relativos que es necesario convertir convenientemente a índices absolutos, para lo cual se emplea este objeto.

Los métodos que se implementan en `SelectedData` son los siguientes:

- `__init__`: simplemente llama al construtor de su superclase.

- `initSelected`: recibe como parámetro un fichero y carga en `selected` los datos de dicho fichero; además, deposita en `indexes` los índices correspondientes; por último, añade a `selected` una columna de nombre `"Selected"`, toda ella de valores `True`, que indicará si la fila está seleccionada o no.

- `isSelected`: recibe un índice relativo y, tras convertirlo a absoluto a través de `indexes`, devuelve si el dato está o no seleccionado (valor de la columna `"Selected"`).

- `setSelected`: recibe un vector con índices relativos y, tras convertirlos a absolutos, modifica el valor de `"Selected"` de cada fila a `True` o `False` según esté o no esté en ese vector.

- `setAll`: pone todos los valores de `"Selected"` a `True`.

Evidentemente, este manejo de los datos implica también cambios en las otras clases del módulo. En `HINTWindow` aparecen un atributo `selected` que

es un objeto de tipo `SelectedData`, un atributo `dataColumns`, que guardará los nombres de las etiquetas seleccionables del conjunto de datos cargado, y un atributo `viewDataChooseWindow` que da acceso a la ventana auxiliar para la selección de las columnas a representar.

En `handlerHINTWindow` se implementa el método `loadDataMenu`, encargado de la gestión de datos. En concreto, `loadDataMenu` genera una ventana predefinida de carga de fichero. Una vez obtenido el nombre del fichero se encarga de inicializar a vacío los datos de las vistas y de cargar (usando `initSelected`) los datos en `selected` de la `HINTWindow`, además de inicializar `dataColumn` a las posibles etiquetas. Tras ello inicializa los datos de las vistas a valores por omisión, se cierran las vistas activas (llamando a los métodos `closeScatterWindow` y `closeMapWindow`) y se activan las opciones del menú de vistas.

Además, se deben modificar los métodos `scatterViewMenu` y `mapViewMenu`, pues antes de crear las ventanas de las vistas correspondientes deben poner los valores seleccionables en `viewDataChooseWindow` (usando `dataColumns`) y crear la ventana de selección de columnas. En el apartado 3.2.2 se especifica la implementación de este tipo de ventana. De igual forma, las columnas de datos seleccionadas se deben pasar a las vistas correspondientes (llamando al método `setPullDownValues` de las clases `ScatterView` y `MapView`).

3.2.2. `datachoose.py`

Este módulo emplea una clase auxiliar llamada `DataChooseWindow`, heredera de `HasTraits`, que únicamente representa tres listas desplegables co-

rrespondientes a las coordenadas X e Y y al color a representar en las vistas. Dentro de esta clase se tienen los siguientes atributos:

- `viewDataChooseWindow`: es la ventana como tal.
- `dataChosenIndexes`: vector que guardará los índices seleccionados en cada desplegable.
- `mainHINTWindow`: referencia a la ventana principal de la aplicación.
- `dataPDX`, `dataPDY`, `dataPDC`: listas que indican los valores a mostrar en los desplegables.
- `dataPullDownX`, `dataPullDownC`, `dataPullDownC`: enumerados que actúan como *wrappers* de los valores de sus respectivos `dataPD`.
- `dataXPDItem`, `dataYPDItem`, `dataCPDItem`: instancias de la clase `Item` que implementan los desplegables como tal.

Respecto a los métodos implementados en esta clase tenemos:

- `__init__`: llama al constructor de la superclase y pone `dataChosenIndexes` a vacío.
- `setValues`: recibe un vector que indica los nombres de las columnas de datos a seleccionar en los desplegables; este método se usa desde los `scatterViewMenu` y `mapViewMenu` de `handlerHINTWindow`, y asigna a los atributos `dataPD` el vector pasado.
- `default_traits_view`: crea los `data-PDItem` asociándolos a los `data-PullDown` correspondientes y crea la ventana incluyendo esos desplegables y un botón de `OK`.

Además, se implementa en este módulo la clase gestora `handlerDataChooseWindow`, que incluye el atributo `viewDataChooseWindowHandler` para acceder a la ventana, el método `init` para asociar ese atributo a la ventana, y el método `close` que extrae los índices de los valores seleccionados en los desplegables y los deposita en `dataChosenIndexes`.

3.2.3. `scatter.py`

La implementación interna de los datos en las vistas *scatter* se ha resuelto definiendo una clase `ScatterData`, heredera de `HasTraits`. Esta clase presenta cinco atributos:

- `size`: cantidad de datos.
- `datax`, `datay`, `datac`: vectores para almacenar los datos de las coordenadas X, Y y el color.
- `globalIndex`: vector que guarda el índice respecto a los datos globales de los datos de la clase; esto es necesario para poder trabajar correctamente con conjuntos seleccionados, pues su índice en los vectores internos no se correspondería con los índices en el `DataFrame` que los guarda en `SelectedData`.

Los métodos que implementa esta clase son:

- `__init__`: sólo llama al constructor de la superclase.
- `initScatterData`: inicializa a vacíos los vectores y pone `size` a 0.

- `addScatterData`: recibe tres vectores asociados a las coordenadas X e Y y al color y los pone sobre los atributos `data` correspondientes, además de añadir a `globalIndex` los valores necesarios.

Además, en la clase `ScatterWindow` se definen dos atributos de la clase `ScatterData`, `selDataPlot` y `allDataPlot`, que corresponden a los datos seleccionados y totales dentro del conjunto de datos disponible. De igual forma, se deben definir los atributos para manejar los desplegables de las columnas de datos a representar, con misión semejante a los definidos en `DataChooseWindow` (`dataPDX`, `dataPDY`, `dataPDC`, `dataPullDownX`, `dataPullDownY`, `dataPullDownC`, `dataXPDItem`, `dataYPDItem`, `dataCPDItem`), además de tres índices enteros (`dataXIndex`, `dataYIndex`, `dataCIndex`) que indicarán qué columna se ha seleccionado en cada desplegable. También es necesario definir un atributo `uiScatterWindow`, de la clase `UI`, para poder hacer el cierre de la vista desde el método `loadDataMenu` de `handlerHINTWindow`.

Los métodos a implementar o modificar en esta clase son:

- `__init__`: pone los índices `data-Index` todos a 0.
- `closeScatterWindow`: llama al método `finish` de `uiScatterWindow`.
- `setPullDownValues`: recibe un vector con los índices que indican a qué valores deben inicializarse los tres desplegables de la vista.
- `default_traits_view`: se añade la creación de los desplegables.
- `updatePullDownChangeX`, `updatePullDownChangeY`, `updatePullDownChangeC`: son métodos que se asocian a una modificación del desplegable correspondiente (mediante la directiva `@on_trait_change`) y que toman

el índice del valor del desplegable y lo depositan en el `data-Index` correspondiente.

En el gestor `handlerScatterWindow` sólo es necesario modificar `init` para asignar el valor de `uiScatterWindow` al valor `info.ui`.

3.2.4. `map.py`

De manera semejante a las vistas *scatter*, la vista geográfica necesita de la implementación de una clase `MapData`, con los mismos atributos y métodos que `ScatterData`. La única diferencia sustancial es que el método `addMapData` controla que los datos estén en un rango de valores correcto para una representación geográfica: $[-180 : 180]$ para el eje X (longitudes) y $[-90 : 90]$ para el eje Y (latitudes); en el caso de que algún dato pasado supere el rango, se satura al valor más próximo de dicho rango.

De la misma forma, en `MapWindow` se implementan atributos semejantes para los datos (`selDataPlot`, `allDataPlot`), el manejo de la ventana (`uiMapWindow`) y el manejo de los desplegables. Las modificaciones en los métodos de `MapWindow` son análogas a las de `ScatterWindow`, así como la ligera modificación del `init` de `handlerMapWindow`.

3.3. Implementación de las vistas

Pasamos finalmente a describir la parte fundamental de la aplicación: la representación gráfica de los datos y su selección. Esto implica usar objetos de las clases de *chaco* que permiten representar datos de forma bidimensional y con color asociado (objetos de la clase `Plot`), así como herramientas

de selección (objetos de las clases `ScatterInspector` y `LassoSelection`). Estos objetos emplearán los datos almacenados en las respectivas clases que representan los datos (clases `ScatterData` y `MapData`).

Además, hay que definir una clase que permita la adecuada representación de los colores de los datos, debido a las particularidades que presentan los objetos `Plot` para asignar los colores a cada punto representado. Esto se conseguirá con una clase `Palette`, implementada en el módulo `palette.py`. También se debe tener en cuenta que toda selección realizada en una de las vistas debe tener su efecto en todas las vistas, lo cual se conseguirá por un tránsito común por la ventana principal y su objeto `selected`. Por último, para la representación geográfica emplearemos la generación de mapas y la traslación geográfica que nos da la clase `Basemap`.

3.3.1. `hint.py`

La modificación principal en `HINTWindow` es la incorporación del método `refreshWindows`. Este método recorre todas las vistas activas y usando sus métodos `setData` hace que los datos internos de la vista sean exclusivamente igual a los seleccionados. Respecto al gestor `handlerHINTWindow`, se añade como última instrucción de los métodos `scatterViewMenu` y `mapViewMenu` una llamada a `refreshWindows`, provocando así la actualización de todas las vistas y, en particular, de la recién creada.

3.3.2. `palette.py`

El módulo `palette.py` implementa únicamente la clase `Palette`, heredera de `HasTraits`. Esta clase tiene el propósito de obtener el objeto

`ColorMapper` apropiado para una secuencia de puntos (y que luego se empleará en los diagramas de puntos correspondientes). Tiene como atributos un objeto `cm` de la clase `ColorMapper` y un vector `cd`. Los métodos que implementa son:

- `__init__`: sólo llama al constructor de la superclase.
- `create_colors`: define una serie de 21 colores en formato RGB, que se sitúan en un vector y se emplean para crear el objeto `cm` mediante el método `from_palette_array` de la clase `ColorMapper`; tras ello, en el vector `cd` se sitúan cadenas vacías y luego en posiciones específicas los nombres de los colores correspondientes, de forma que un nombre de color se asocia a una posición.
- `obtain_colors`: este método recibe una lista de cadenas con nombres de colores y construye un vector con las posiciones de cada color pasado dentro de `cd`; si no encuentra el color, se asigna siempre negro (0); además, debido a la naturaleza de la representación en colores, es necesario añadir en dicho vector dos datos finales de primer (0) y último (255) índice, a fin de garantizar todo el rango de colores de forma correcta; ese vector construído se retorna.

3.3.3. `scatter.py`

Las modificaciones a introducir en la clase `ScatterWindow` son mucho más sustanciales. Por un lado, a nivel de atributos se definen los objetos:

- `allplotdata`: de la clase `ArrayPlotData`; es el que se asocia a la representación gráfica real de los datos.

- `plot`: de la clase `Plot`; es el objeto donde realmente se representan gráficamente los datos.
- `indexSelected`: de la clase `AbstractDataSource`, es el que permite obtener los índices fruto de la selección.
- `scatterInspector`: objeto de la clase `ScatterInspector` que permite la selección individual de datos por pulsaciones de ratón.
- `scatterInspectorOverlay`: objeto de la clase `ScatterInspectorOverlay` que permite definir la marca sobre los datos provisionalmente seleccionados.
- `grpSelOver`: objeto de la clase `LassoOverlay` que permite la selección grupal.
- `indSelAct`, `grpSelAct`: booleanos que indican si está activa la selección individual o grupal, respectivamente.
- `pal`: de la clase `Palette`; define la paleta de colores a utilizar.

Las modificaciones en métodos también son sustanciales. Por un lado, en métodos ya descritos, tenemos las siguientes modificaciones:

- `__init__`: para `allplotdata` se definen cuatro dimensiones de valores (x, y, x de seleccionados, y de seleccionados), y se ponen a vacío; se asocia `allplotdata` a `plot`, y se añaden dos `plot` a dicho objeto (uno para la totalidad de datos y otro para los seleccionados), identificándolos por `scatterplot` y `scatterplotsel`; el primero de ellos usa como marcador un cuadrado, y el segundo un aspa (X); se inician los

booleanos de selección activa a falso y se crean `scatterInspector` y `scatterInspectorOverlay` asociados a `plot`; este último define el marcador de los seleccionados provisionales como una cruz (+); por último, se inicializa `indexSelected` al total de índices de los datos y se crean los colores de la paleta `pal`.

- `default_traits_view`: al crear la vista de la ventana (el objeto `View`), le incorpora un objeto `Item` que es el objeto `plot`, lo que provoca la aparición del espacio de dibujo y selección en la interfaz.

- `updatePullDownChange-`: hay que añadir la llamada a un método `inactivateSelection`, que hace que se pase a modo en que no hay ninguna selección activa (ni individual ni grupal).

Además, aparece una gran cantidad de métodos relacionados con la representación gráfica y la selección. Los siguientes son los utilizados desde `HINTWindow` cuando se ejecuta `refreshWindows`, aunque algunos de ellos también se emplean cuando se hace alguna acción de los menús de la vista *scatter*:

- `setData`: es el método que se llama desde `refreshWindows` (en `HINTWindow`) para hacer la actualización de datos; recibe los objetos `dataScatter`, `selected` y `dataColumns` de `HINTWindow`; activa todas las opciones del menú (para permitir la selección), da valores a los `dataPD-` y `dataPullDown-` y asigna a `allDataPlot` y `selDataPlot` los valores pertinentes de `selected` (la totalidad y los seleccionados, respectivamente, empleando los métodos `obtainAll` y `obtainSelected`); modifica entonces los rangos del `plot` para ajustarlos a los datos (método

`setRanges`); añade luego dos datos extra artificiales fuera de los rangos (necesarios para que la representación de color sea adecuada) y realiza sobre `allplotdata` la asignación de valores a cada una de las dimensiones representadas (x, y, x de seleccionadas, y de seleccionadas, colores); por último, llama al método `createNewPlot` que se encargará de redibujar el diagrama de puntos con los nuevos datos.

- `obtainAll`: recibe `selected` como parámetro; únicamente crea un objeto del tipo `ScatterData`, lo inicializa y le añade la totalidad de valores de las columnas de `selected` indexadas por los valores elegidos en los desplegables (para x, y, color); finalmente, retorna el `ScatterData` construído.

- `obtainSelected`: actúa de manera similar a `obtainAll`, pero sólo añade los datos cuya columna `Selected` está a `True`.

- `setRanges`: calcula los mínimos y máximos de los valores de x e y en `allDataPlot` y luego modifica los rangos de `plot` para que den un 10 % de margen respecto a la escala e incluir adecuadamente los puntos en el gráfico.

- `createNewPlot`: es el encargado de crear el `plot` con la disposición actual de datos; primero crea un nuevo objeto `plot` asociándole de nuevo `allplotdata`; tras ello, dibuja la totalidad de datos (llamada al método `plot`) con color (tipo `cmap_scatter`) y cuadrados, y luego sólo los seleccionados con aspas negras; reinicia entonces los rangos (`setRange`) y los objetos `scatterInspector` y `scatterInspectorOverlay`, y tras ello

restaura `indexSelected` a los índices del `plot` actual; por último, inicia las herramientas de selección si están activas (tras confirmar una selección, el tipo de selección que se estuviera haciendo sigue disponible); para ello, se añaden a las `tools` de `plot` el objeto `scatterInspector` y a las `overlays` de `plot` el objeto `scatterInspectorOverlay`; en el caso de la selección grupal también hay que crear un objeto `LassoSelection` para ponerlo como `active_tool` del `plot` y luego añadir a los `overlays` un `LassoOverlay`; por último, se llama al método de `plot` `request_redraw` que se encarga de redibujar el aspecto del gráfico.

Los métodos listados a continuación son los que se emplean para responder a las acciones de los menús, con lo que serán llamados desde los métodos del gestor `handlerScatterWindow`:

- `selectedPoints`: se encarga de obtener los índices de los puntos seleccionados en el `plot` (por `indexSelected`), los convierte a los índices globales (por el atributo `globalIndex` de `allDataPlot`), manda esos índices a `setSelected` del `selected` de `mainHINTWindow` y llama a `refreshWindows` para actualizar la selección en todas las vistas.

- `resetSelection`: llama a `createNewPlot`, de forma que no ha cambiado el conjunto de seleccionados y redibuja la selección previa.

- `showAll`: llama a `setAll` del `selected` de `mainHINTWindow` y tras ello al `refreshWindows`, de forma que en todas las vistas aparecerán todos los puntos como seleccionados.

- `inactivateSelection`: inactiva el modo de selección actual, es decir, se pone en modo de no selección; para ello pone `active_tool` del

`plot` a `None`, pone los booleanos de selección correspondientes a `False`, si la selección grupal esta activa llama a `addGroupSelectedPoints` y pone `grpSelOver` del `plot` a `None`; por último, hace el `remove` de `scatterInspector` y `scatterInspectorOverlay` y hace el `request_redraw` del `plot`, de manera que los que se habían preseleccionado siguen preseleccionados.

- `addGroupSelectedPoints`: incluye los datos seleccionados grupalmente como seleccionados individualmente; se emplea desde `inactivateSelection`.

- `activateIndividualSelection`: hace los `append` de `scatterInspector` y `scatterInspectorOverlay` sobre `plot`, y pone el booleano `indSelAct` a `True`.

- `activateGroupSelection`: crea un objeto de la clase `LassoSelection` para ponerlo como `active_tool` del `plot`, hace los `append` de `scatterInspector`, `scatterInspectorOverlay` y `grpSelOver` sobre el `plot`, y pone el booleano `grpSelAct` a `True`.

Por último, queda modificar los métodos de la clase gestora `handlerScatterWindow`. En concreto, se tienen las siguientes modificaciones:

- `scatterShowallSelection`: llama a `showAll`.

- `scatterResetSelection`: llama a `resetSelection`.

- `scatterConfirmSelection`: llama a `inactivateSelection` y a `selectedPoints`.

- `scatterSelectionNone`: llama a `inactivateSelection`.
- `scatterSelectionIndividual`: llama a `inactivateSelection` y a `activateIndividualSelection`.
- `scatterSelectionGroup`: llama a `inactivateSelection` y a `activateGroupSelection`.

3.3.4. `map.py`

En la representación geográfica aparecen atributos y métodos en `MapWindow` semejantes a los de `ScatterWindow`. Además, aparecen dos atributos enteros `plotSizeX` y `plotSizeY` que dan el tamaño de la ventana (empleados para generar la imagen geográfica a un tamaño apropiado al de la ventana en que se insertan) y un atributo `bmap` de la clase `Basemap` que se emplea en la generación del mapa geográfico. Además se define una cadena `imageMapFileName` que da el nombre del fichero PNG sobre el que se generará el mapa ajustado a los datos a presentar.

Las modificaciones realizadas en los métodos ya implementados serían:

- `__init__`: se genera un mapa mundial con el contorno continental en gris y los rellenos a blanco, y se salva en el fichero definido por `imageMapFileName`; a la hora de definir el `plot` se carga el mapa en un objeto `ImageData` y se asocia al `plot`, dando valores a `plotSize-` en función de las dimensiones de la imagen y poniendo los atributos `width` y `height` de `plot` a los valores adecuados a ese tamaño; tras ello, se introducen las mismas modificaciones que en el `__init__` de `ScatterWindow`.

- `default_traits_view`: se modifica de forma análoga al de `ScatterWindow`.

- `updatePullDownChange-`: se modifican de forma análoga a los de `ScatterWindow`.

Aparecen también los mismos métodos implementados para `ScatterWindow`, excepto `setRanges` (pues los rangos los fija en este caso el tamaño de la imagen, que ya se ajusta a los datos a representar). Varios de los métodos presentan ciertas diferencias con los implementados en `ScatterWindow`, y se detallan a continuación:

- `setData`: las diferencias principales es que se calculan las longitudes y latitudes límite del mapa que puede representar con los datos pasados y que se crea el mapa con esos límites; además, como los datos internos son geográficos (longitudes y latitudes), se deben crear los datos asociados al `plot` (coordenadas en pantalla), lo cual se hace llamando a propio objeto `bmap` con los vectores de longitud y latitud y luego ajustándolos al tamaño real de la imagen;

- `createNewPlot`: la primera diferencia fundamental es que la ventana actual debe cerrarse, pues las dimensiones pueden cambiar al cambiar los datos a representar; tras ello, la otra diferencia es que se crea el `plot` con la imagen asociada; el resto es semejante al método de igual nombre de `ScatterWindow`.

El resto de métodos siguen un esquema análogo a los correspondientes de `ScatterWindow`, y los métodos del gestor `handlerMapWindow` se modifican de forma idéntica a los de `handlerScatterWindow`.

3.4. Instalación y pruebas

La implementación final de **HINT** se hizo sobre una máquina virtual para garantizar su portabilidad a diversas plataformas. Para ello se empleó el *software* de virtualización *VirtualBox*, en su versión libre. La máquina virtual creada instala una distribución Ubuntu 10.04, con memoria RAM de 1536 Mb, memoria vídeo de 128 Mb, y tamaño de disco de 16 Gb. Dicha máquina virtual dispone de un único usuario `hint`, con contraseña `Hint-user`.

Para poder utilizar la aplicación, aparte de instalar el código fuente, es necesario instalar las bibliotecas que se emplean en la aplicación. La compatibilidad de versiones es muy delicada debido al gran número de bibliotecas que interactúan, con lo cual se lista a continuación cada biblioteca con su modo de instalación desde la consola:

- *Python*: es la versión 2.6, que es la que viene por omisión con la versión de Ubuntu 10.04.

- *chaco*: se debe instalar la versión 3.2.0-2, lo que se puede hacer directamente con `apt-get` mediante `sudo apt-get install python-chaco`.

- *dateutil*: esta biblioteca debe instalarse porque es requerida por *pandas* y no se instala con ella por un fallo en el manejo de dependencias; es necesaria la versión 2.1, con lo que la forma de instalarse esta biblioteca es bajarse el *tarball* `python-dateutil-2.1.tar.gz` y ejecutar `easy_install python-dateutil-2.1.tar.gz`.

- *pandas*: se requiere la versión 0.5.0; se instala bajando el *tarball* `pandas-0.5.0.tar.gz` y empleando `easy_install`.

- *matplotlib*: se requiere para el uso de *basemap* en la generación de mapas para la vista geográfica; para la versión que se empleará de *basemap* es necesaria la versión 1.0.1 de *matplotlib*, instalable a través de `easy_install` empleando el *tarball* `matplotlib-1.0.1.tar.gz`.

- *libgeos-dev*: necesaria también para *basemap*, puede instalarse con `sudo apt-get install libgeos-dev`, que instalará la versión 3.1.0-1.

- *basemap*: debe ser la versión 1.0.2; para instalarla se debe bajar el *tarball* `basemap-1.0.2.tar.gz`, descomprimirlo (`tar -zxf basemap-1.0.2.tar.gz`), modificar el archivo `setup.py` en su línea 30, escribiendo en esa línea `GEOS_dir = '/usr/'`; tras ello ya se puede ejecutar desde dentro del directorio `basemap-1.0.2` la orden `python setup.py install`, que instalará correctamente la biblioteca.

Tras esta puesta a punto ya se puede ejecutar **HINT**. La forma normal de hacerlo sería desde la línea de órdenes, escribiendo `python hint.py`; de hecho, así funciona en un entorno no virtualizado, pero no se ha conseguido hacerlo funcionar así en la máquina virtual.

Por tanto, la forma de ejecutarlo en la máquina virtual requiere lanzar un intérprete interactivo de *python* desde el directorio donde se encuentran los fuentes de **HINT**, escribiendo `python` en la línea de órdenes. A continuación, en el intérprete interactivo se deben escribir estas instrucciones:

```
import hint
hint.HINTWindow.configure_traits()
```

Al hacer esto nos aparece la interfaz principal, con el aspecto de la Figura 3.1. Al acceder a los menús `File` y `View` podemos seleccionar las distintas

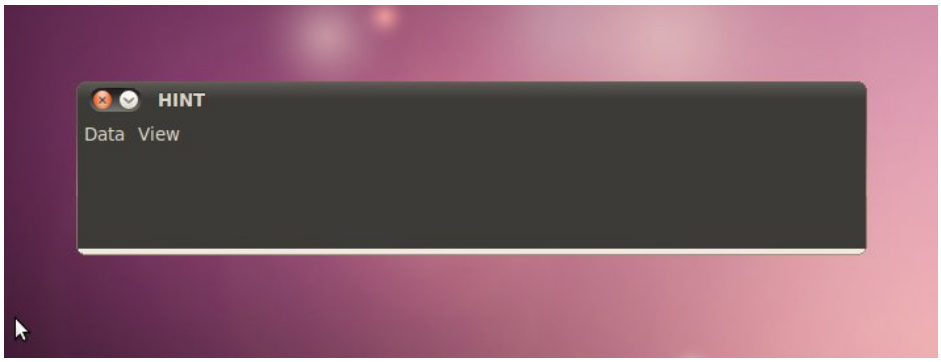

Figura 3.1: Ventana inicial de **HINT**.

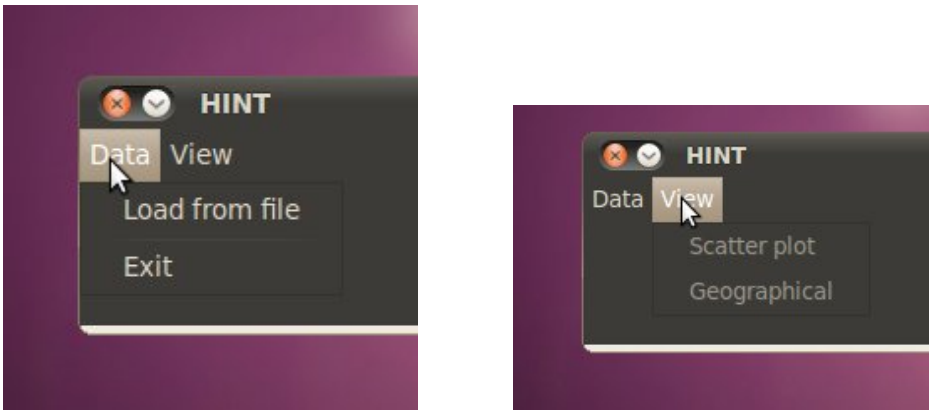

Figura 3.2: Menús `File` y `View`.

Figura 3.3: Cuadro de diálogo para la carga de fichero.

opciones, como se puede ver en la Figura 3.2; inicialmente, las opciones de `View` están inactivas.

La operación inicial habitual es la apertura de un fichero, que se hace empleando `Load from file` del menú `Data`. Al elegirla aparece el cuadro de diálogo de la Figura 3.3, donde lo habitual es pulsar el botón `Browse`, que lleva a un navegador de archivos semejante al de la Figura 3.4 del que se elige el fichero con los datos. Tras ello, se pulsa el botón `OK` para cargar los datos; `Cancel` puede usarse en cualquier momento para evitar la carga.

Una vez se han cargado los datos, se puede acceder a las opciones del menú `View`. Ya se elija `Scatter plot` como `Geographical`, lo que aparecerá será la ventana de selección de las columnas del fichero a representar en la vista elegida, como se muestra en la Figura 3.5. Tras pulsar `OK` se da paso a la vista elegida, con todos los datos seleccionados. En la Figura 3.6 se puede ver una representación con dos vistas *scatter* y una vista geográfica. La aplicación admite hasta 10 vistas *scatter* (tras lo cual se inactiva la opción del menú `View`) y una única vista geográfica.

Si se quiere realizar en las vistas una selección, se accede al menú `Selection` de la vista correspondiente, que se puede ver en la Figura 3.7. Cuando se

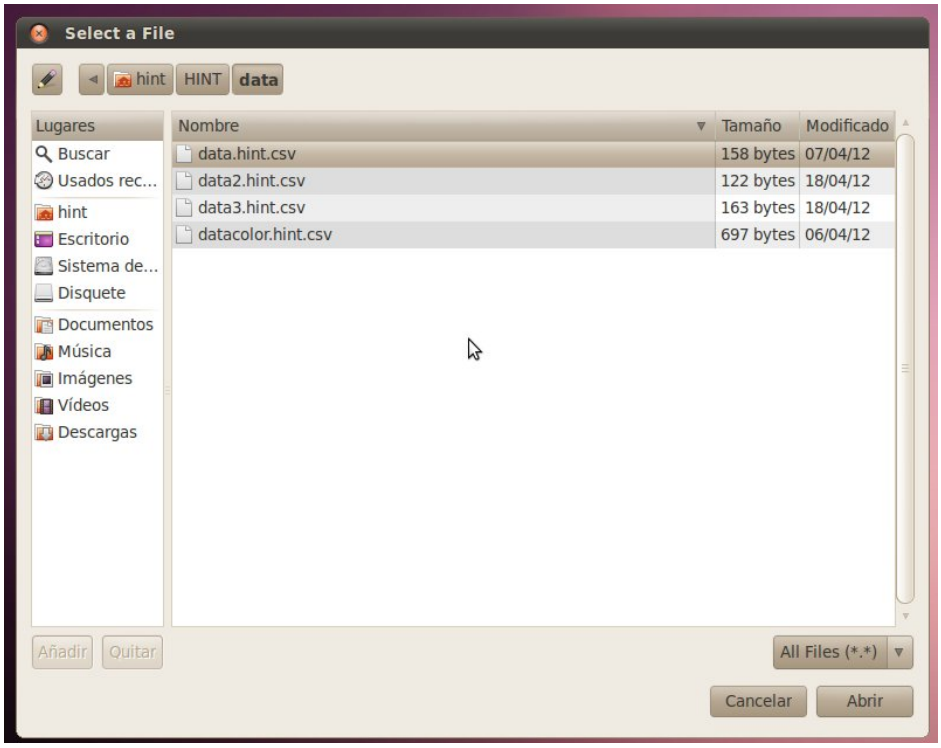

Figura 3.4: Navegador para la selección de ficheros de datos.

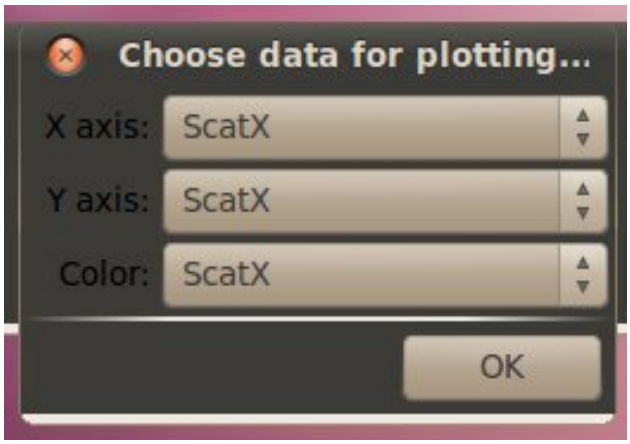

Figura 3.5: Ventana de selección de las columnas a representar en la vista.

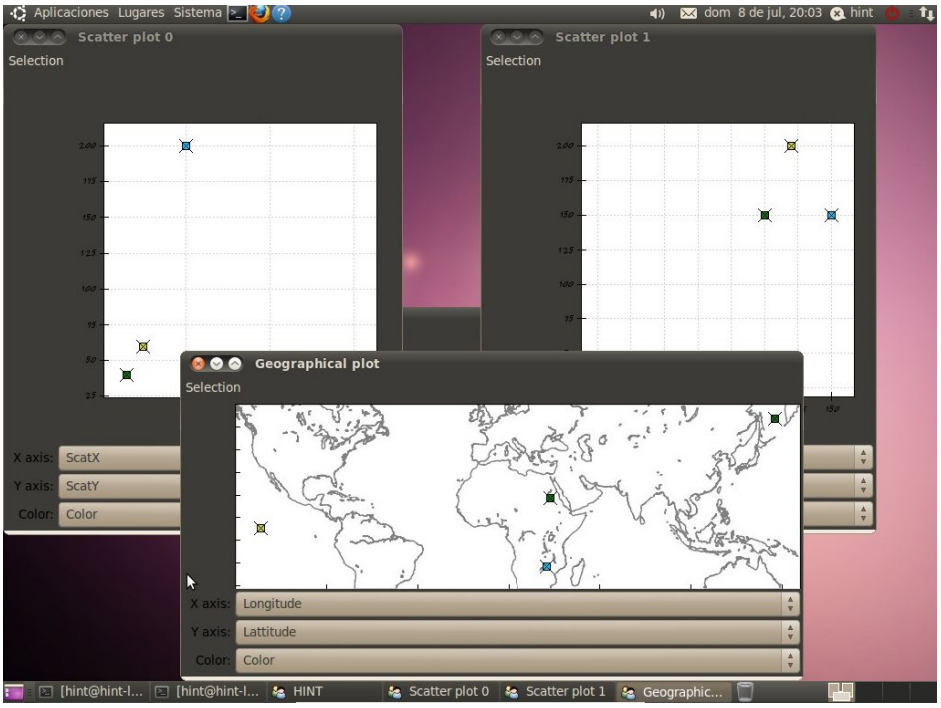

Figura 3.6: Vistas *scatter* y geográfica activas.

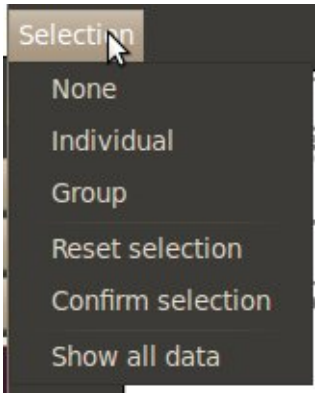

Figura 3.7: Menú `Selection` de las vistas.

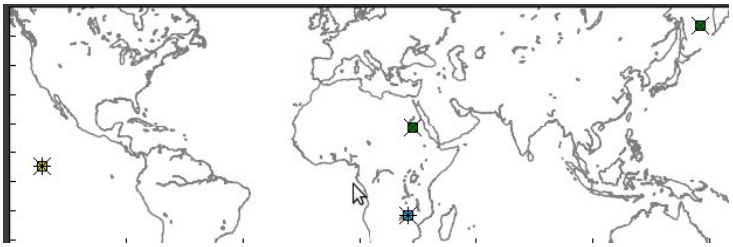

Figura 3.8: Proceso de selección individual. Los preseleccionados se marcan con una cruz (+).

realiza la selección individual se va marcando con una cruz cada elemento seleccionado (Figura 3.8); cuando se realiza la selección grupal, se puede usar el ratón para marcar un área de selección (Figura 3.9). Cuando se tiene la selección apropiada, se usa la opción `Confirm selection` para que en todas las vistas queden marcadas con un aspa sólo aquellas seleccionadas (Figura 3.10).

Las vistas se puede cerrar individualmente empleando los botones de uso de ventana (en la esquina superior izquierda). La carga de un nuevo fichero

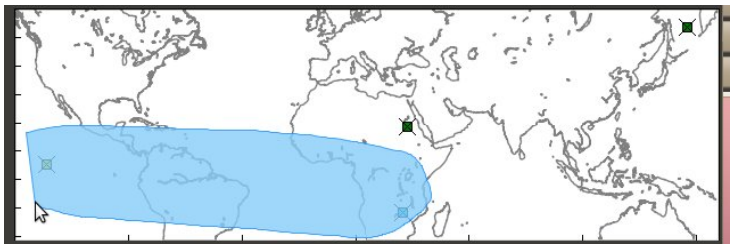

Figura 3.9: Proceso de selección grupal (área azul).

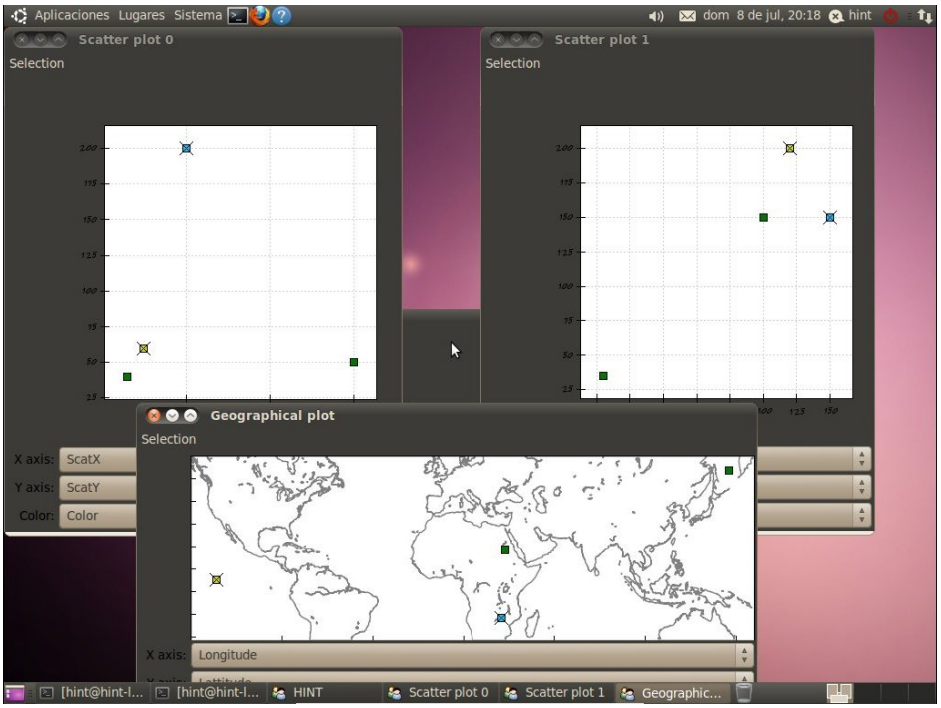

Figura 3.10: Resultado de una selección. Tras seleccionar sobre la vista geográfica los datos amarillo y azul, estos son los únicos que aparecen con un aspa en todas las vistas.

de datos provoca el cierre de todas las vistas. La elección de `Exit` en el menú `Data` provoca el fin de la aplicación.

HINT puede emplearse usando una versión de VirtualBox 3.1.6 o superior. Para ello basta con descargarse la imagen de disco correspondiente y situarla en el directorio donde se sitúen dichas imágenes (normalmente, en el *home* del usuario dentro de `.VirtualBox/HardDisks`). Luego se crea la nueva máquina virtual con el asistente, dándole como disco duro el descargado. Tras arrancar la máquina virtual se puede arrancar **HINT** como se ha descrito más arriba.

Capítulo 4

Conclusiones

En esta memoria hemos descrito el trabajo realizado para implementar una primera versión (o prueba de concepto) para el análisis gráfico interactivo de datos biológicos. Se ha construído una interfaz gráfica específica y se han empleado las bibliotecas apropiadas para conseguir las capacidades gráficas e interactivas. Se ha programado la aplicación en un lenguaje de programación (Python) que haría fácilmente extensible su funcionalidad. La funcionalidad de la aplicación se ha evaluado de forma informal con potenciales usuarios de la misma, que han mostrado un buen índice de satisfacción en su uso.

4.1. Ventajas y desventajas de la aplicación

La aplicación permite una interfaz amigable, intuitiva e interactiva para el manejo de los datos biológicos, con algunas funcionalidades que hasta el momento actual no estaban disponibles y que podrían resultar muy útiles en el análisis de datos biológicos masivos. En la implementación de la aplicación

se ha descubierto el uso de la biblioteca *pandas* como una excelente alternativa a la representación de datos con metadatos asociados (en este caso, etiquetas de individuos y de características biológicas). La implementación en Python la hace fácilmente extensible a otras funcionalidades, ya que es un lenguaje de programación bien conocido en la comunidad bioinformática.

Sin embargo, la aplicación es una simple prueba de concepto limitada en varios aspectos. A nivel de funcionalidad, sólo presenta dos tipos de vistas (*scatter* y geográfica), lo que puede ser limitado para ciertos análisis. Tampoco permite gestión de los datos seleccionados (grabarlos en disco, someterlos a nuevos procesos de análisis, etc.), aunque implementar estas funcionalidades es relativamente sencillo. La implementación en Python también la hace poco eficiente a nivel computacional (al ser un lenguaje interpretado). La elección de las bibliotecas *chaco* para la representación gráfica y la interacción no ha sido tan acertada como se pensó inicialmente, ya que ciertas funcionalidades de la aplicación han sido difíciles de implementar por la falta de correcta documentación de dicha biblioteca.

4.2. Trabajos futuros

La aplicación **HINT** presenta un buen punto de partida para realizar aplicaciones más ambiciosas en este campo. Por un lado, está la inclusión de otras posibles vistas a representar, como podrían ser árboles filogenéticos o mapas cromosómicos, así como nuevas funcionalidades de manejo de datos (salvado de datos, análisis de datos por aplicaciones externas y representación directa de los mismos, etc.). Por otro lado, estaría su reimplementación

en un lenguaje de programación más eficiente (tipo C++) o cambiar el uso de la biblioteca *chaco* por otras bibliotecas con mejor mantenimiento (por ejemplo, hay proyectos en marcha dentro de los mantenedores de *matplotlib* para añadir capacidades interactivas más flexibles). Un trabajo que catapultaría el uso de la aplicación sería implementarla como una aplicación web, de manera que cualquier usuario con un navegador pudiera acceder a la misma y realizar sus análisis biológicos interactivos.

Bibliografía

[Abdi and Williams, 2010] Abdi, H. and Williams, L. (2010). Principal component analysis. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2:433–459.

[Cairo, 2012] Cairo (2012). Cairo graphics library. `http://www.cairographics.org/`, Consultada junio 2012.

[Enthought, 2008] Enthought, I. (2008). Chaco: 2-dimensional plotting. `http://code.enthought.com/projects/chaco/`, Consultada junio 2012.

[Enthought, 2012] Enthought, I. (2012). Enthought - scientific computing solutions. `http://www.enthought.com/`, Consultada junio 2012.

[Hall, 2007] Hall, N. (2007). Advanced sequencing technologies and their wider impact in microbiology. *The Journal of experimental biology*, 210(Pt 9):1518–1525.

[Hunter, 2007] Hunter, J. D. (2007). Matplotlib: A 2D graphics environment. *Computing in Science and Engineering*, 9(3):90–95.

[Lambda Foundry, 2012] Lambda Foundry, I. (2012). Pandas: Python data analysis library. `http://pandas.pydata.org/`, Consultada junio 2012.

[Python-Software-Foundation, 2012] Python-Software-Foundation (2012). la 0.6.0. `http://pypi.python.org/pypi/la`, Consultada junio 2012.

[Sanger and Coulson, 1975] Sanger, F. and Coulson, A. R. (1975). A rapid method for determining sequences in DNA by primed synthesis with DNA polymerase. *Journal of Molecular Biology*, 94(3):441–446.