UNIVERSIDAD NACIONAL DE LA PLATA

FACULTAD DE INFORMÁTICA

TESINA DE LICENCIATURA

Titulo: Cliente para plataforma de búsqueda de biomarcadores con valor pronóstico/predictivo de cáncer
Autor: Diego Ariel Martínez
Director: Dra. Claudia Pons
Codirector: -
Asesor profesional: Mg. Matías Butti
Carrera: Licenciatura en Sistemas

Resumen

El objetivo de esta tesina es el diseño y desarrollo de una aplicación cliente extensible, portable, de fácil distribución en el marco de una plataforma, cuyo origen se remite a la tesis 'Metodología analítica e integradora para la generación de biomarcadores de pacientes con cáncer de mama sobre la base de perfiles de expresión génica' de maestría en Explotación de datos y descubrimiento de conocimiento de Matías Butti. En dicho trabajo se presenta una plataforma, denominada Bioplat, para la identificación, validación y optimización de biomarcadores en cáncer. La plataforma Bioplat está compuesta de varios componentes y una de los principales es el cliente desarrollado en este trabajo, en el cual se describe la integración del mismo con el resto de la plataforma, tanto desde el punto de vista funcional y de sus requerimientos, como desde lo técnico. Además, se presentan las soluciones de software propuestas a la estructura de la plataforma que permitan fácilmente, que ésta sea extensible por el equipo de desarrollo Bioplat, por desarrolladores que deseen enriquecer la plataforma con ideas y/o desarrollos propios compartidos luego con la comunidad Bioplat o por usuarios finales.

Palabras Claves

Bioinformática, genética, cáncer, Eclipse, OSGi, Java, R

Conclusiones

Con el cliente integrado a la plataforma, se logró realizar un producto de fácil distribución e instalación para los investigadores.
Se definió un marco de extensión consistente, que permite al cliente, y así a la plataforma, enriquecerse en nuevas herramientas para profundizar el estudio de los biomarcadores.

Trabajos Realizados

Definición del marco teórico que da soporte a la plataforma.
Descripción de las principales características del cliente a través de un recorrido de sus principales funciones.
Descripción del diseño y presentación de las herramientas e implementaciones más destacadas que dan solución al cliente, dotándolo de las características buscadas, motivación y objetivos de esta tesina

Trabajos Futuros

Inclusión de nuevas herramientas a través de los mecanismos definidos en esta tesina.
Adaptaciones de nuevas características de las tecnologías utilizadas para caracterizar al cliente gráfico
Adaptaciones como parte del crecimiento de la arquitectura general de la plataforma.

Fecha de la presentación: Mayo 2016

Universidad Nacional de La Plata
Facultad de Informática

Tesina de grado

Cliente para plataforma de búsqueda de biomarcadores con valor pronóstico/predictivo de cáncer

Directora: Dra. Claudia Pons

Asesor profesional: Mg. Matías Butti

Autor: Diego Martínez

Mayo 2016

A mi familia, por estar siempre
A Matías, por la oportunidad y su gran aporte
A Claudia, por el apoyo

Gracias

Índice general

Índice detallado

''La ciencia se compone de errores, que a su vez, son pasos hacia la verdad''
Julio Verne

1 Introducción

1.1 Antecedentes

Esta tesina surge como continuación del trabajo iniciado en la tesis de maestría de Matías Butti en *Explotación de datos y Descubrimiento del Conocimiento*, presentada en la facultad de Ciencias Exactas y Naturales de UBA en el año 2013, titulada ''Metodología analítica e integradora para la generación de biomarcadores de pacientes con cáncer de mama sobre la base de perfiles de expresión génica'' [1]. En el trabajo de Matías se presenta, fruto del trabajo desarrollado en conjunto con un grupo de profesionales del área de la oncogenómica funcional, un prototipo de Bioplat como integración de diferentes herramientas e ideas surgidas en el grupo de trabajo, para la identificación de biomarcadores con poder pronóstico/predictivo en cáncer; además de algunos resultados de análisis genómicos en un tipo de cáncer de mama.

Luego de concluida la tesis y de haberme incorporado al equipo, la aplicación continuó creciendo, consiguiendo publicaciones en importantes revistas de divulgación científica.

La oncogenómica adquirió gran relevancia en la comunidad científica de cáncer en los últimos años gracias a importantes avances en la obtención y procesamiento computacional de grandes volúmenes de datos. El trabajo referenciado está abocado al estudio y desarrollo de los fundamentos entre las relaciones existentes entre la presencia de ciertos genes y su relación con la progresión de tumores. Estos son los fundamentos básicos que sostienen la plataforma, en la cual se definen las entidades genéticas que detallaremos más adelante y las relaciones existentes entre ellas, aplicando algoritmos de *data mining* que permiten analizar la incidencia significativa de la interacción de los genes en determinados procesos oncológicos.

1.2 Conceptos de biología molecular

1.2.1 Genes

El gen es la unidad básica de información del ADN. Es a partir del gen que se inicia el dogma central de la biología molecular que concluye en la generación de las proteínas que cumplirán las funciones en nuestro organismo. El gen es también, la unidad básica de herencia de los seres vivos.

En el contexto de este trabajo, cumple un rol central pues, es a su vez, componente básico de las entidades que maneja la plataforma.

1.2.2 Genoma

El genoma de un individuo está dado por la secuencia de bases presentes en su ADN. Las bases son 4 posibles que las identificaremos con las letras A, C, G o T, por los nombres Adenina, Citosina, Guanina y Timina respectivamente. En el caso de la especie humana la secuencia tiene una longitud aproximada de 3.000.000.000 de bases. En ciertos sectores de esa secuencia, están codificados los genes, que en este contexto se pueden visualizar como una subsecuencia de esta larga secuencia que es el genoma del individuo.

1.2.3 Transcriptoma

Cuando un gen se convierte en proteína, se dice que el gen se expresó. No todos los genes se expresan en todos los tejidos humanos. Y dependiendo de la necesidad que tenga ese tejido de la proteína en determinado momento, el gen correspondiente se expresará más o menos. El transcriptoma de un paciente en un tejido determinado, es cuánto se expresó cada gen en ese tejido.

El experimento Bioplat (como se desarrolla en la sección Experimentos Bioplat) contiene parte del transcriptoma de cada paciente y sus datos de supervivencia.

1.2.4 Mutación genética

En la generación de proteínas y enzimas intervienen multitud de procesos biomoleculares, que tienen como punto de partida, la copia y replicación de genes. Estos genes pueden cambiar su estructura inesperadamente, dando lugar a errores en la replicación antes mencionada. En esta situación se dice que el gen presenta mutaciones. Estos genes mediante procesos biológicos normales son eliminados. Ahora bien, cuando estas mutaciones y errores se acumulan en gran cantidad pueden dar lugar a un descontrol en el proceso de replicación a nivel celular. Provocando que la población celular crezca descontroladamente imposibilitando su eliminación natural por parte del organismo. Esta acumulación y crecimiento descontrolado de células en alguna región del organismo, puede dar lugar a lo que se conoce como tumor, en primera instancia, y si la acumulación de errores en los procesos de replicación persiste, ''empeora'', puede derivar en neoplasias, o cáncer.

La biología molecular ayuda a comprender que el cáncer es, en efecto, una forma de enfermedad genética, debido a que depende de alteraciones en genes específicos que se encuentran dañados en las células enfermas. [2]

1.2.5 Genómica

La genómica tiene como objetivo principal predecir el comportamiento de los genes a partir de su secuencia o de sus interacciones con otros genes. El conjunto de fundamentos y técnicas involucradas en el estudio de la genómica es multidisciplinar, involucrando áreas tales como pueden ser, obviamente, la biología molecular, la estadística, la informática, entre otras.

La genómica es una de las áreas que más ha crecido en los últimos años a partir de importantes avances en las distintas ciencias, pero fundamentalmente en los soportes

tecnológicos para la secuenciación de ADN de manera cada vez más eficiente (por ejemplo los secuenciadores de última generación), y su posterior procesamiento, cuyo coste computacional es muy alto. Este procesamiento incluye la comparación de distintas secuencias de ADN, identificación del origen de proteínas a través de la identificación de estas secuencias, entre otras opciones.

El nacimiento del proyecto del Genoma Humano, básicamente el estudio de la secuenciación de ADN del ser humano, dio lugar a un abordaje del estudio de la genómica desde dos puntos de vista, uno estructural y otro funcional. En primer lugar, la genómica estructural, analiza la composición de las secuencias de ADN, mediante una descripción digitalizada y estandarizada de su estructura, y por el otro la genómica funcional que analiza sistemáticamente los resultados de la recolección de información de las secuencias de ADN, determinando la función e interacción entre genes, mediante la aplicación de aproximaciones experimentales, cuyo origen es un grupo celular o tejido, que sirve de muestra.

Es de particular interés, entonces, en nuestro caso, centrarnos en el análisis genómico funcional. Estos análisis se basan, en general, en el Transcriptoma, es decir en una cuantificación de los niveles de expresión de cada uno de los genes de interés.

En un trabajo mancomunado interdisciplinariamente, el análisis de datos genómicos ofrece los siguientes campos de estudio:

- Comparar secuencias genómicas (Genoma) similares presentes en diferentes entidades biológicas y comprender la función de dichas secuencias;
- Realizar predicciones acerca de todas las proteínas codificadas por una especie, tejido o tipo celular en particular;
- Establecer las variaciones genéticas entre distintas poblaciones de una misma especie;
- Secuencias genómicas de diferentes especies, y entender procesos evolutivos.

Uno de los principales objetivos de estos análisis funcionales, es la identificación de un grupo de genes, cuyo patrón de expresión se encuentre asociado a un fenotipo particular. El fenotipo puede ser una característica física, bioquímica o hasta incluso de

comportamiento; en nuestro caso una enfermedad. Este grupo de genes forma el concepto de *gene signature*.

La incursión en estos conceptos, y herramientas de análisis permitieron cambiar el enfoque de estudios de enfermedades complejas, como lo es el cáncer, hacia sus basamentos genéticos. De esta manera, los *genes signatures* toman gran relevancia en el ámbito de la medicina, cuando estos pueden ser interpretados como biomarcadores diagnóstico, pronóstico o predictivo, de una patología. Permitiendo una mejor clasificación de los pacientes, para así tomar decisiones con mejores certezas por ejemplo a la hora de indicación de tratamientos, pudiendo adelantar si estos tendrán o no efecto positivo en el paciente en particular.

1.2.6 Oncogenómica

Una de las áreas de la biomedicina que más se ha beneficiado con el estudio de la genómica en general y con el análisis y caracterización del genoma humano en particular, ha sido la oncología; tanto para entender los mecanismos básicos de los procesos de transformación neoplásica, como para el desarrollo de nuevos ensayos para un mejor pronóstico y evaluación del riesgo en pacientes oncológicos, que ayuden a determinar el tratamiento más adecuado. Esto ha abierto una nueva área de investigación en oncología, basada en la caracterización genómica de las neoplasias: la oncogenómica. [3]

El análisis de datos oncogenómicos, no solo ha permitido la identificación de aquellos grupos de genes que pueden estar diferencialmente expresados, nivel relativo de su presencia, entre una célula normal, un carcinoma in situ o uno infiltrante (se puede expandir más allá del sitio donde se inició), sino que además permiten mejorar el pronóstico de la progresión de cada subtipo tumoral, plantear el tratamiento más adecuado y brindar información para el desarrollo de nuevas estrategias terapéuticas.

Periódicamente son publicados y registrados centenares de nuevos estudios en oncogenómica funcional. Es posible acceder a estos estudios y realizar análisis propios a partir de ellos ya que son publicados en diversos repositorios accesibles públicamente.

Existen actualmente dos tipos de repositorios, los primarios y los secundarios. Entre los primarios podemos nombrar a NCBI[1] y *ArrayExpress*[2] que contienen datos directos de los patrones de expresión génica. Luego tenemos los secundarios, que son derivaciones de los repositorios primarios, con el agregado de nuevos datos e hipótesis como mutaciones, relaciones evolutivas, entre otros. En este grupo de repositorios podemos mencionar GeneSigDB[3] *(Dana-Farber Cancer Institute, Harvard School of Public Health, USA)* y MolSigDB[4] *(Broad Institute – MIT & Harvard, Cambridge, USA)* [1]

1.3 Contexto de la investigación original

La clasificación del cáncer en subtipos moleculares puede resultar de utilidad para una selección correcta y personalizada de tratamiento. Esta clasificación mejora el pronóstico basado meramente en características histopatológicas, las cuales en muchos casos no logran encontrar diferencias que, molecularmente, sean notables.

Entonces la hipótesis de la línea de trabajo en la que se enmarca esta plataforma, es que es posible identificar un conjunto de genes, cuyos niveles de expresión permitan pronosticar cómo progresará la enfermedad, en casos donde los métodos actuales no lo permitan.

La hipótesis que planteó el camino inicial de la investigación, es que la integración de múltiples ''*gene signatures*'' de perfiles de expresión génica permitiría identificar no solo un espacio pronóstico en común, sino también un grupo de genes común entre estudios (*'meta-signature', nuevo gene signature a partir de la combinación de otros*) con poder predictivo/pronóstico, que represente las principales vías de señalización desreguladas en la enfermedad en estudio. [1] La generación de *meta-signatures* fue una de las motivaciones iniciales para iniciar la plataforma; sin embargo con el crecimiento de Bioplat, esta herramienta se convirtió en solo una de las posibles formas de generar un *gene signatures* (ver sección Herramientas para generar/importar *gene signatures*).

[1] http://www.ncbi.nlm.nih.gov/geo/
[2] http://www.ebi.ac.uk/arrayexpress/
[3] http://compbio.dfci.harvard.edu/genesigdb/
[4] http://www.broadinstitute.org/gsea/msigdb/index.jsp

1.4 Motivaciones y objetivos

El software resultante comentado en los Antecedentes, al que referenciaremos como prototipo, tenía por objetivo realizar prueba de conceptos sobre las ideas surgidas en el grupo. Luego de validadas estas ideas y de encontrar utilidad a las mismas, el desafío siguiente era integrarlas y generar herramientas unificadas en una única plataforma, que cumpla todos los requisitos que un software de calidad exige. Una de las componentes de esta plataforma era el cliente. Así surge el primer objetivo de esta tesina, que es justamente el desarrollo de una aplicación preparada para ser parte de esta nueva plataforma.

Dada la complejidad inherente del área, el nuevo cliente debe proveer de una interface rica en características visuales, que sea relativamente sencilla de utilizar unificando e integrando los distintos conceptos. Con capacidad de ofrecer asistencia a los usuarios en las distintas funcionalidades que la plataforma ofrece.

El prototipo presentaba, por otro lado, una gran dificultad de distribución a las PCs de los usuarios que quisieran tenerlo. Muchas veces influía y entraba en conflicto con otras herramientas que el potencial usuario tuviera preinstalada. Requiriendo así la intervención de los expertos en el área para poder configurar la aplicación en esas PCs. Claramente esto atenta contra la distribución de la aplicación abiertamente, dando lugar a un nuevo objetivo de esta tesina, que es justamente permitir la distribución de la aplicación de forma sencilla y sin provocar las interferencias que podía traer la instalación del prototipo.

Con el uso del prototipo se desprendió una particularidad importante. En las computadoras donde fue instalado, solían contar con un conglomerado de aplicaciones satélites que se ejecutaban conjuntamente con el prototipo, pero sin presentar ninguna particularidad de integración con este, ya que después de todo no era su objetivo tampoco, y cuando así lo requería se armaba una herramienta *ad hoc* para disparar el proceso de ''integración'' manualmente que adaptara de alguna manera los datos y los

hiciera fluir de una aplicación a la otra. Muchas veces este proceso era engorroso, y propenso a errores con lo cual lo hacía insostenible en el tiempo. En este punto es donde se empezó a pensar en la construcción de una plataforma integral que ofreciera posibilidades de integración con herramientas de este tipo, utilidades, cualquier otro aporte que pudiera enriquecer la plataforma. De esta manera aparece un nuevo objetivo para el cliente, que es dar soporte a extensiones para las que la plataforma esté preparada

Otro de los grandes objetivos del cliente, es dar un marco de extensibilidad a las operaciones que se realizan sobre las distintas entidades de la plataforma. La motivación de este punto es permitir que de un modo conveniente se pueda enriquecer el cliente con operaciones provistas por los distintos usuarios o desarrolladores que así lo deseen. De esta manera se defina una API[6] que permita la interacción con los elementos del cliente abstrayéndose de la implementación interna de este, focalizando así el esfuerzo en el desarrollo a contribuir, a la vez que permita una integración natural con la plataforma.

[6] *Application Programming Interface*

2 Marco de trabajo

En esta sección introduciremos el marco de trabajo informático del cual se parte para el desarrollo del cliente. Se esquematizará la arquitectura general de toda la plataforma al momento de iniciar el nuevo desarrollo. Se profundizará en los aspectos de mayor relevancia a la hora de dar soporte al desarrollo del cliente gráfico, es decir los elementos con los que interactuará directamente.

Por último mencionaremos la dinámica de trabajo del grupo sobre la plataforma en general, indicando algunas prácticas llevadas a cabo para tener un desarrollo documentado, colaborativo, planificado y consistente.

2.1 Arquitectura general del sistema

La Figura 2.1-1, esquematiza la arquitectura general de la plataforma en instancias de prototipo, previo a comenzar con el desarrollo del cliente motivo de esta tesina.

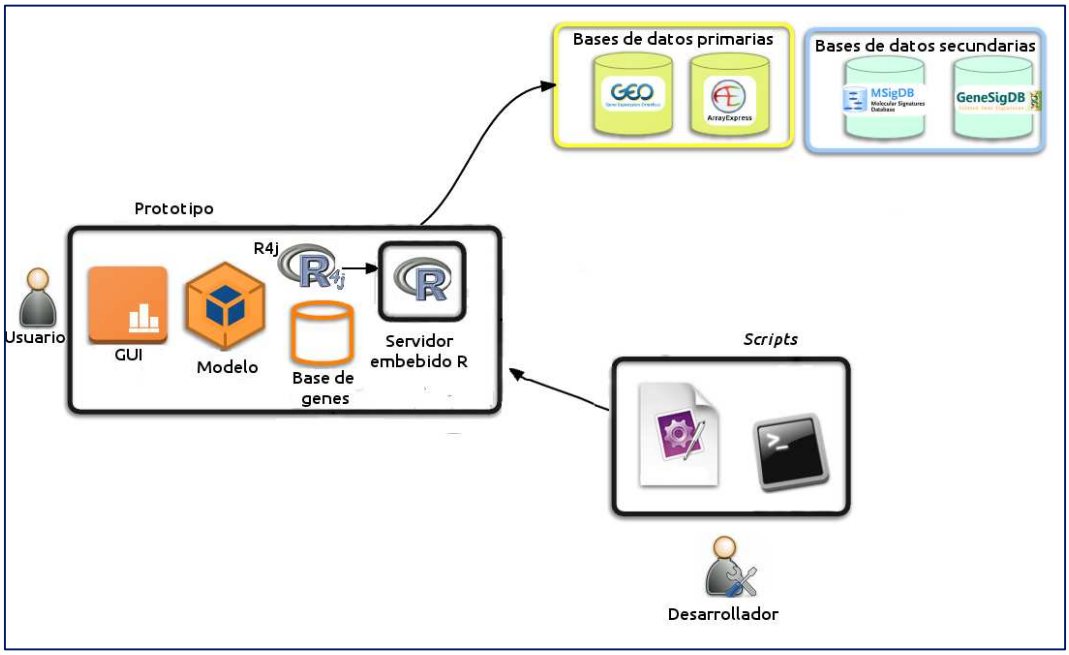


Figura 2.1-1 - Primera arquitectura general de la plataforma, en instancias de prototipo/pruebas de concepto

Podemos comentar y notar con respecto a la arquitectura ilustrada en la Figura 2.1-1, que el cliente contiene embebido un subcomponente llamado *''Servidor embebido R''*, el cual en este escenario se ejecuta en la misma máquina donde lo hace el cliente gráfico. Esto hacía considerablemente más compleja la distribución y posterior instalación del cliente en las PCs de los usuarios finales, atentando contra la portabilidad del cliente.

Las herramientas, *''scripts'',* que fueron surgiendo alrededor del cliente *prototipado*, tenían una limitación muy grande al no poder integrarse fácilmente a la aplicación prototípica, ya que esta no estaba preparada para esto, con lo cual muchas de estas herramientas quedaban con mucho potencial por explotar al no poder hacer un mejor uso de los datos o recursos de la plataforma.

Cabe mencionar que el prototipo presentaba baja riqueza a nivel visual, básicamente por su esencia de ser un prototipo. Esto implicaba dificultad en el uso.

2.2 Modelo de la plataforma

El cliente consta de un núcleo de desarrollo, de ahora en más Modelo, que se encarga de proveer las entidades del sistema: genes, *gene signatures*, experimentos Bioplat, etc. Además define los lineamientos a seguir para su extensión, dando la posibilidad de proveer de nuevos algoritmos de optimización de biomarcadores, posibilidad de nuevas fuentes de donde importar los *datasets* para hacer nuevos análisis, entre otros puntos donde la plataforma básica permite su extensión. El modelo está principalmente desarrollado en Java.

El Modelo hace uso de la capa R4j (ver Figura 2.1-1), desarrollada por el propio equipo, como intermediario entre R y Java, para realizar los cálculos estadísticos requeridos por la manipulación de las entidades del sistema.

El Modelo mantiene, por otra parte, una base de datos local de genes y otras entidades básicas, de las cuales hace uso intensivo. Estas entidades se actualizan periódicamente, pero resultan ser relativamente estables en el tiempo. El uso intensivo, y con el objetivo de hacer eficiente este uso, determinó que la base de datos se distribuya junto con el cliente, permitiendo tener dicha base en memoria. Se utiliza el motor de base de datos h2[7], el cual es puramente desarrollado en java, completamente portable y permite ser ejecutado de modo embebido, es decir como parte totalmente integrada, en este caso al cliente gráfico. En esta base de datos cada gen tiene un conjunto de identificadores, dependiendo de los distintos estándares definidos y soportados por la plataforma, además otros datos de interés.

2.3 Dinámica del proceso de desarrollo

La dinámica del desarrollo se organizó en el marco de desarrollo ágil. Tomando pautas del modelo *Scrum*. Principalmente se tomaron los siguientes aspectos de esta metodología: se mantiene un *backlog* de tareas, se arman *sprints* de dos semanas de duración, donde se asignan a cada uno de los integrantes del grupo tareas que irán

[7] http://www.h2database.com/

resolviendo. Es decir, perseguir la idea de tener nuevos *features* implementados cada 15 días y poder entregarlos para su posterior aceptación o no por parte del grupo de usuarios que forma parte integral del proyecto. De esta manera, se delineó una dinámica retroalimentada de *repriorización* de tareas o cambio de requerimientos que, al tenerlos acotados en dichos *sprints,* minimizó los riesgos de desviar el desarrollo hacia implementaciones que no sirvieran o que no cumplieran las necesidades exactas del usuario. Usuario que muchas veces va delineando los requerimientos, cambiándolos o refinándolos, a medida que interactúa con los nuevos *features* entregados.

	A	B	C
1	Responsable	Sprint	Nombre
93	DIEGO	2016-03-07SP19	partir base-lib: terceros y propios
94	DIEGO	2016-03-07SP19	actualización target eclipse 4: etapa 1 legacy eclipse 3, migración nuevo eclipse 4
135	DAVID	2016-03-07SP19	En caso de que este remoto, el wizard de import from .cel debe tirar un dialogo de error en vez de abrir el wizrd.
144	DAVID	2016-03-07SP19	Con el cambio de matias, ahora al tener un valor incorrecto en los datos de expresion, se cambia por NaN. Sin embargo, el gen no es eliminado. O el sample tampoco eliminarse en caso de q para todos los genes tenga NAN
145	DAVID	2016-03-28SP20	HeatMap: En el heatmap, en vez de mostrar el geneId mostrar el GeneSymbol
146	DAVID	2016-03-28SP20	!!!!HeatMap: Al heatMap atenuar los colores para que queden más marcadas las diferenciass. Para eso aplicar median-center primero por samples y luego por genes. replace each value by [value – Median(row that value belongs to)]. Agregar además el ajuste de color para que quede en un rango menor y así resaltar más las diferen
147	DAVID	2016-03-28SP20	!!!KaplanMeier: referenciar que color es cluster1 y que color es cluster 2!!!
148	DAVID	2016-03-28SP20	HetaMap: En el heatmap, sacar el label del sample
149	DAVID	2016-03-28SP20	El export to text file de la validacion quedo fea la ventana. Hay un espacio enorme arriba
150	DAVID	2016-03-07SP19	Reparar el boton de R. Ojo que comente la linea que iba llevando el registro porque en algunos casos de importFromCTGA que tenian muchos samples me tiraba outOfMemoryException.
151	DAVID	2016-03-07SP19	En la validacion, en caso de que el os_month o el os_statuis sea vacio, eliminar el caso
152	DAVID	2016-03-07SP19	Error de la estadistica desde el experimento. CLassCastException
153	DAVID	2016-03-07SP19	Cuando hay más de un sample que debe ser eliminado (por ejemplo porque ambos samples tienen todos valores incorrectos en los datos de expresion), en el observa solo uno. Esto es necesario para considerar terminada la tarea 156
154	DAVID	2016-03-07SP19	Cuando hay que eliminar más de un gen (porque ambos genes tienen valores incorrectos) en el observations informa solo uno.
155	DAVID	2016-03-07SP19	Poder informar cuales fueron los samples que se sacaron de la validacion por no tener month o status con valores validos. En ese mismo lugar hacer una validacion de correcto para lanzar la validacion. Agregar tambien un campo observations (tanto en la ventana antes de lanzar la validacion como en la solapa de suvivalAnalysis de g que muestra los resultados). Este campo debe informar cuales samples fueron removidos.
156	DAVID	2016-03-07SP19	En caso que no se pueda clusterizar,por ejemplo porque todos los valores son muy iguales entonces da un solo cluster , da un error entendible?
161	JUAN	2016-02-23SP18	Al arrancar Bioplat, verificar ambas conexiones (bioplat server y sitio web). Si es ok, mostrar en la vista de mensajes un mensaje que diga que se pudo conectar a biop internet. Si falla, le sugerimos que utilice los botones de comprobar conexion para conocer las sugerencias para reparar el error.
162	JUAN	2016-02-23SP18	Agregar menu Check connections. Allí debe haber dos submenues: 1-"check connection with Bioplat Server". Esta opcion ejecuta el codigo del test R4JRemoteConnec falla se debe sugerir que le pida a su adminsitrador que le de permiso a la url de nuestro servidor (buscar en el codigo cual es) y como alternativa a eso que vaya a window/prefernece/proxy para configurar el proxy a mano. 2-"Check connection with other web bioinformatic tools" que debe pegarle a www.google.com. Si falla se d que le pida al admin que le de acceso a todas las urls que estan publicadas en el file de geneURLS
163	JUAN	2016-02-23SP18	Eliminamos la entrada de configuracion de proxy del windows/preferences
164	DAVID	2016-03-07SP19	HetMap: En el hetMap agregar la referencia de lo que es up y lo que es down (mostrar como una degrade desde el rojo hacia el verde) poniendo del lado del verde dov del rojo up.
165	DAVID	2016-03-07SP19	Corregir la exportación de la matriz de expresion de una validacion cuando al experimento se le sacaron algunos samples. En partiular falla cuando se selecciona expo
166	DAVID	2016-03-07SP19	Cuando desde el start se lanza el dialogo de import from copy external gene signature from external database, aparece ocupando toda la pantalla a lo ancho. No ocurr lanza desde la welcome
167	DAVID	2016-03-28SP20	Implementar el PSO completamente en R
168	DAVID	2016-03-07SP19	Configurar servidor productivo. Para ello coordinar con Cristian Damomio (cristiand@uai.edu.ar). Intentar hacerlo sobre docker.
169	DAVID	2016-03-28SP20	Monitor de actividad en el rserve productivo
170	DAVID	2016-03-28SP20	Armar la maquina virtual de Docker con r y el web server. Charlarlo con Diego.
			Problemas actuales de Linux: varias columnas se ven mas chicas que el tamaño maximo del texto, no se ve la vista de genes y tira error (para lo cual es necesario inst

Backlog bioplat client | Backlog Bioplat cloud | Backlog bioplat admin | Backlog Bioplat backend | Refactor | Grandes Temas | Bug

Figura 2.3-1 - Backlog de tareas

En la Figura 2.3-1 se presenta un extracto del *backlog*, donde se aprecia la organización de diversas tareas agrupadas en *sprints*. Cada tarea cuenta con una breve descripción y el o los miembros del grupo involucrados en la resolución de la misma.

Para la administración del código fuente, y como es común en proyectos de estas dimensiones, se utiliza un repositorio de control de cambios. Se utiliza un esquema de versionado definido, y un conjunto de convenciones para hacer el avance del desarrollo más simple. Actualmente se está migrando de SVN a GIT. Ya que este último presenta considerables ventajas sobre el primero.

Entre los miembros del grupo se utiliza Dropbox[8] para mantener compartida y actualizada toda la documentación del proyecto.

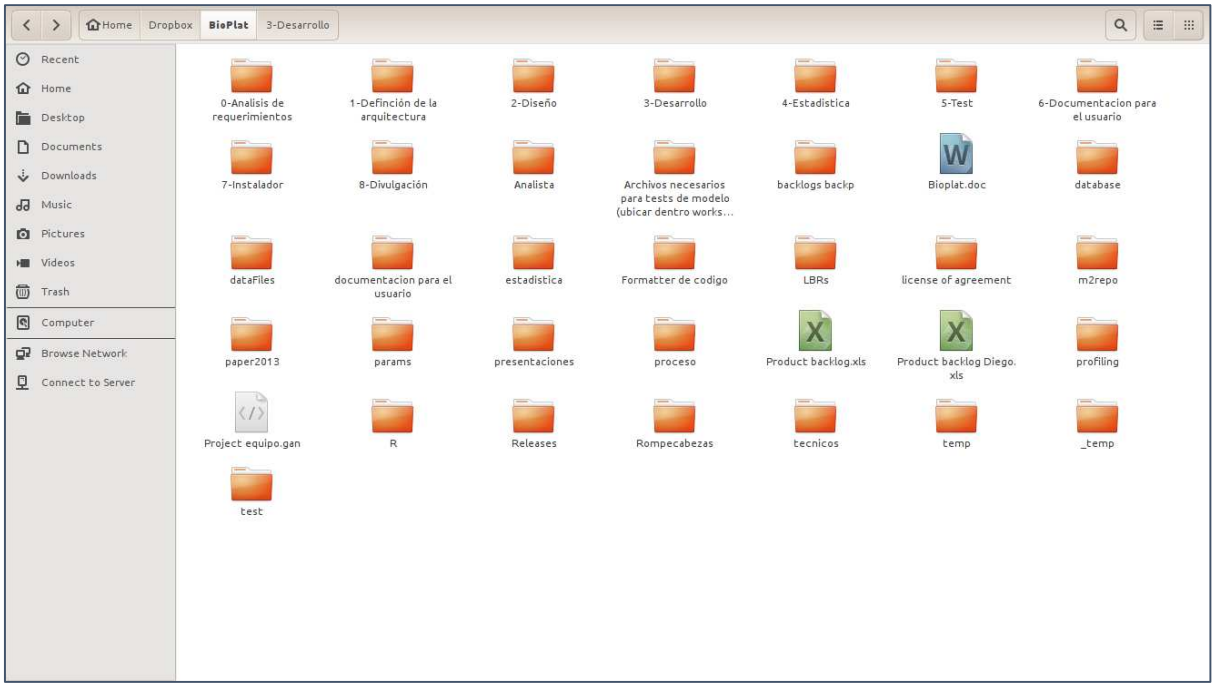


Figura 2.3-2 - Directorio de documentación y recursos compartidos

Así se organiza el repositorio en distintos directorios según el tipo de información que estos contengan (Figura 2.3-2). Se tiene por ejemplo: ''0-Análisis de requerimientos'' con todo lo definido por los usuarios, ''1-Definición de la arquitectura'' con documentos de análisis técnicos sobre los diversos temas correspondientes en estudio (*clustering*, etc), ''2-Diseño'' que contiene diagramas UML con el diseño de la aplicación, ''3-Desarrollo'' aquí se detalla los requisitos a la hora de armar un entorno de desarrollo para emprender algún cambio, ''4-Estadística'' con explicación de los métodos utilizados, ''5-Test'' que contiene archivos para realizar los test manuales y test de integración automatizados, ''8-Divulgación'' que contiene presentaciones de la plataforma en diferentes Congresos locales e internacionales; otro tipo de ejemplo es ''*mp2repo*'', directorio utilizado como repositorio de dependencias, que ante nuevas versiones de componentes que sea necesario compartir entre los desarrolladores, son colocadas aquí y luego vistas y compartidas por todos los miembros del grupo; por último podemos ver también la planilla que representa al *backlog*, documento que mencionamos anteriormente. Entre otros archivos y directorios.

[8] https://www.dropbox.com/

3 Resultados

Los resultados sobre el cliente tienen dos aristas. Por un lado las soluciones funcionales propuestas y por el otro sus soluciones tecnológicas, es decir la implementación con la que se da respuesta a los requerimientos funcionales. Con el fin de hacer un abordaje ordenado, presentaremos primero los resultados del cliente a nivel funcional, lo que nos servirá para profundizar en el entendimiento del sistema tanto a nivel funcional como de dinámica de uso, a través de un recorrido por las distintas partes que lo componen. Para luego hacer una mejor aproximación de los aspectos tecnológicos y de programación que dan soporte a estas funcionalidades.

3.1 La Plataforma desde el cliente

3.1.1 Introducción

La aplicación cuenta con una pantalla de bienvenida que grafica distintos caminos posibles de inicio:

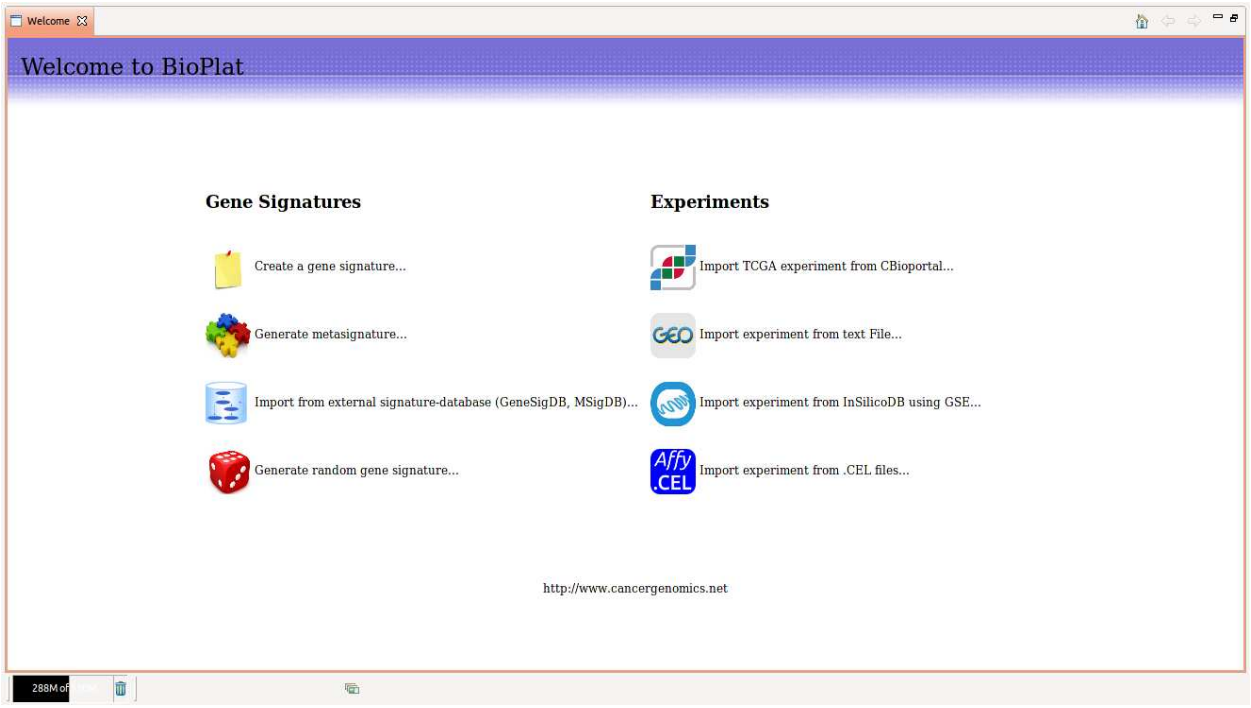


Figura 3.1-1 - Pantalla de bienvenida a la plataforma

Uno de estos caminos puede comenzar con la confección de un *gene signature* para la posterior validación y eventual optimización del mismo. Otro modo de dar inicio, es importando experimentos, que luego servirán en los distintos análisis que realiza la plataforma.

Iniciaremos entonces la descripción del cliente, analizando un *gene signature* y las diferentes herramientas que ofrece la plataforma para trabajar con los mismos.

3.1.2 *Gene Signatures*

3.1.2.1 Definición

A partir de ahora, utilizaremos el término *gene signature* como conjunto de genes que representan un potencial biomarcador con valor pronóstico del cáncer en estudio. Es decir, representa una hipótesis. A partir de validaciones estadísticas con datos de supervivencia usando Bioplat sobre diferentes experimentos y posteriores validaciones, este *gene signature* se podría convertir en un biomarcador.

3.1.2.2 Visualización

Un gene *signature* cargado en la aplicación se aprecia como lo ilustra la Figura 3.1-2

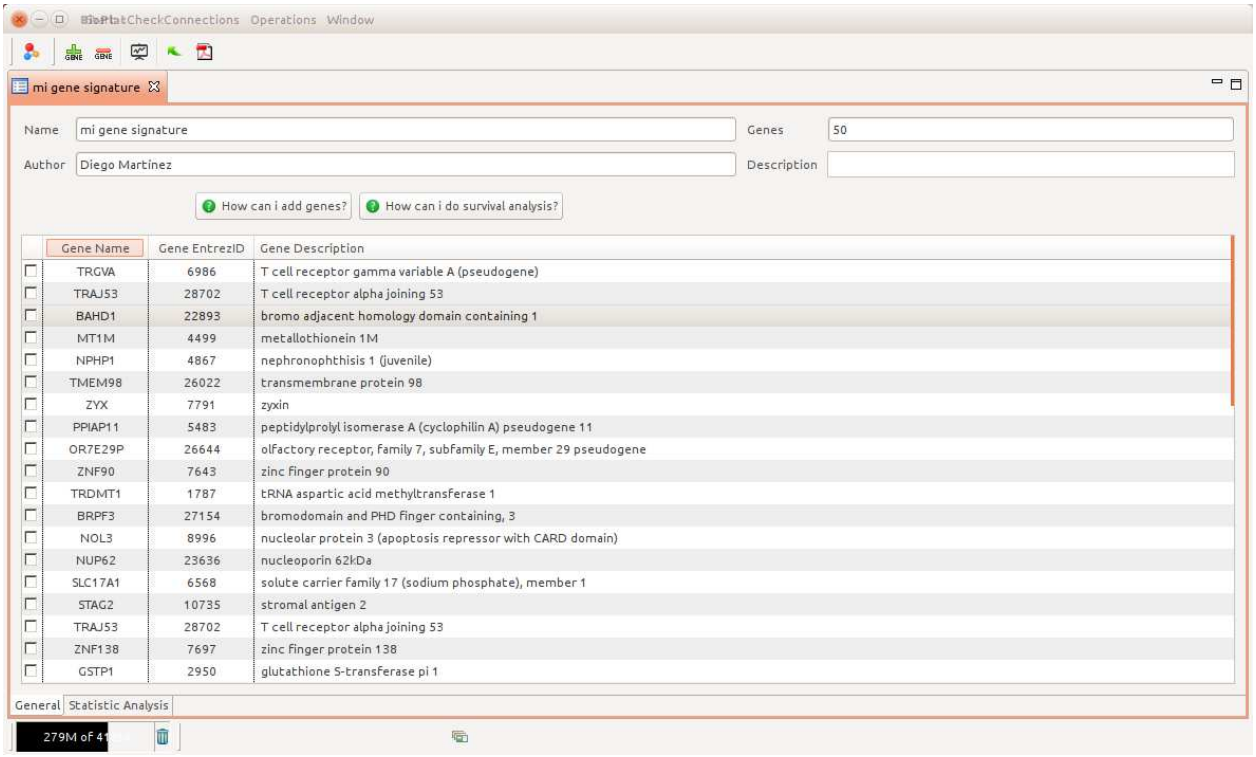


Figura 3.1-2 - Visualización de un gene signature

Las entidades del sistema se visualizan y modifican en lo que llamaremos Editores. Definiremos, a modo de introducción, Editor como la componente visual que

agrupa y muestra una entidad compleja. En el editor del *gene signature* (Figura 3.1-2), se ven dos grupos de datos. La solapa titulada ''General'' donde se listan, en forma tabular, los genes que conforman la entidad. Como en cualquier grilla, el usuario puede elegir ver u ocultar las columnas según sus necesidades. Las columnas que aparecen como visibles por defecto son: *GeneSymbol*, el *EntrezID* y una descripción corta. Están ocultas las columnas con IDs alternativos de los genes. La información de cualquiera de los genes puede ser profundizada, utilizando la vista de genes (ver sección Vista de genes).

En la solapa titulada ''*Statistic Analysis*'', se presentan los resultados de los diferentes análisis estadísticos utilizados para validar el poder pronóstico que puede llegar a tener el conjunto de genes postulado por el *gene signature* en cuestión.

Para realizar estos análisis, se utilizan estudios de perfiles de expresión génica para los cuales se dispongan datos de supervivencia. Estos estudios, que en la plataforma son llamados experimentos Bioplat, pueden ser importados a la misma a través de diferentes herramientas que esta ofrece (ver sección Herramientas para generar/importar experimentos Bioplat). Estos experimentos pueden provenir de estudios públicos o estudios propios del investigador. Así, un análisis se lleva a cabo aplicando un proceso a un *gene signature* en combinación con un experimento. Como resultado de este proceso se presentan diferentes apreciaciones estadísticas. Todos los cálculos estadísticos que se realizan están apoyados en el soporte que da el lenguaje de programación R[9], especialmente diseñado para este fin. Los cuales se procesan en el servidor R provisto, como se mencionó en secciones anteriores y que retomaremos más adelante (ver sección Servidor R como servicio). Todos los scripts R ejecutados por la plataforma pueden ser exportados por el usuario para replicarlo o combinarlo con otras herramientas. En la sección Validación estadística de *gene signatures*, se profundizará en los resultados estadísticos ofrecidos por la plataforma.

3.1.2.3 Operaciones básicas

Cuando el editor de un *gene signature* tiene foco, el cliente gráfico habilita un conjunto de operaciones contextuales específicas para esta entidad, tal como muestra la Figura 3.1-3. Las operaciones más destacadas, recuadradas en la misma figura, están

[9] https://www.r-project.org/about.html

disponibles también como botones en lo que llamaremos barra de herramientas o *toolbar*:

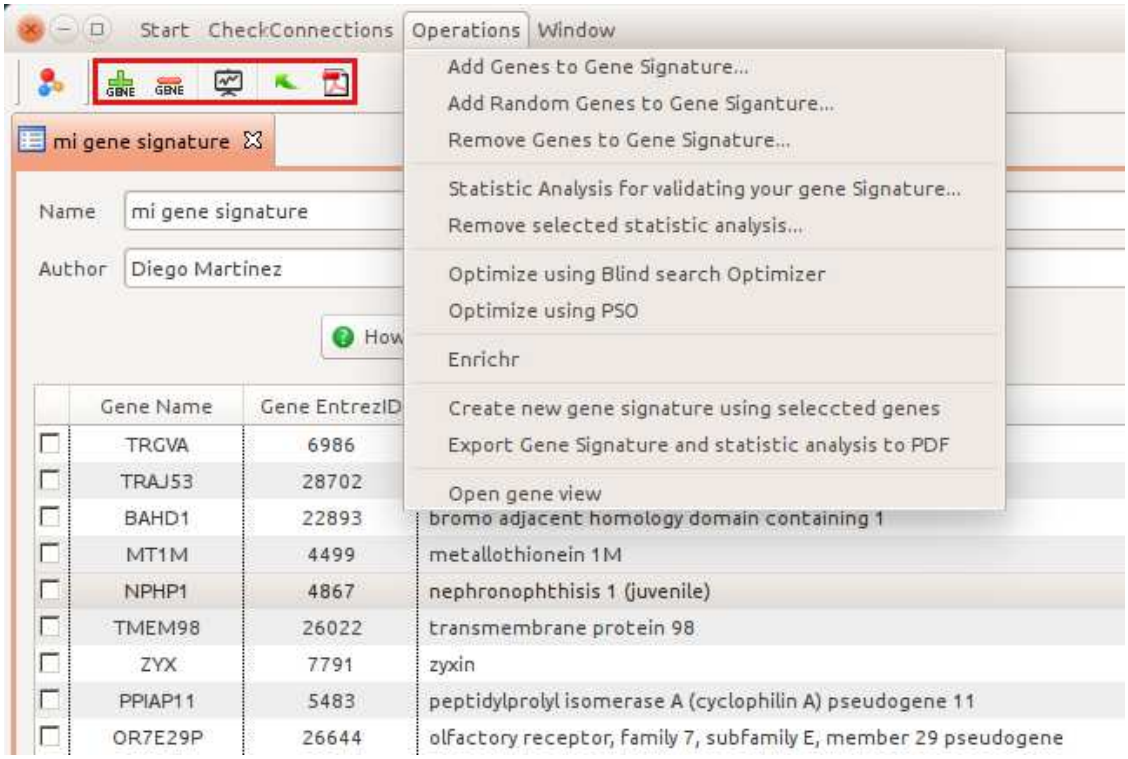


Figura 3.1-3 - Operaciones básica de un Gene Signature

A continuación, haremos una breve descripción de las operaciones disponibles:

- Edición de la lista de genes: agregar o remover genes manualmente, a partir de la indicación de sus nombres o cualquiera de sus identificadores, recordemos que la herramienta intenta normalizar los identificadores de genes, permitiendo que estos

sean referenciados según el sistema de identificación que utilice el usuario final.

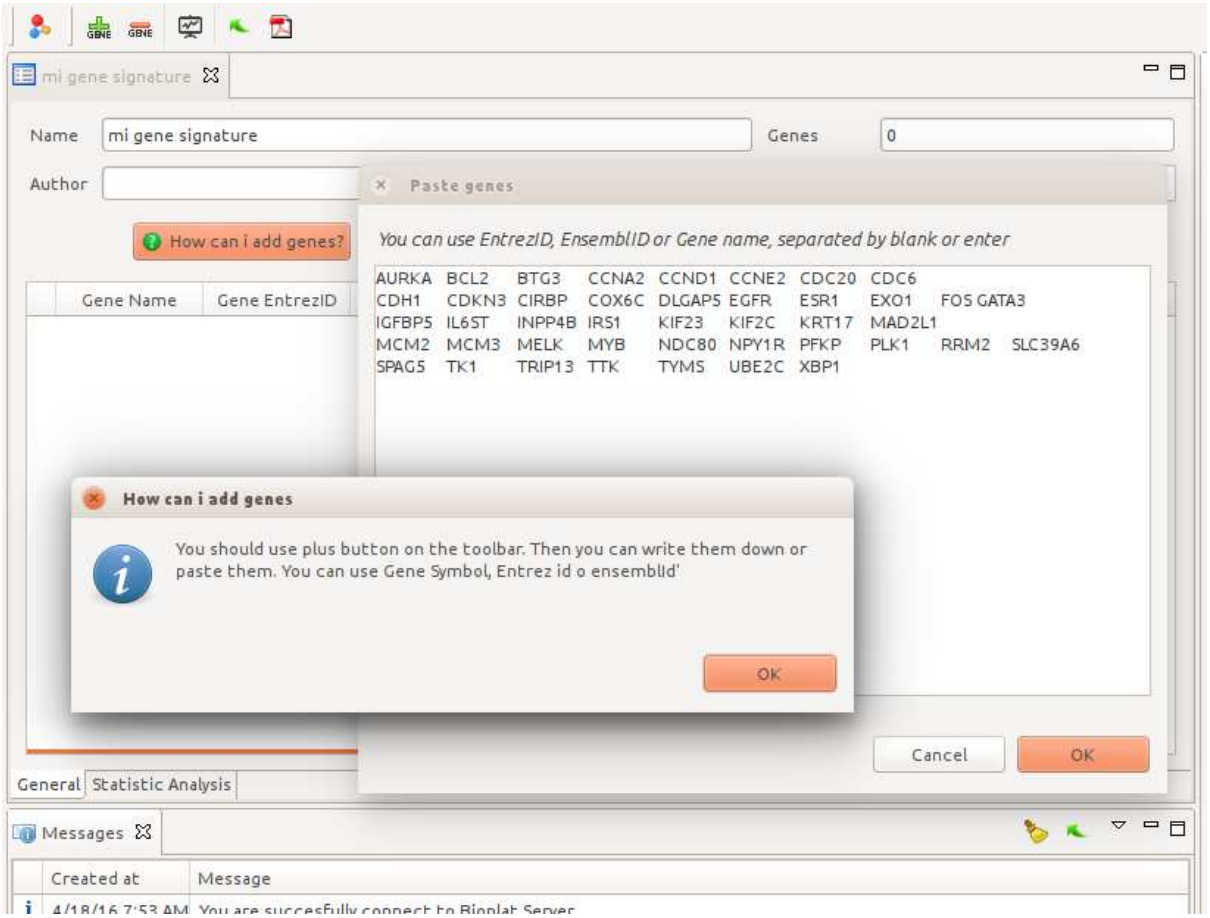


Figura 3.1-4 - Adición de genes a un Gen Signature

- Análisis estadísticos: Un *gene signature* es sometido a distintos análisis estadísticos, para esto es escoge un conjunto de experimentos Bioplat junto con una configuración para su aplicación. Estos resultados se aprecian en la solapa correspondiente. Veremos esta operación detalladamente en la sección Validación estadística de *gene signatures*;

- Optimización de *gene signature*: La plataforma actualmente ofrece un conjunto de algoritmos para hacer una optimización de la entidad, por ejemplo reducir el número de genes involucrados en los análisis. Este punto representa también requerimientos importantes del cliente, serán detallados en la sección Optimización de *gene signatures*;

- Integración con otras herramientas: Un ejemplo de esta herramienta es la integración con el sitio Enrichr[10], que da información relevante al conjunto de genes presentes en el *gene signature;*

- Realización de operaciones con un conjunto de genes de la entidad. Es posible por ejemplo, la creación de un nuevo *gene signature* a partir de un grupo de genes, previamente seleccionados (Figura 3.1-5);

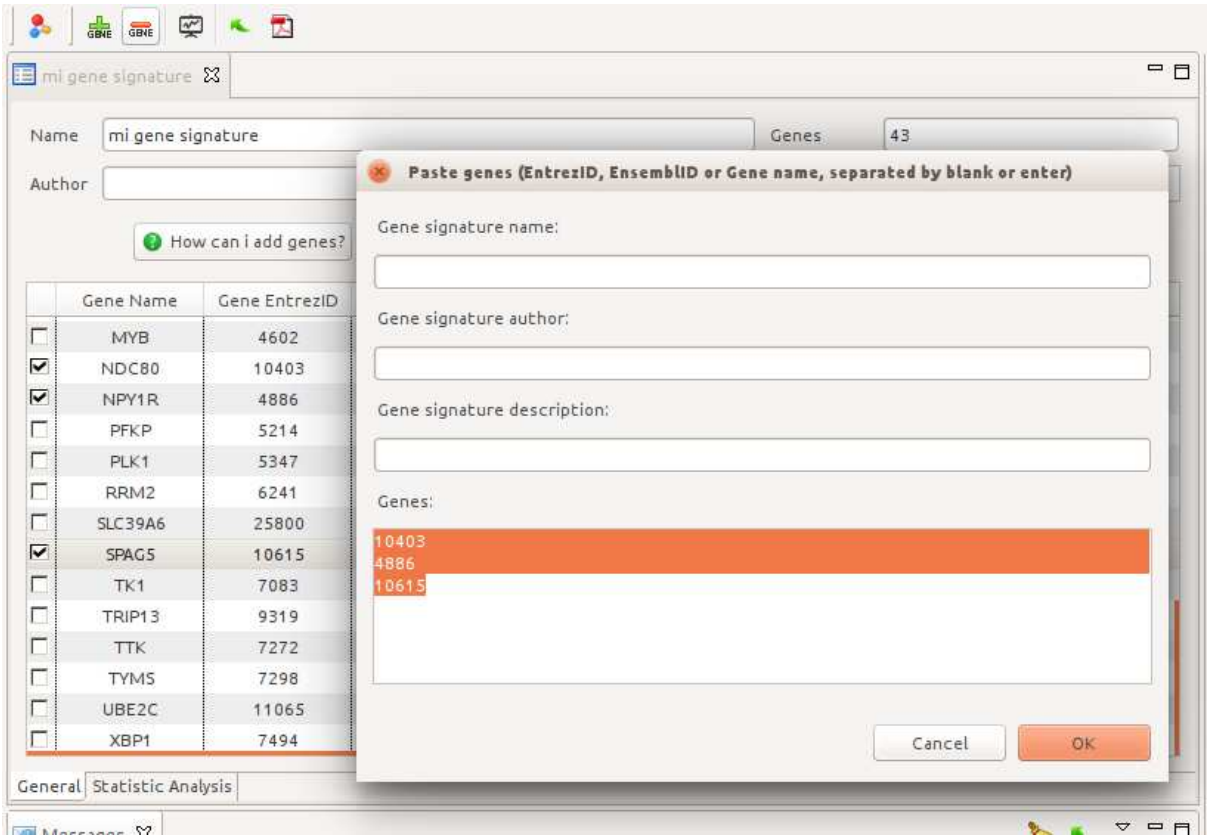
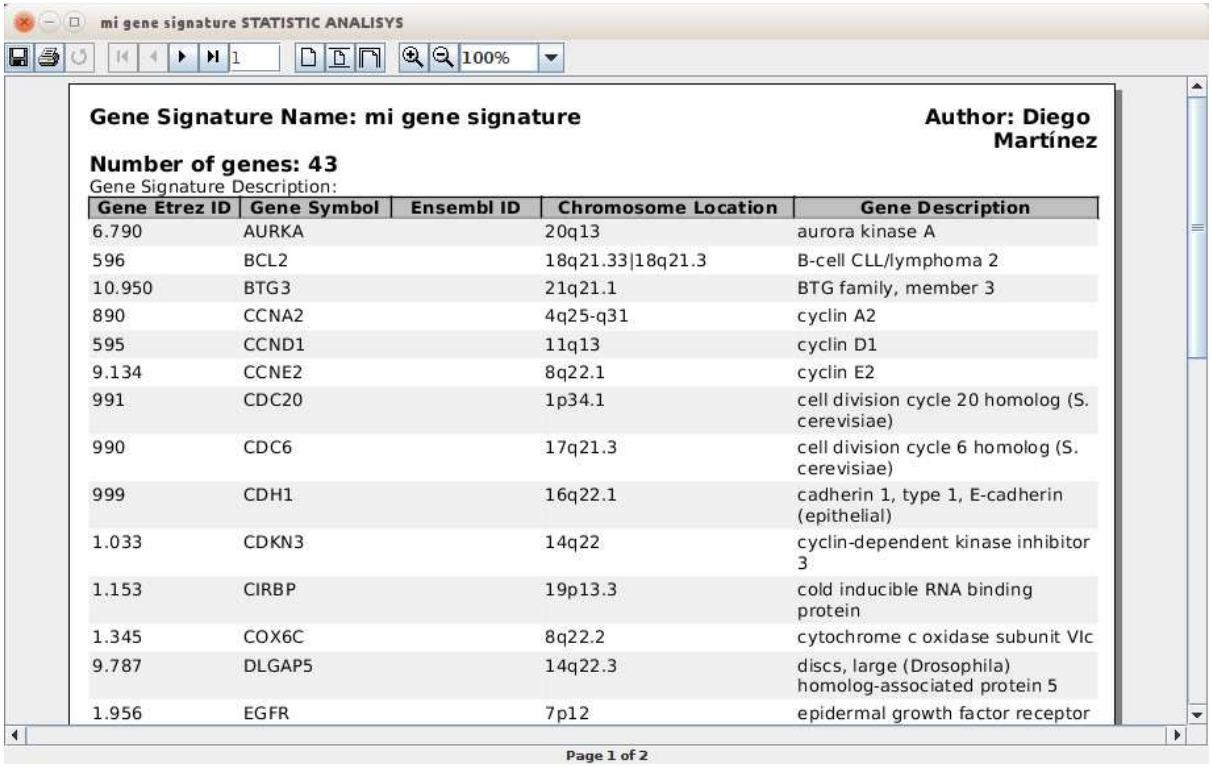


Figura 3.1-5 - Nuevo gene signature a partir de una selección

- Reportes: A partir de la información presente en el *gene signature*, se pueden obtener reportes imprimibles que incluyen toda la información básica de la entidad (autor, nombre, descripción) y la lista de genes en forma tabular. Además, se incluyen todos los análisis estadísticos realizados. En la Figura 3.1-6 vemos un ejemplo;

[10] http://amp.pharm.mssm.edu/Enrichr/



Gene Etrez ID	Gene Symbol	Ensembl ID	Chromosome Location	Gene Description
6.790	AURKA		20q13	aurora kinase A
596	BCL2		18q21.33|18q21.3	B-cell CLL/lymphoma 2
10.950	BTG3		21q21.1	BTG family, member 3
890	CCNA2		4q25-q31	cyclin A2
595	CCND1		11q13	cyclin D1
9.134	CCNE2		8q22.1	cyclin E2
991	CDC20		1p34.1	cell division cycle 20 homolog (S. cerevisiae)
990	CDC6		17q21.3	cell division cycle 6 homolog (S. cerevisiae)
999	CDH1		16q22.1	cadherin 1, type 1, E-cadherin (epithelial)
1.033	CDKN3		14q22	cyclin-dependent kinase inhibitor 3
1.153	CIRBP		19p13.3	cold inducible RNA binding protein
1.345	COX6C		8q22.2	cytochrome c oxidase subunit VIc
9.787	DLGAP5		14q22.3	discs, large (Drosophila) homolog-associated protein 5
1.956	EGFR		7p12	epidermal growth factor receptor

Figura 3.1-6 - Ejemplo reporte de gene signature

- Acción de apertura de vista de genes, para integración con otros sitios desde donde se obtiene información complementaria (ver sección Vista de genes)

3.1.2.4 Herramientas para generar/importar *gene signatures*

El usuario de la plataforma cuenta con distintos modos de confeccionar un *gene signature*, como se ve en la pantalla de presentación (Figura 3.1-1), sobre la columna del apartado izquierdo, y se detallan a continuación:

Creación de un *gene signature* vacío: Crea una entidad sin genes. La cual se completará según los estudios e hipótesis del usuario. El escenario de uso es el de un investigador que tiene la hipótesis que un conjunto de genes con los que viene trabajando tiene potencial valor pronóstico en el cáncer que estudia o también un investigador que leyó sobre un conjunto de genes en un artículo y quiere realizar validaciones estadísticas o quiere completarlo con genes de interés o quiere eliminar

ciertos genes que supone no aportan a su poder pronóstico;

Un *gene signature* importado de algunas de las bases de datos secundarias: Estas bases de datos están disponibles públicamente a través de internet. Las bases de datos con las que se integra Bioplat son GeneSigDB[11] y MolSigDB[12]. En la versión inicial se desarrolló un acceso via web services[13], pero luego por cuestiones de performance se decidió hacer una réplica de la información en la base de datos local de Bioplat, teniendo en cuenta que la actualización no es frecuente;

Generado a partir de un proceso heurístico. Es la creación de un *gene signature* producto de aplicar un algoritmo a un conjunto de otros *gene signatures* publicados que cumplen un criterio elegido por el investigador. A este *gene signatures* resultante lo llamaremos Metasignature. Si bien actualmente solo está implementado el algoritmo ALIX[14], la solución está desarrollada como punto de extensión para incorporar fácilmente nuevos algoritmos métodos de selección de genes;

Un *gene signature* generado aleatoriamente: El usuario elige una cantidad fija de genes y luego el sistema determina aleatoriamente cuales incluir. Este caso se usa principalmente para pruebas.

Una vez el usuario tiene la entidad cargada en el sistema, así como se indicó que se le pueden realizar operaciones, también puede ser editado en sus datos más simples, como ser el nombre, autor, descripción, etc.

Se destaca que una vez que el usuario tiene un *gene signature* importado en la plataforma, este es tratado en general sin importar su procedencia, aun así puede ser útil tener alguna operación particular para *metasignatures,* por ejemplo.

Por último, es importante hacer énfasis en las operaciones de optimización y validación. Las cuales representan un punto a explotar muy importante dentro de la

[11] http://www.genesigdb.org/genesigdb/
[12] http://software.broadinstitute.org/gsea/msigdb
[13] Operaciones publicadas: http://genesigdb.org/genesigdb/restschema.jsp
[14] Abreviatura de *''At Least In X gene signatures''*. Es una función que dado un conjunto de *gene signatures*, retorna los genes que cumplan la condición de estar presente en al menos X, de los gene signatures del conjunto de entrada.

plataforma. Es de notar que, tanto las validaciones, como los procesos de optimización de *gene signatures* son, intrínsecamente, complejos y de alto coste computacional, así toman vital importancia en el desarrollo del cliente, con lo cual volveremos más detalladamente sobre estas operaciones en secciones subsiguientes.

3.1.3 Experimentos Bioplat

3.1.3.1 Definición

Un experimento es una agrupación de muestras para un determinado tipo de tumor. Las muestras se extraen de tejidos de organismos con afección tumoral. Cada una de estas muestras tiene datos de expresión de los genes (podría contener el Transcriptoma completo) y datos clínicos que incluyen datos de supervivencia como la sobrevida o tiempo transcurrido hasta la reaparición del tumor (se los referencia como datos de progresión o *follow-up*).

Los Experimentos Bioplat, sirven para realizar validaciones estadísticas de supervivencia, para comprobar el valor pronóstico/predictivo de los *genes signatures* (ver sección Validación estadística de *gene signatures*) y para optimización de los mismos (ver sección Optimización de *gene signatures*).

3.1.3.2 Visualización

La Figura 3.1-7, muestra un experimento visualizado en su editor.

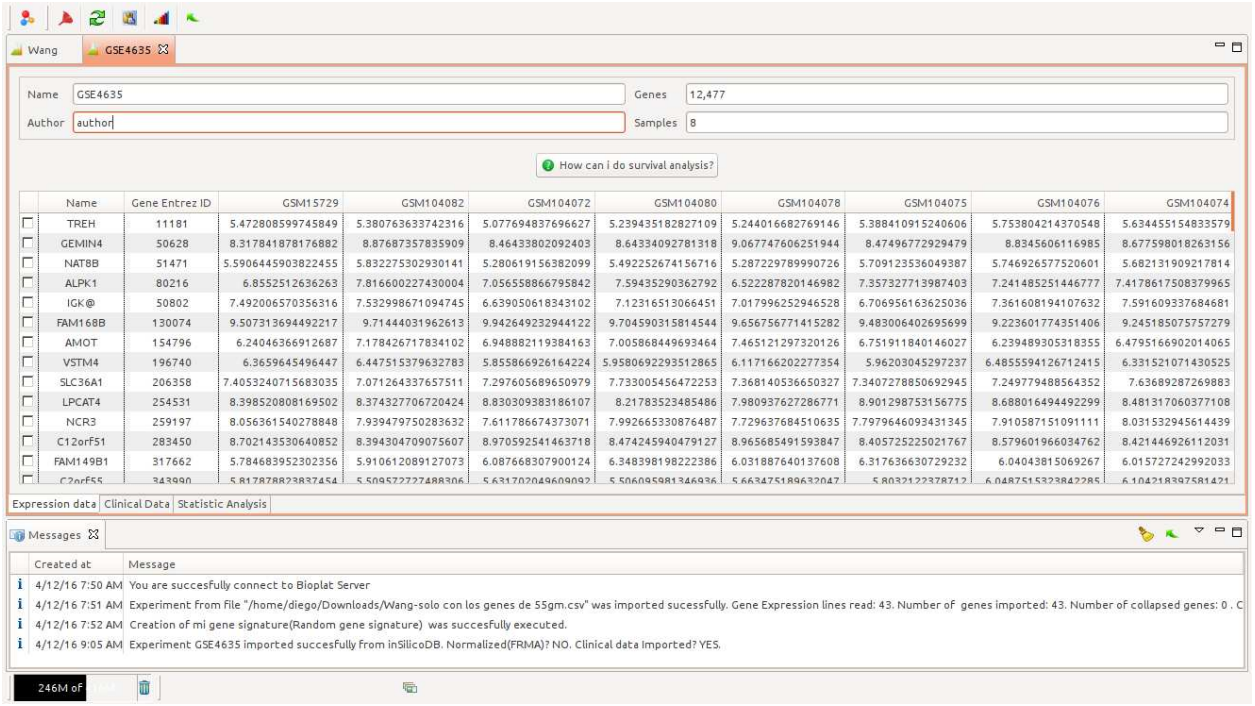


Figura 3.1-7 - Visualización de un experimento Bioplat

En dicho editor, se advierten distintos conjuntos de datos. Esta información está organizada lógicamente, en distintos apartados según su significado o relación de uso e interpretación.

En la solapa *''Expression data''* se muestra una grilla con las siguientes columnas de información. *Name* y *EntrezId*: son, respectivamente, el nombre simbólico del gen analizado en el experimento y el id ''normalizado'' por la plataforma. Las columnas restantes, tituladas GSMXXX, representan a las distintas muestras, o *samples*, presentes en el estudio. La celda (gen, muestra) indica en la grilla el nivel de expresión del gen en el tejido donde se tomó la muestra. Según el dogma central de la biología molecular, un gen se transcribe a ARNm y luego es traducido a proteína, la cual es quien cumplirá la función biológica que le compete. Así, el nivel de expresión génica mide cuánto se transcribió el gen en la muestra. Esta es una medida de utilidad para encontrar similitudes y diferencias entre distintos casos/pacientes, con el objetivo de encontrar hipótesis biológicas de interés.

En la siguiente solapa se presentan los datos clínicos, estos datos representan características observables para cada muestra, como por ejemplo edad, sexo del paciente,

condición de fumador, entre otras variables. Como se mencionó anteriormente, los datos clínicos más destacados son los datos de supervivencia.

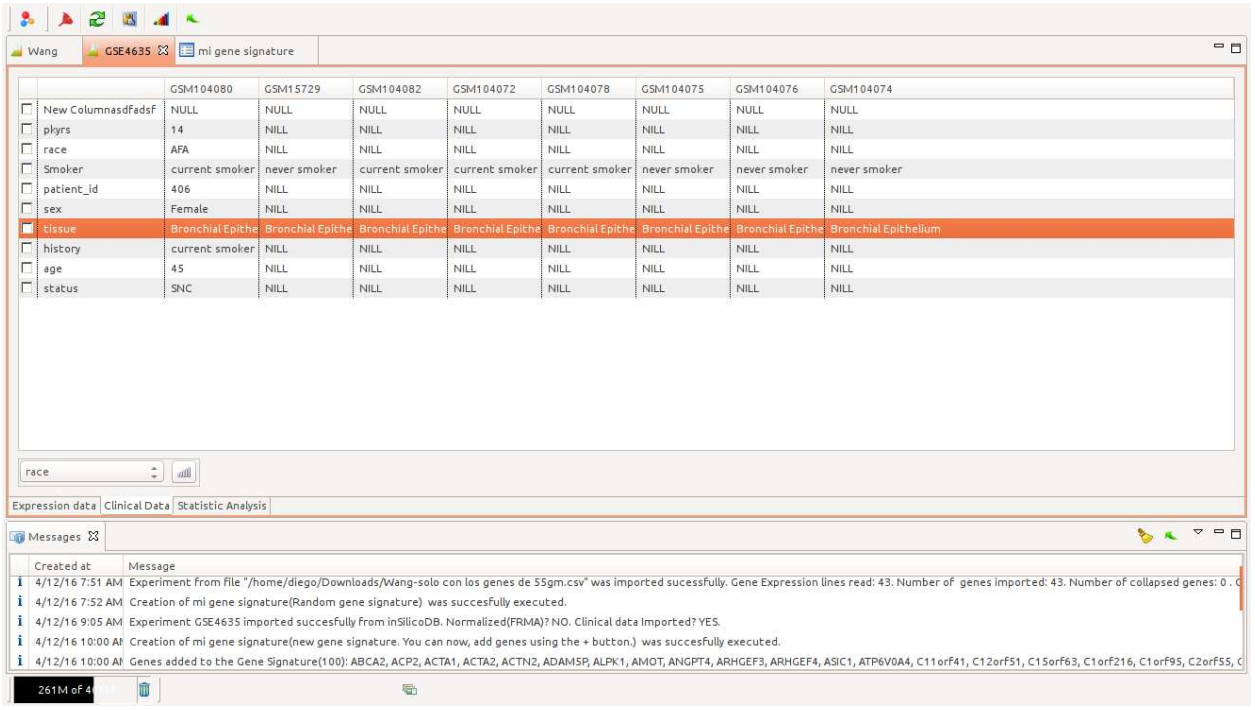


Figura 3.1-8 - Visualización de datos clínicos en experimentos Bioplat

La tercera y última solapa para experimentos es la ''*Statistic Analysis*'', donde se visualizarán análisis realizados sobre el experimento. Estos análisis están relacionados con una operatoria que se verá más adelante, llamada ''Análisis estadístico con *cluster* manual'', dentro de la sección Operaciones básicas.

3.1.3.3 Operaciones básicas

Una vez el usuario tiene el experimento importado en la plataforma, el usuario puede aplicarle operaciones como se presenta en la Figura 3.1-9. Las operaciones más importantes están disponibles también como botones desde el *toolbar*.

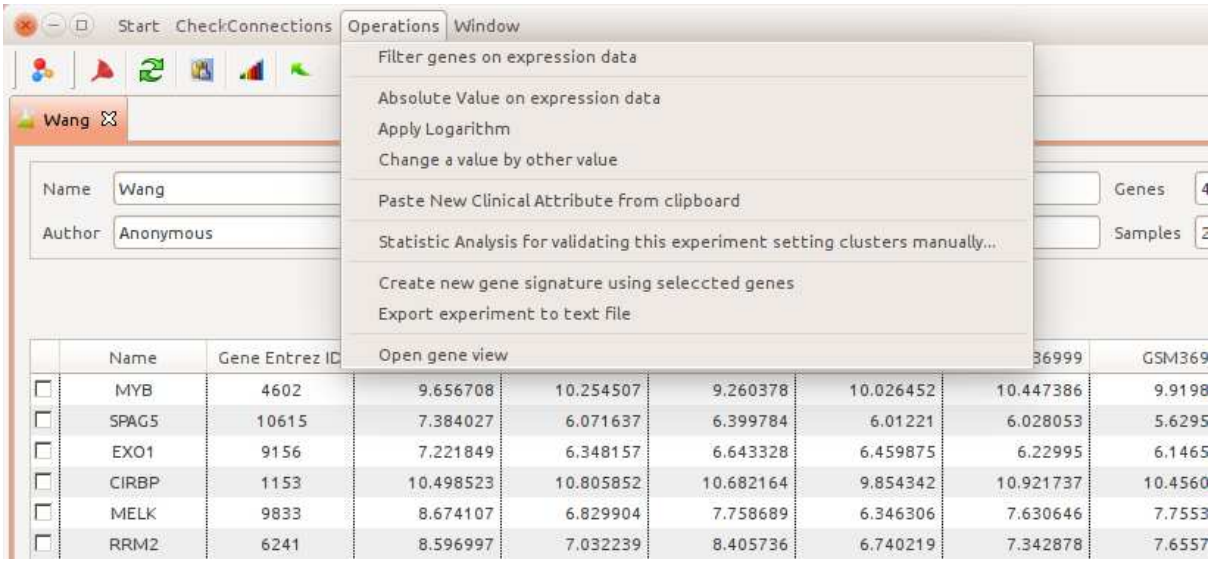


Figura 3.1-9 - Operaciones básicas para experimentos Bioplat

Las operaciones disponibles son:

- Filtro de genes. Esta función permite reducir la lista de genes del experimento. Para ello el usuario puede escribir/pegar la lista de genes utilizando cualquiera de los estándares de ID soportados por la plataforma Bioplat;

- Funciones normalizadoras de expresiones génicas. Actualmente se proveen las funciones Valor absoluto y Logaritmo que desplazan los valores todas las expresiones, reemplazando de un valor específico por el resultado de aplicarle la función;

- Cambiar valores específicos. El usuario desea modificar las ocurrencias de un valor presente en el experimento por otro, a modo de ajuste. Esta función se puede utilizar tanto para valores de expresión génica como de datos clínicos;

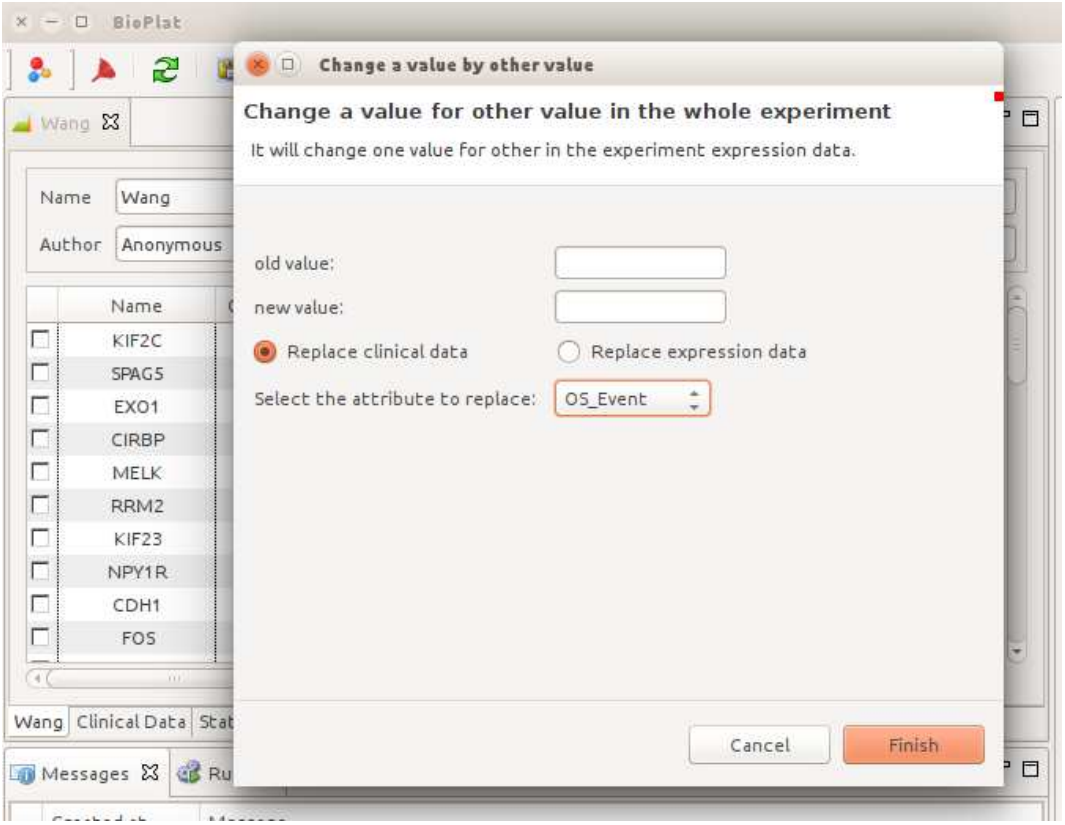


Figura 3.1-10 - Wizard de ajuste de valores

- Pegado de nuevos datos clínicos desde el *portapapeles* del sistema. Es posible agregar datos clínicos o *follow-up* a un experimento, extraídos de otras herramientas o archivos. El texto a procesar debe respetar algunos de los formatos que soporta la plataforma, por ejemplo datos en formato CSV;

- Análisis estadístico con cluster manual. Como se verá en la sección Validación estadística de *gene signatures*, el primer paso de la validación es realizar un *cluster* (agrupamiento) de los casos según los valores de expresión génica del experimento Bioplat con el que se valida, considerando solo los genes postulados por el *gene signature*. Sin embargo, el sistema ofrece una forma de validación de experimentos sin participación de *gene signatures*, útil solamente cuando el *cluster* de los casos lo quiere realizar manualmente el usuario. Entonces el primer paso de la validación lo

hará manual el usuario, y no el sistema como en el caso de la validación de *gene signatures*. Luego se medirá la correlación entre los *clusters* configurados y los datos de supervivencia correspondientes. Es de notar que en este caso no son utilizados los datos de expresión génica.

Esta opción es de utilidad solo cuando el usuario conoce de antemano los *clusters* y solo quiere usar Bioplat para medir la correlación con sus datos de supervivencia. Si el usuario quiere realizar la validación completa, comenzando por el *cluster* según los niveles de expresión de todos los genes del experimento, deberá primero crear un *gene signature* vacío y pegar todos los genes de dicho experimento, para luego realizar la validación estadística de *gene signatures*.

Los resultados (estadísticas y gráficos) que se muestran son los mismos que en el caso de la validación estadística utilizando gene signatures (ver sección Validación estadística de *gene signatures*).
La Figura 3.1-11 muestra la pantalla de configuración de la validación con *cluster* manual. Y la Figura 3.1-12 muestra la pantalla que permite realizar el *cluster* manual;

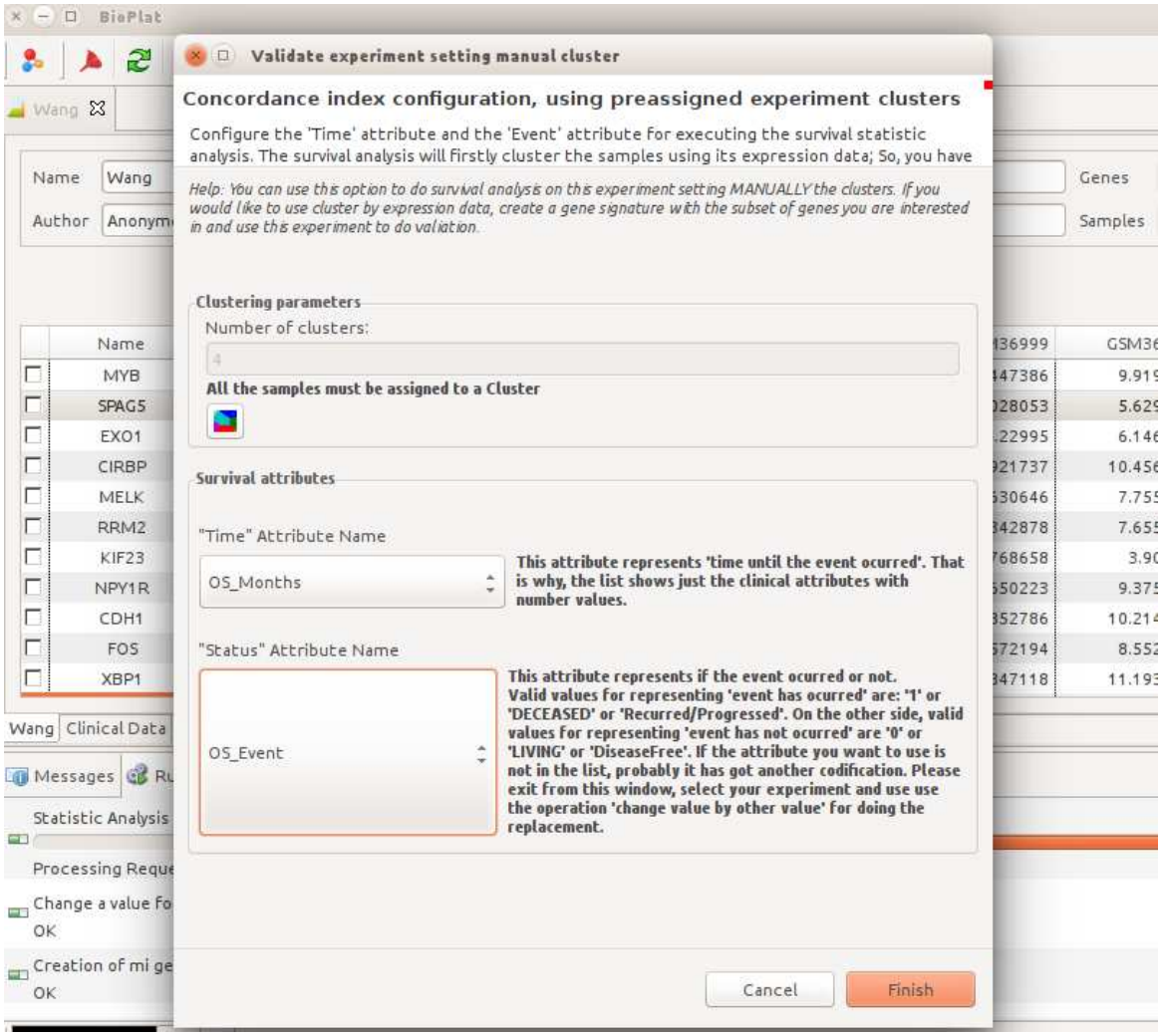


Figura 3.1-11 - Configuración de experimento Bioplat con cluster manual

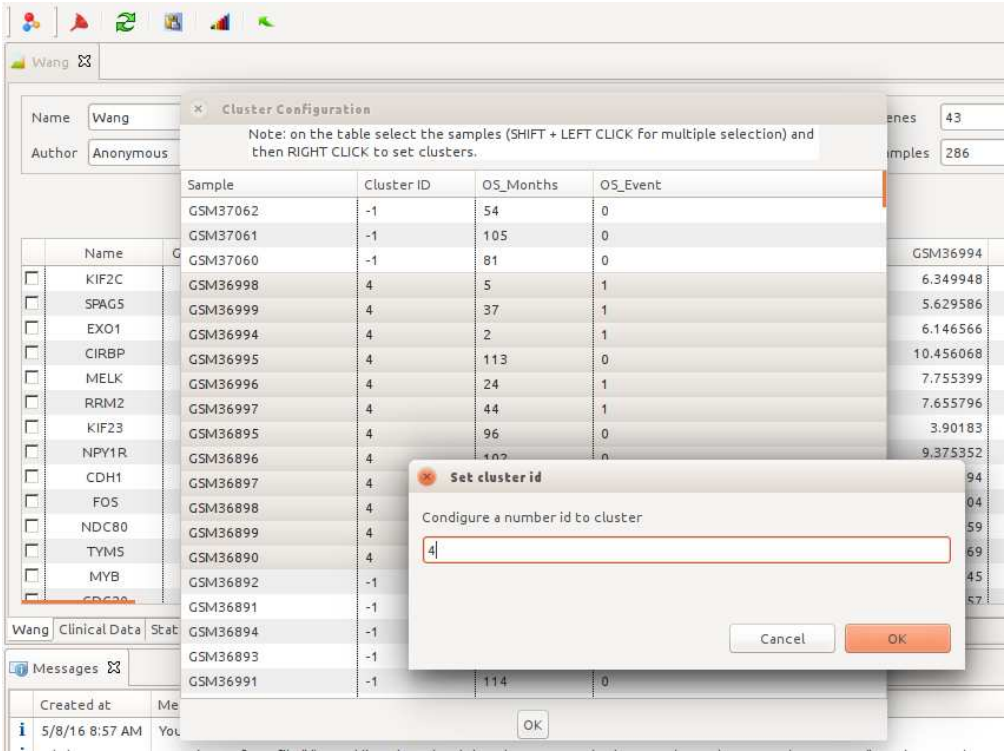


Figura 3.1-12 - Configuración de cluster manual para experimentos Bioplat

- Creación de *gene signature*: Se comparte la operación de creación de un nuevo *gene signature* a partir de genes seleccionados en el experimento. Siendo este uno de los casos de reutilización de operaciones entre distintas entidades de la plataforma;

- Exportación a un archivo. El usuario tiene la posibilidad de guardar el experimento modificado en un archivo, para su posterior utilización en la plataforma u otra herramienta que interprete el formato del mismo;

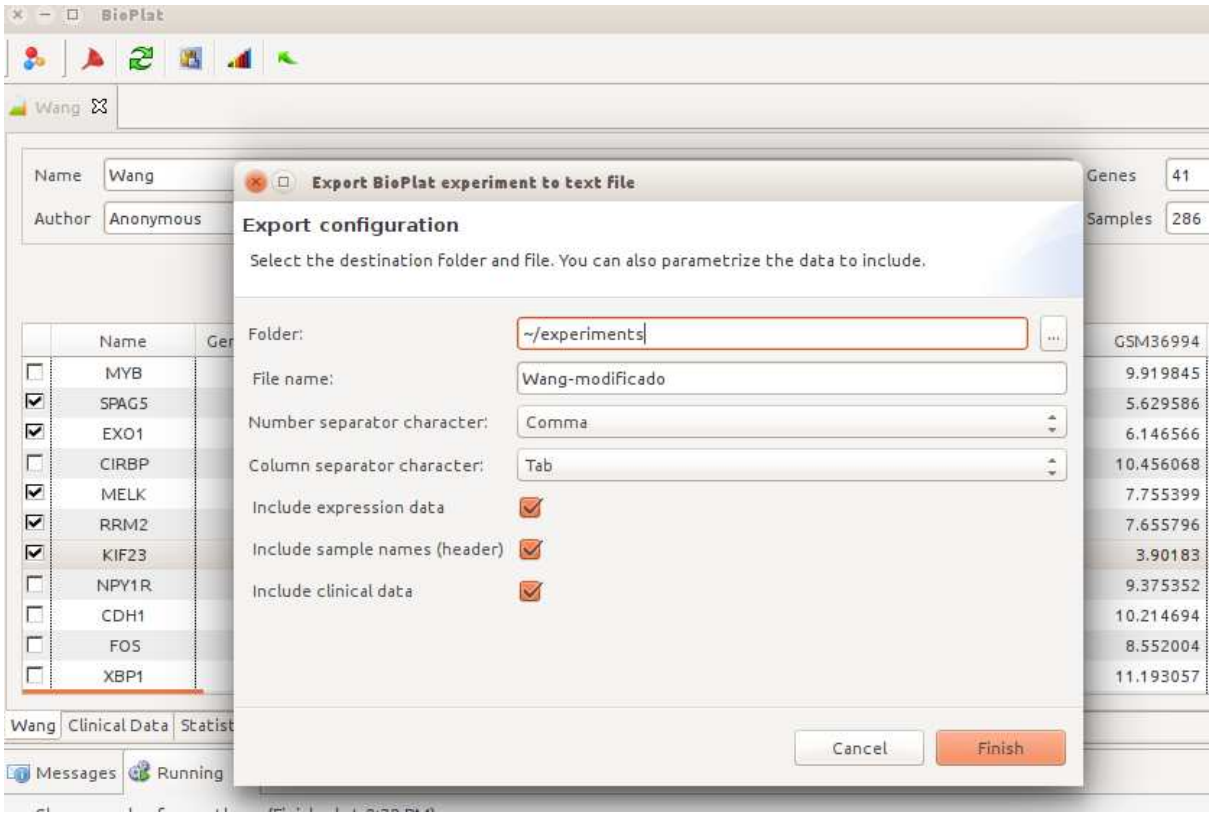


Figura 3.1-13 - Exportación de experimento a archivo

3.1.3.4 Herramientas para generar/importar experimentos Bioplat

De la misma manera que para *gene signatures*, existen distintas estrategias de creación de experimentos, a través de su importación a la plataforma desde distintas fuentes. Actualmente la plataforma brinda las siguientes opciones:

Importación de Experimentos TCGA (The Cancer Genome Atlas) vía CBioPortal[15]. TCGA es el banco de datos más grande y utilizado de cáncer. A través de un conjunto de pasos se permite que el usuario vaya configurando los parámetros para la importación. Esta base de datos está en constante actualización, agregando los estudios de mayor

[15] http://www.cbioportal.org/

relevancia para la comunidad científica del cáncer. Provee información de diferentes tipos de cáncer en diferentes localizaciones. Considerando como requerimiento central, ofrecer al investigador la posibilidad de disponer en todo momento de la información actualizada de TCGA (por la importancia que tiene esta herramienta en Bioplat), se decidió evitar la replicación de la información en nuestra BD e implementar una integración *online* con CBioportal, en el momento que el usuario decide realizar la importación. La integración se hace a través de una API R ofrecida por CBioportal. Debido a la importancia de esta herramienta, se describirá cada uno de los pasos para la importación.

En el paso 1 tenemos un listado de los estudios presentes en la base de datos de cBioPortal. El usuario busca y selecciona el que sea de su interés. Esto se ilustra en la Figura 3.1-14;

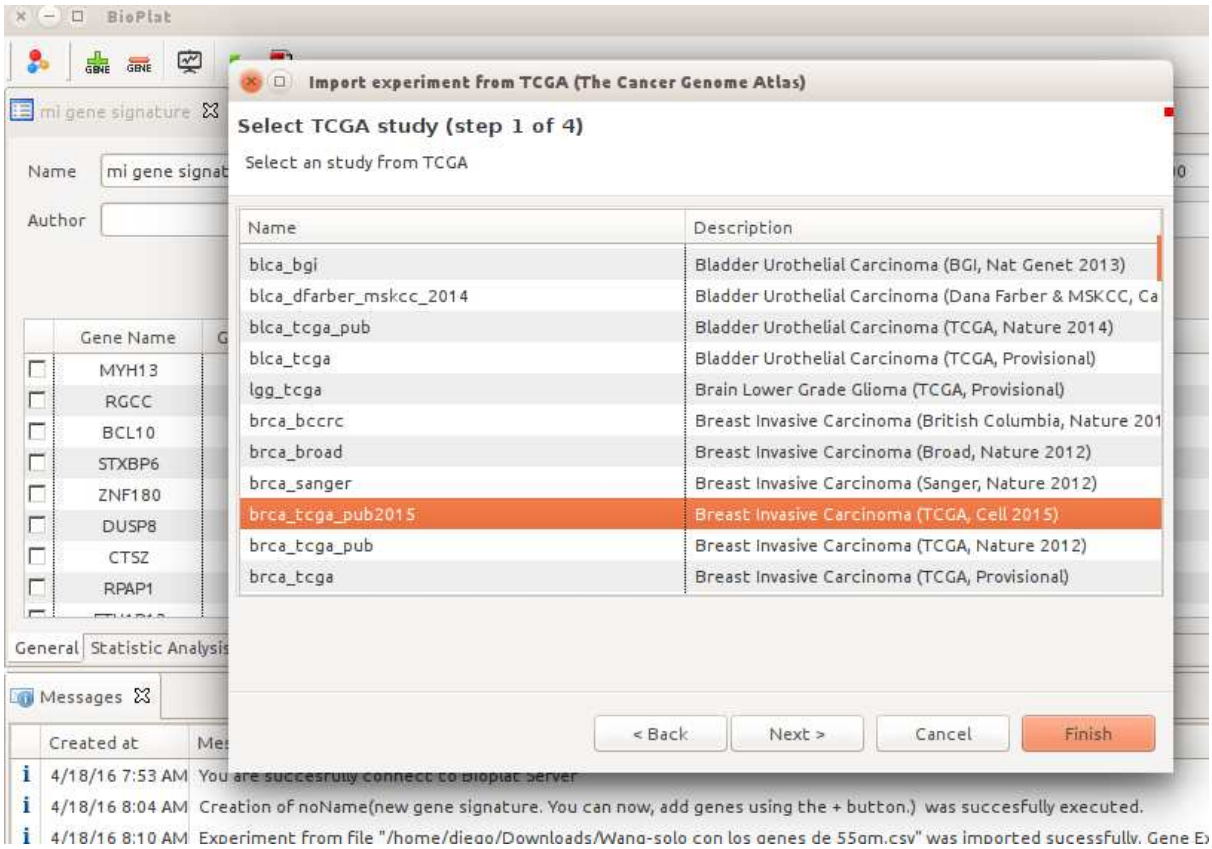


Figura 3.1-14 - Importación de experimentos desde TCGA. Primer paso

A partir de un estudio, CBioportal arma subconjuntos de casos con características comunes. A estos subconjuntos los llama *''Case List Name''*. Por ejemplo, el *Case list ''All complete tumors''* agrupa solo los casos que tengan toda la información completa. Como muestra la Figura 3.1-15, el usuario en el paso 2 deberá seleccionar qué *''case list name''*

desea importar. Luego, deberá elegir qué información biológica desea importar. La más utilizada suele ser: *mRNA expression (RNA Seq V2 RSEM)* que representa los datos de expresión génica curados.

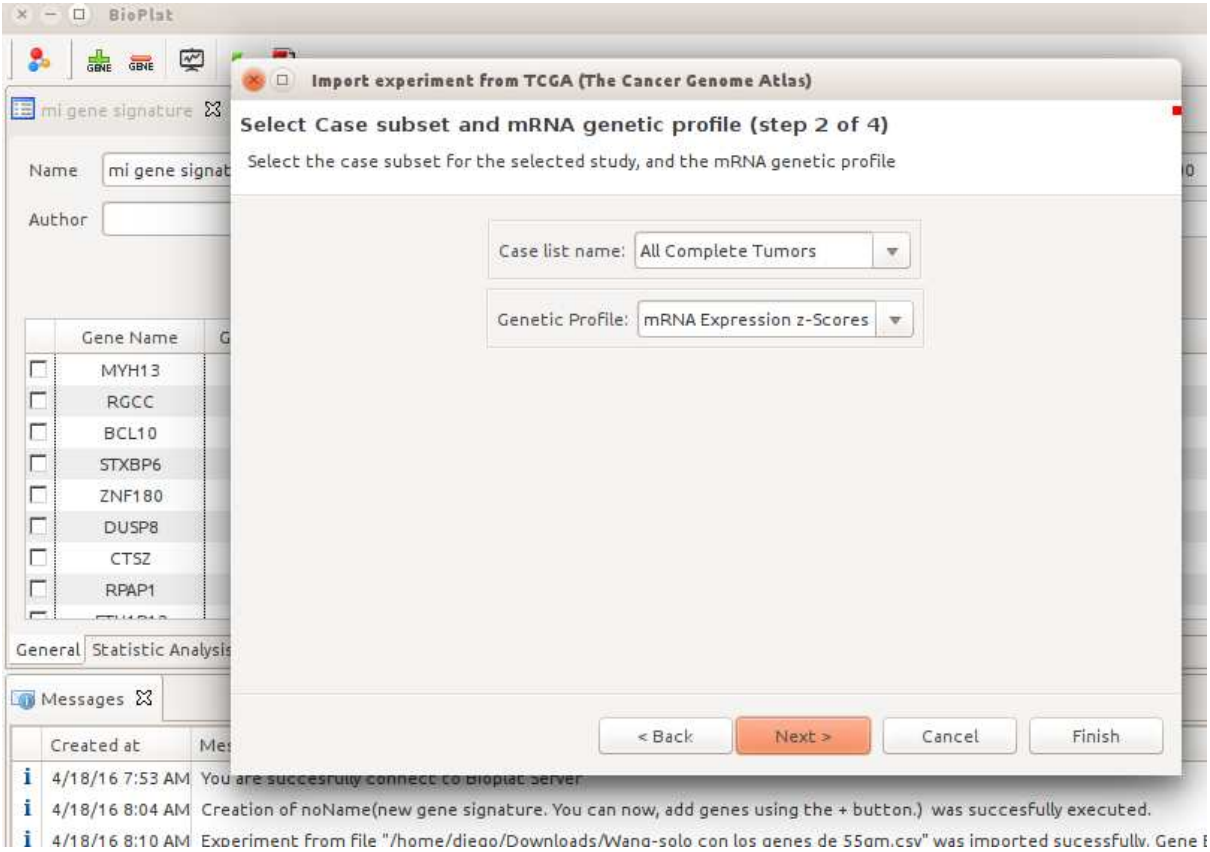


Figura 3.1-15 - Importación de experimentos desde TCGA. Segundo paso

A continuación el usuario debe especificar los genes para los que desea importar información. Para ello, el importador ofrece dos herramientas: la selección de los genes de alguno de los *genes signatures* disponibles en el escritorio Bioplat (previamente generado/importado. Ver sección correspondiente) o suministrar la lista de genes. Ilustramos este paso en la Figura 3.1-16

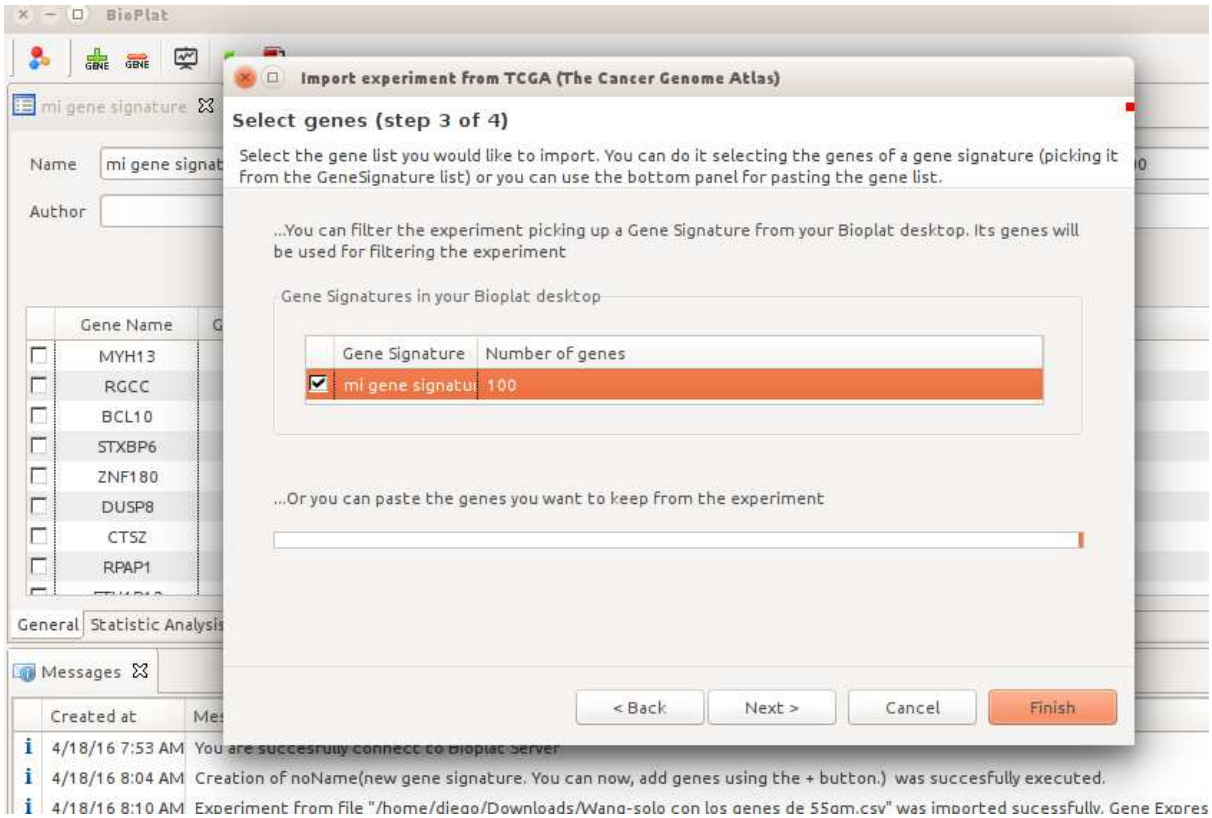


Figura 3.1-16 - Importación de experimentos desde TCGA. Tercer paso

Como último paso, es posible seleccionar qué datos clínicos se quieren importar desde el experimento cBioPortal. Se ilustra este aspecto en la Figura 3.1-17. Este paso es opcional, el cliente habilita su finalización en el paso anterior, siendo por omisión, importados todos los datos disponibles en el experimento;

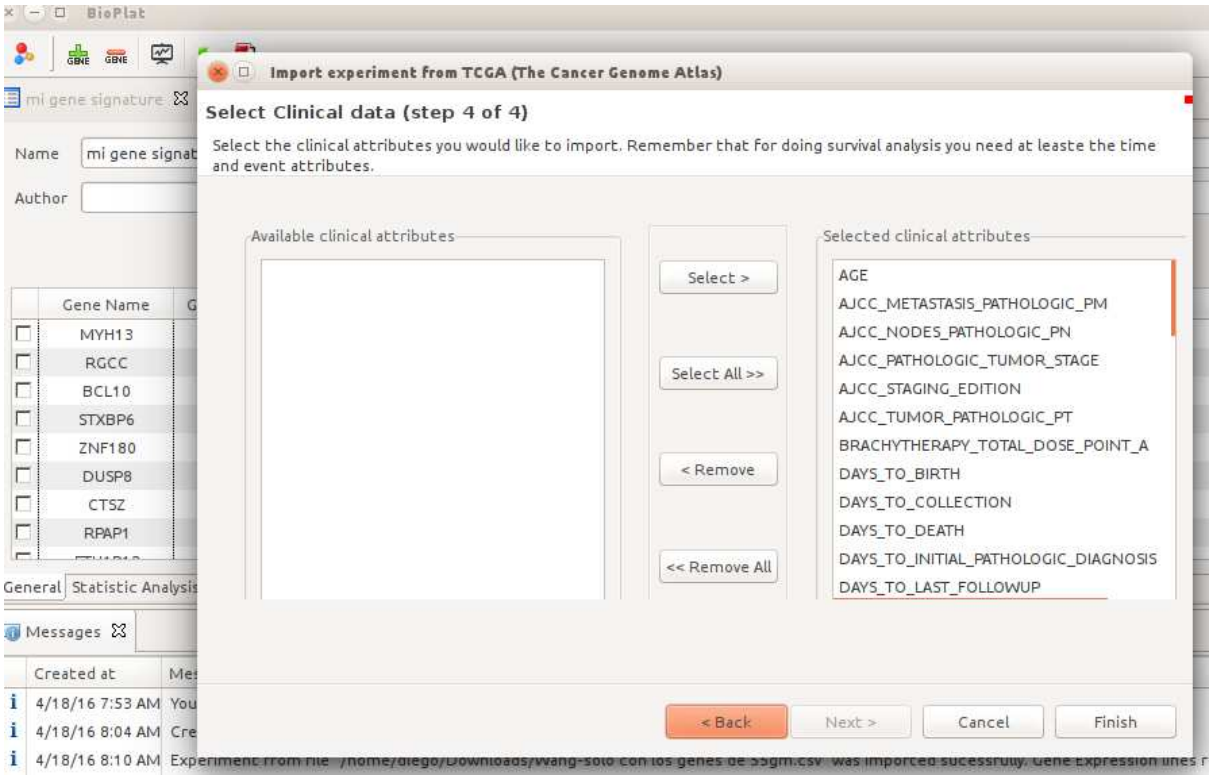


Figura 3.1-17 - Importación de experimentos desde TCGA. Cuarto paso

Importación de experimento desde un archivo. El usuario descarga un archivo manualmente desde algún sitio público o generándolo con datos propios. Dicho archivo debe respetar el siguiente formato: las primeras líneas deben ser para datos clínicos, luego viene una línea para *header*, finalmente vienen los datos de expresión (ver archivo a la derecha en la Figura 3.1-18)

La configuración que debe realizar el usuario, presentada en pantalla es:

- *Path* de ubicación del archivo;
- Estrategia para colapsar el archivo. En ciertas tecnologías utilizadas para medición de los niveles de expresión génica, los resultados pueden contener el mismo sector del

genoma más de una vez secuenciado; lo cual se traduce en que el ID del gene puede estar repetido en la matriz del archivo original. El experimento Bioplat exige eliminar esta duplicación (por un requerimiento funcional). Para ello ofrece estrategias que permitan seleccionar de entre varias filas con ID repetido, solo una de ellas. Las opciones disponibles son: la fila que tenga mayor varianza, la fila que tenga mayor media, la fila que tenga mayor mediana.

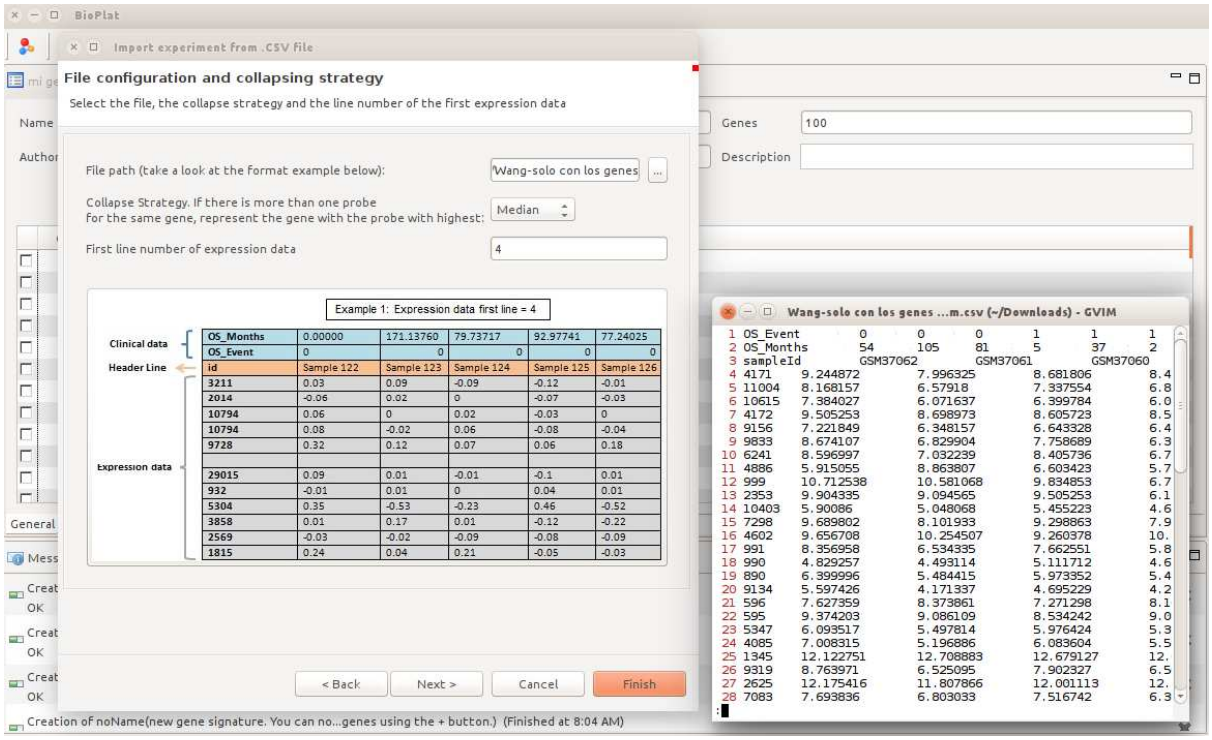


Figura 3.1-18 - Importación de experimentos desde archivo

Experimento InSilicoDB usando GSE. InSilicoDB[16] es un repositorio de datos genómicos curados, muy utilizado en la comunidad científica. Pertenece a una empresa Belga. Con la que luego de un trabajo mancomunado de ajuste en los servicios (técnicamente Web Services) que ofrecen, se consiguió un acceso a la información de una manera conveniente para la integración con Bioplat.

El usuario debe indicar el GSE (que es el identificar de estudios estandarizado por GEO[17]) y si quiere o no importar los datos clínicos. Además, debe especificar qué plataforma se utilizó para la secuenciación y si quiere bajar los datos de expresión génica normalizados o no. Toda esta configuración está ilustrado en la captura de pantalla de la Figura 3.1-19

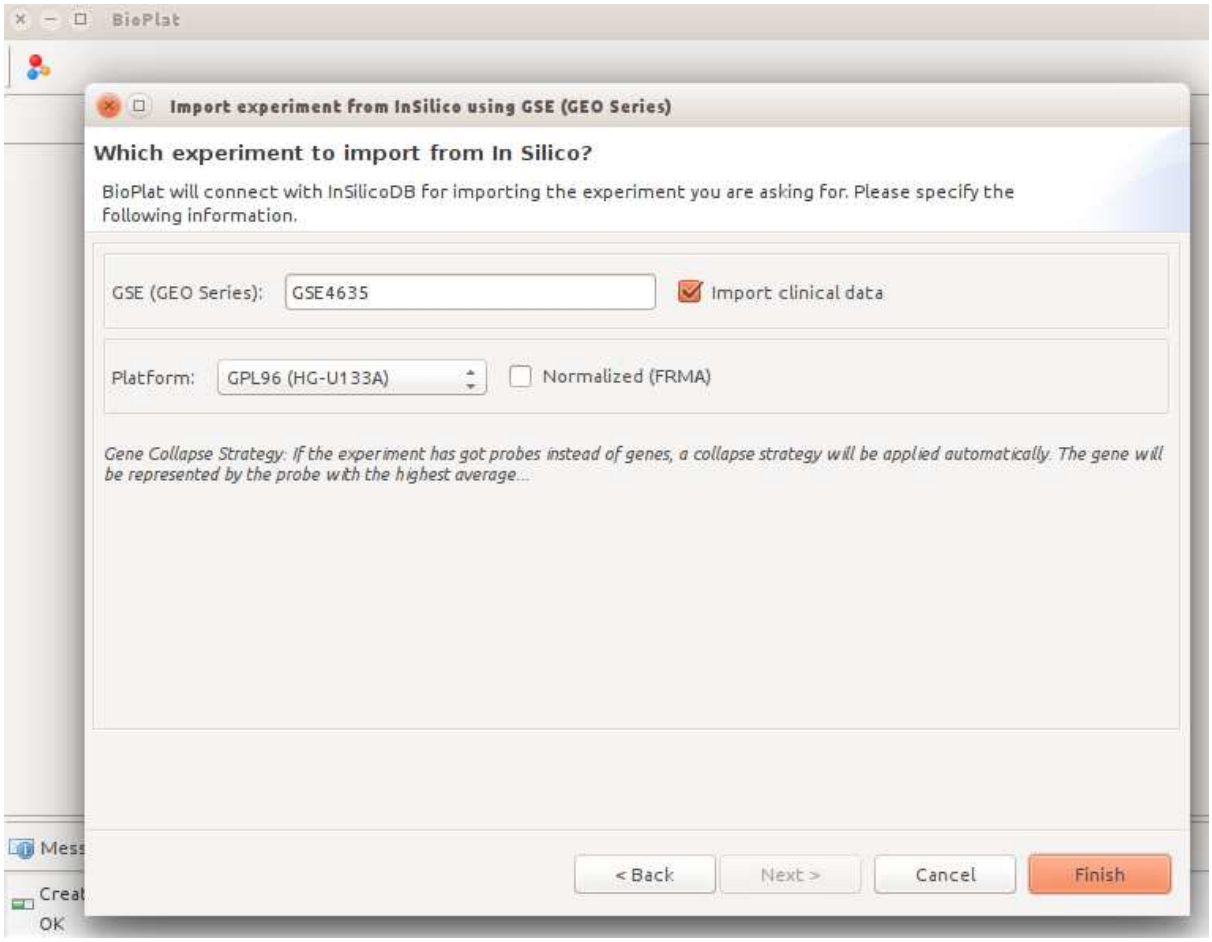


Figura 3.1-19 - Importación de experimentos Bioplat desde InsilicoDB

[16] https://insilicodb.com/
[17] http://www.ncbi.nlm.nih.gov/geo/

- Desde archivos Cel (AffyMetrix[18]). Los archivos Cel definen un formato estandarizado que generan los chips de esta empresa que miden perfiles de expresión génica;

Es importante destacar que si bien hay distintas fuentes desde donde importar los experimentos, la forma de visualización está estandarizada tal como muestra la sección Visualización de esta entidad, antes mencionada. Esto facilita el uso de las diferentes herramientas por parte del usuario, abstrayéndose del origen del mismo. De manera análoga ocurre con los *genes signatures*.

[18] http://www.affymetrix.com/

3.1.4 Validación estadística de *gene signatures*

En esta sección profundizaremos en una de las operaciones más importantes de la plataforma, el análisis estadístico de un *gene signature*, el cual constituye la comprobación experimental de los *genes signatures* en los que un usuario esté trabajando. Como iremos viendo la validación en sí, ofrece grandes posibilidades de explotación.

Ahora bien, el usuario teniendo su *gene signature* en estudio y un conjunto de experimentos Bioplat, puede empezar a operar con ellos combinándolos, es decir poner a prueba su hipótesis (evaluar el valor pronóstico de la lista de genes postulada). Si bien la plataforma provee gran variedad de método estadísticos y gráficos, el análisis estadístico es un punto de extensión más de la plataforma; es muy simple incorporar nuevos métodos estadísticos o gráficos. Todos los resultados presentados le sirven al usuario para evaluar el poder pronóstico/predictivo de su *gene signature* haciendo la valoración estadística correspondiente, transformando así aquellos datos en información concreta para el avance en sus conclusiones.

Hay dos aspectos importantes para mostrar respecto al análisis estadístico: las opciones de configuración y los resultados presentados. A continuación se indican los pasos de una validación puntual a modo ejemplo. Detallando aspectos importantes en los distintos pasos de la configuración.

Teniendo el *gene signature* a evaluar cargado en la plataforma, el primer paso es seleccionar los distintos experimentos Bioplat importados previamente en la plataforma (ver sección Herramientas para generar/importar experimentos Bioplat), que se utilizarán para realizar el análisis estadístico, como se observa en la Figura 3.1-20, que presenta un resumen del modo en que los experimentos se aplicarán. Al presionar el botón de agregado, rotulado con el signo ''+'', comienza la configuración a aplicar. En primera instancia se seleccionarán los experimentos cargados en el cliente, como muestra la Figura 3.1-21. Luego, como paso siguiente, se debe especificar la configuración para realizar el *clustering* de los pacientes, según sus niveles de expresión génica. Entre los parámetros a configurar se incluyen los siguientes (ver Figura 3.1-22):

- *Clustering parameters*: Cantidad de clusters. Se puede usar la sintaxis de rango con un punto seguido de otro, por ejemplo ''2..4''. Cabe aclarar que por cada valor de *cluster*, se genera una configuración distinta. En el ejemplo anterior, si se eligió un solo experimento, se obtendrán 3 resultados diferentes. Uno con 2 clusters, otro con 3 y por último uno con 4;

- *Survival attributes*: Como en todo análisis estadístico de supervivencia, se necesita un atributo ''*status*'' que indica si ocurrió o no el evento y otro atributo time que representa cuanto tiempo pasó hasta que el evento ocurrió. O si no ocurrió, cuando tiempo pasó hasta la última vez que se supo del caso.

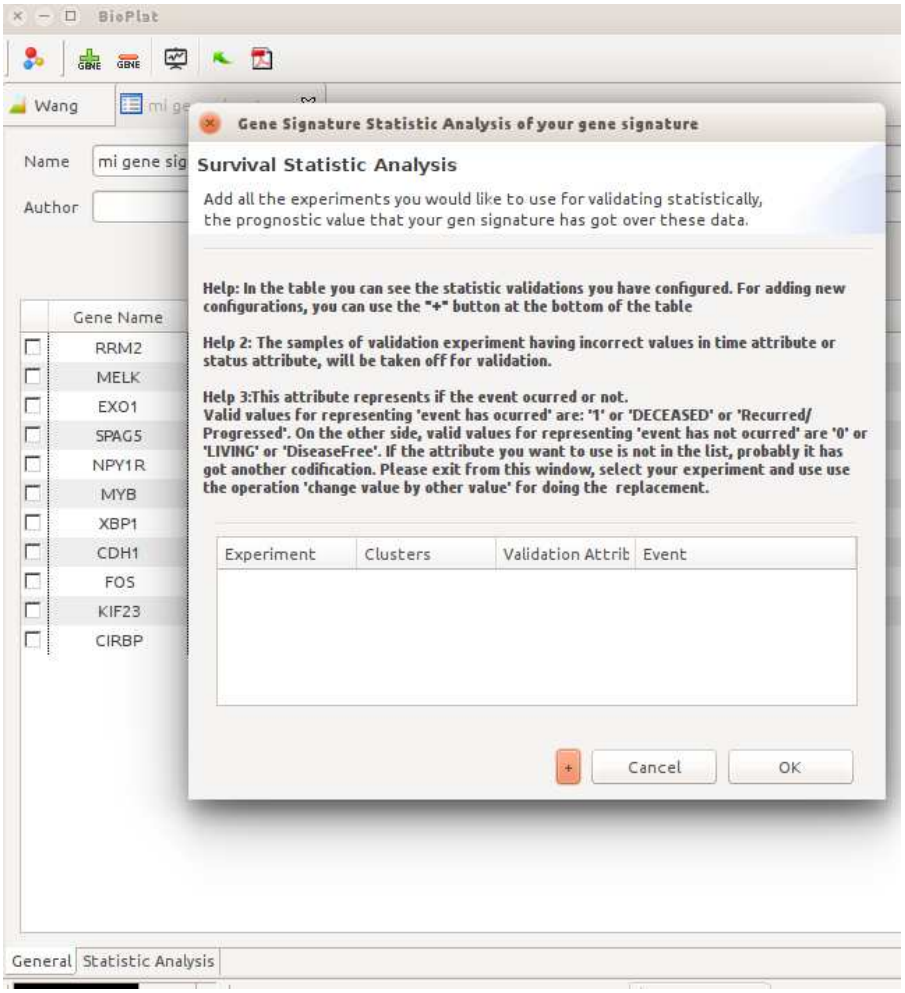


Figura 3.1-20 - Configuración de experimentos a aplicar

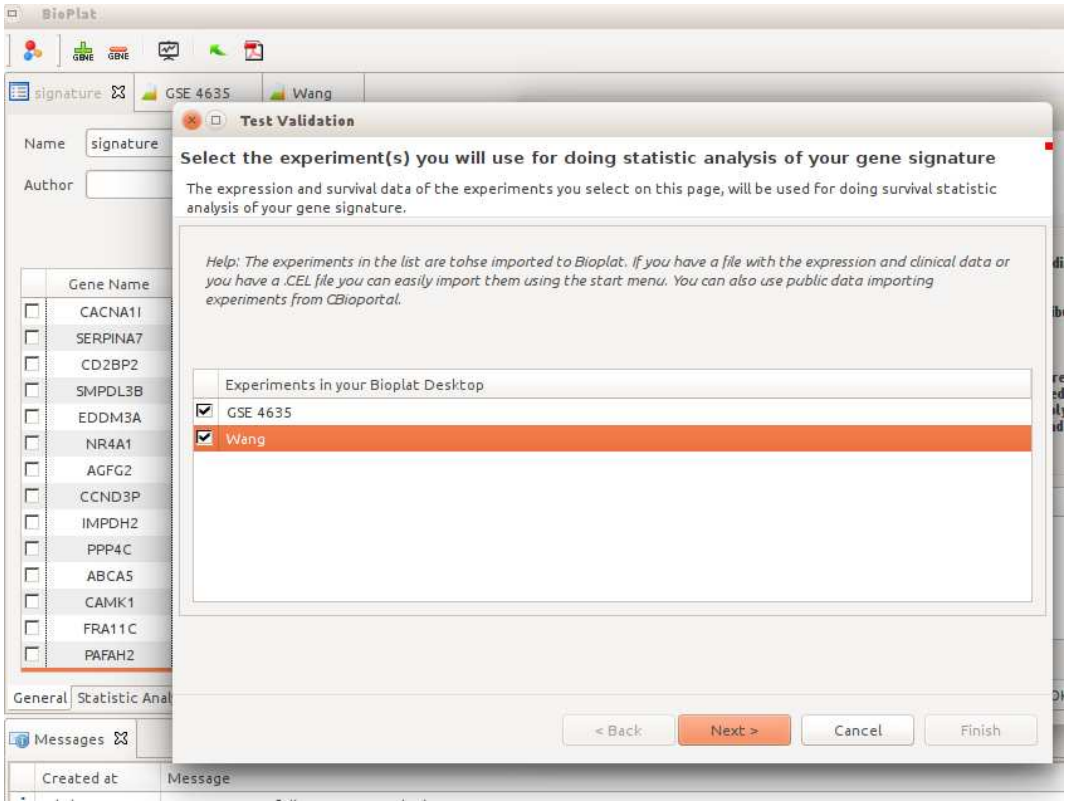


Figura 3.1-21 - Selección de experimentos

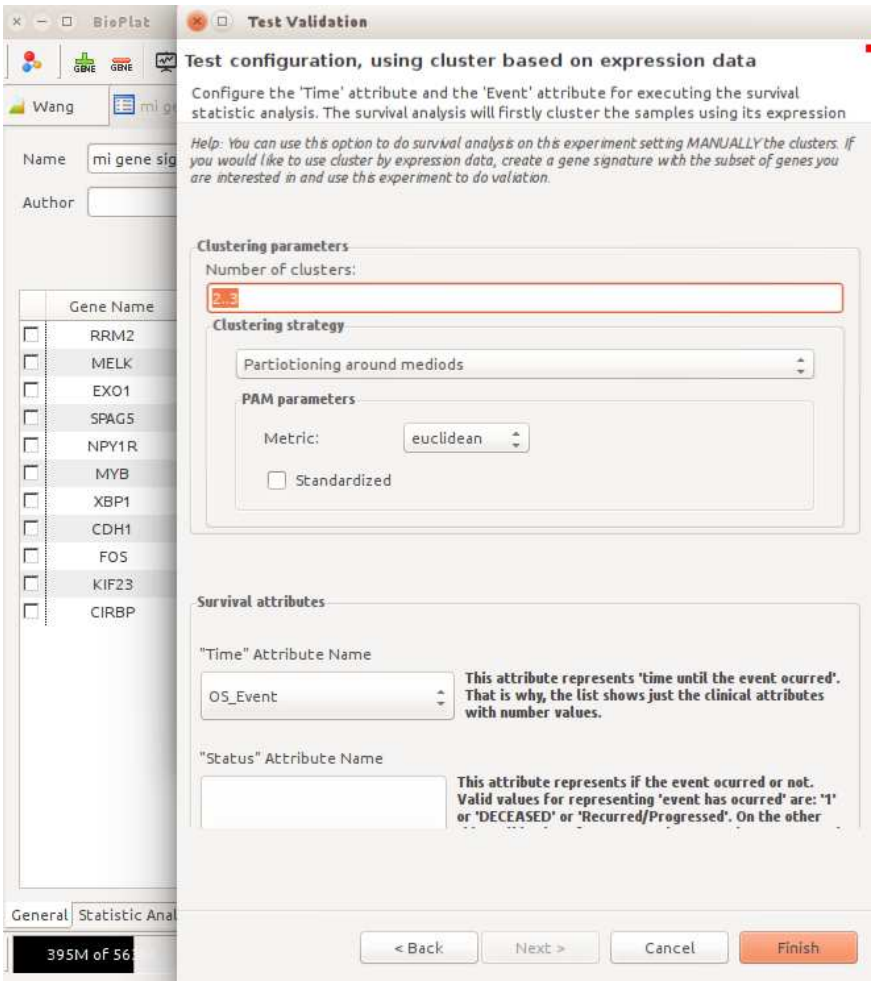


Figura 3.1-22 - Configuración de parámetros para los experimetos Bioplat a aplicar

Una vez concluida la configuración, se dispara la correspondiente validación. Esta validación al finalizar presentará los resultados en el editor del *gene signature* en la solapa de análisis estadísticos como vemos en la Figura 3.1-23. Por cada configuración (dada por un experimento y una cantidad de *clusters*) se presentará una fila nueva en la grilla de resultados estadísticos como se muestra en la misma figura. Cada una de estas filas muestra algunos resultados y botones para acceder a gráficos y otras funcionalidades relacionadas con dicha validación.

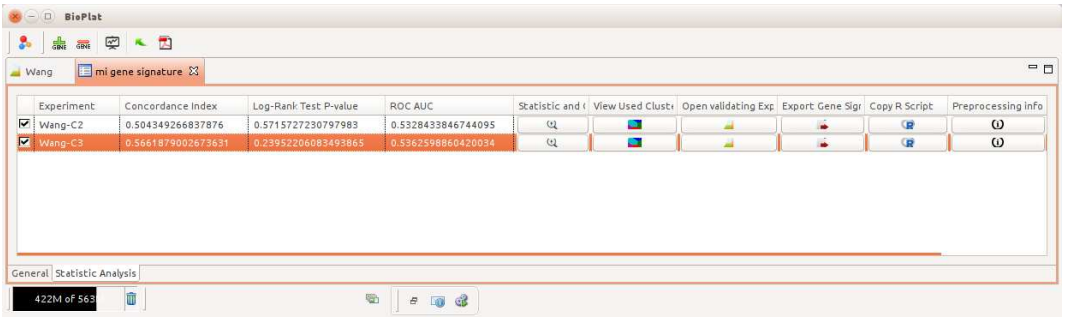


Figura 3.1-23 - Visualización de resultados de la validación

Los resultados que se incluyen actualmente son:

- Los valores de los resultados estadísticos más importantes: *Concordance-index*[19], *Log-rank test p-value*[20], *ROC AUC*[21];

- Un botón para visualizar más detalle del resultado del análisis. Incluye:
  - *Log-Rank Test Chi Squared*: estadístico y su p-value;
  - *Concordance-index*: estadístico y su p-value;
  - Curvas ROC: ilustradas en la Figura 3.1-24;
  - Mapa de calor (*HeatMap*). Se muestra en la Figura 3.1-25. Este mapa permite visualizar si existen o no diferencias de expresión entre los diferentes *clusters* generados. El tono de los rojos representa alta expresión génica y el tono de los verdes, baja.
  - Curvas de *Kaplan Meier*: Es el gráfico más representativo para medir el poder pronóstico. Muestra la correlación entre los *clusters* generados y los datos de supervivencia.

- La visualización del agrupamiento calculado, esto es la asignación del id de *cluster* correspondiente a cada muestra del experimento, se ilustra en la Figura 3.1-27;

[19] http://www.bioconductor.org/packages/3.3/bioc/manuals/survcomp/man/survcomp.pdf
[20] https://cran.r-project.org/web/packages/survival/survival.pdf
[21] https://cran.r-project.org/web/packages/survivalROC/survivalROC.pdf

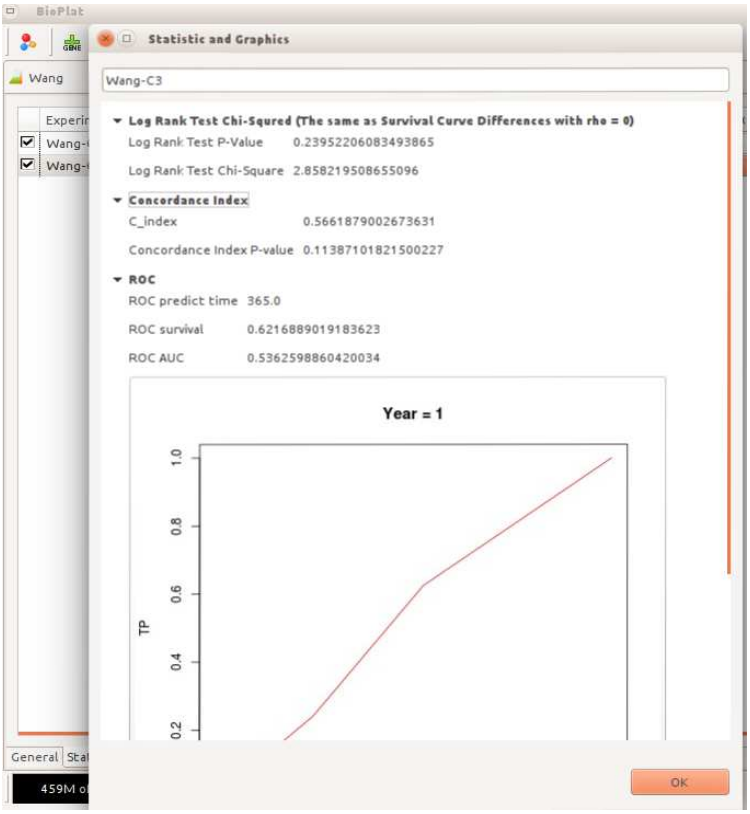


Figura 3.1-24 - Visualización de resultados estadísticos

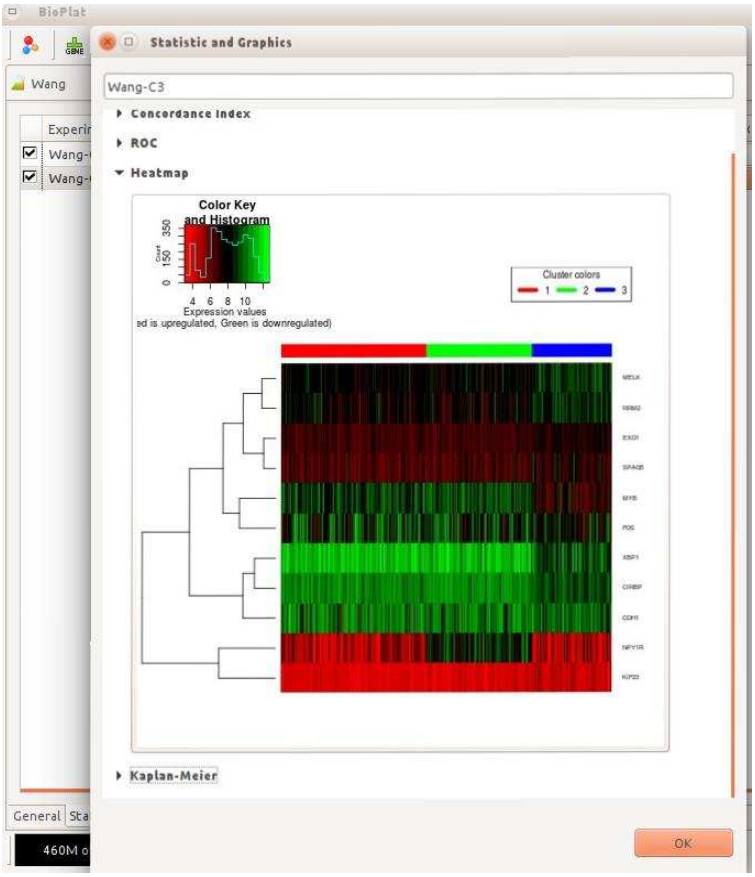


Figura 3.1-25 - Visualización de mapas de calor

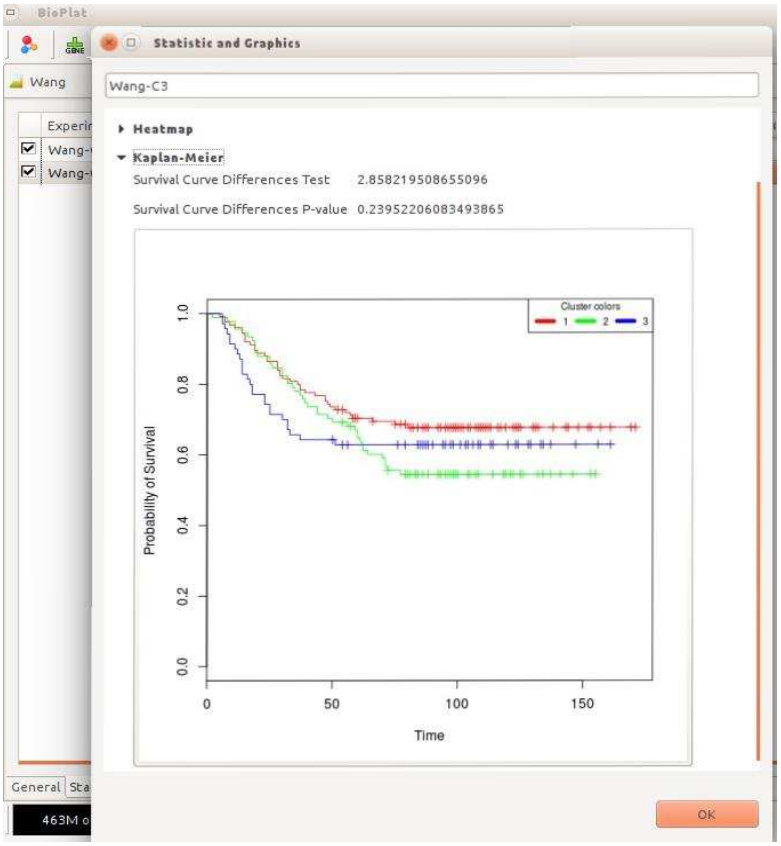


Figura 3.1-26 - Visualización de resultados: Kaplan-Meier

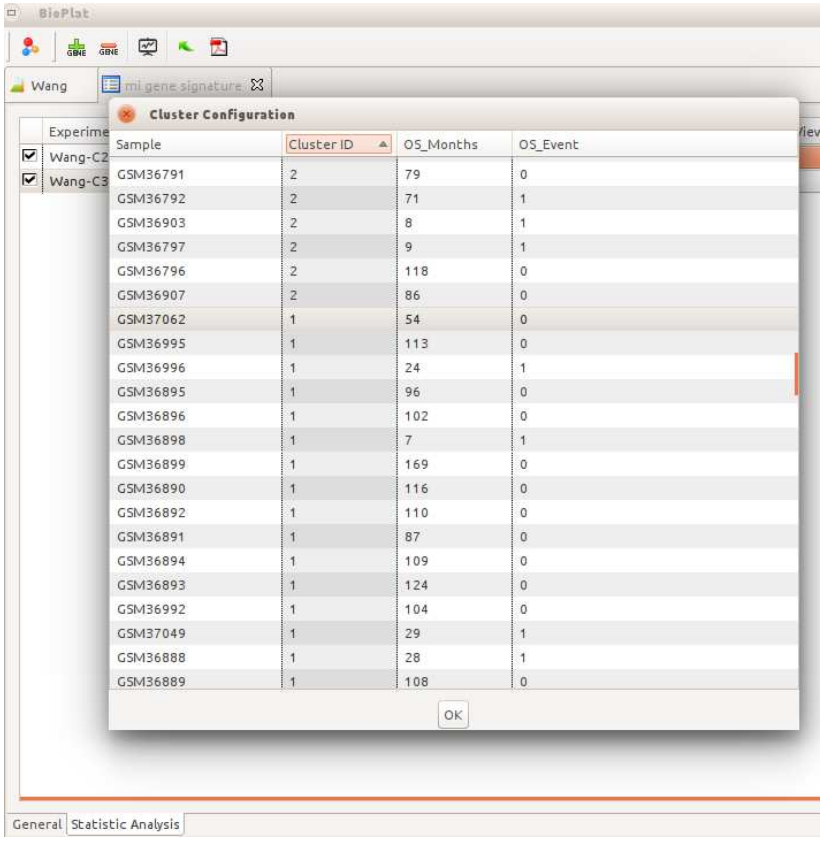


Figura 3.1-27 - Visualización de cluster utilizado

3.1.5 Optimización de *gene signatures*

Dijimos que un *gene signature*, a partir de su conjunto de genes, puede funcionar como marcador pronóstico de un proceso tumoral. Sin embargo, es frecuente querer evaluar si un gen determinado aporta o no información a un *gene signature*, desde el punto de vista del pronóstico. Ante tal sospecha, se quiere generar un *gene signature* sin ese gen y volver a evaluar su valor pronóstico/predictivo. Esto mismo puede ocurrir con varios genes sospechosos de no aportar información al *gene signature*.

La optimización de un *gene signature,* entonces, persigue la idea de optimizar, justamente, los genes presentes, es decir, mantener los genes que tengan real implicancia en la valoración de resultados del *gene signature* y los genes que sean neutros, que no aporten significancia en el resultado, sean descartados. Para esto se realiza una estrategia estadística de *feature selection*. Así un *gene signature* con un número reducido (optimizado) de genes es importante ya que puede facilitar posibles estudios subsiguientes.

Para agilizar esta actividad, la plataforma permite generar y validar automáticamente *genes signatures,* resultates de la combinación de genes de otro. Esta optimización se basa, entonces, en la búsqueda de la combinación de genes del *gene signature* original que, removidos, maximizan el valor pronóstico o predictivo en los experimentos evaluados. Como objetivo paralelo aparece la búsqueda del modelo más parsimonioso, que en este caso sería reducir la cantidad de genes del *gene signature* que mantenga (o mejore) el valor pronóstico del original.

Cualquiera de estos algoritmos y heurísticas para optimización provistos por la plataforma, necesitan de una medida que les permita comparar dos *gene signatures* y determinar cuál es el que presenta mejor valor pronóstico en el contexto de un experimento determinado. Actualmente la medida de comparación es el *concordance-index*, pero considerando que este es un punto crítico para los algoritmos, es que se está trabajando en nuevas métricas. La estrategia de comparación fue diseñada en la plataforma para ser fácilmente reemplazada sin afectar al núcleo de los algoritmos.

La plataforma ofrece actualmente dos algoritmos que automatizan esta actividad. El primero de los algoritmos busca en el espacio de soluciones completo, es decir generando todas las combinaciones de genes posibles, se lo llamó *Blind Search* y el segundo utiliza una heurística para reducir el espacio de soluciones, basado en el algoritmo PSO[22].

Estos algoritmos requieren una configuración en varios pasos, incluyendo la selección del experimento Bioplat para la fase de entrenamiento, para la fase de validación y también para la de *testing*. Para más detalle de la estrategia remitirse a la tesis citada [1]

Estos procesos son intrínsecamente complejos y costosos en tiempo computacional ya que en general, hacen validaciones sobre las combinaciones de genes, que forman subconjuntos del *gene signature* del cual se parte, para luego aplicar las validaciones que configure el usuario. Esta es una de las razones por las surgió en el cliente requerimiento de dar la posibilidad de correr operaciones de mediana/larga duración en segundo plano, permitiendo al usuario realizar cualquier otra operación. Un ejemplo de esto se ilustra en la Figura 3.1-28, donde vemos que una vez lanzada la operación en este caso *Blind Search,* aparece una vista con las mejores diez performances de los *gene signatures* que se derivan del original, y también aparece un diálogo que permite enviar el proceso a una vista especial, para visualizar su avance, y permitiendo que el usuario pueda realizar otra tarea en la plataforma. Cabe mencionar que el reporte de avance, cuando puede ser calculado como es este caso, da al usuario un *feedback* exacto de cuanto resta de la operación para que esta finalice.

[22] Particle Swarm Optimization Algorithm

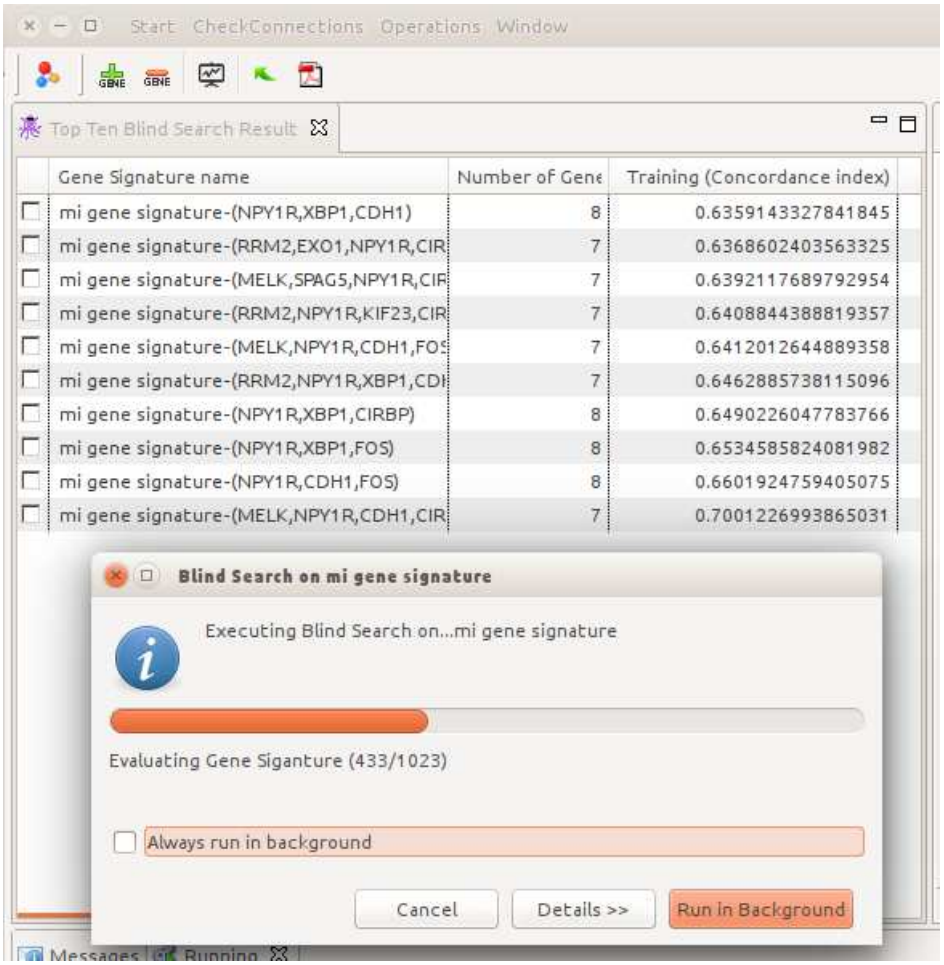


Figura 3.1-28 - Optimización utilizando Blind Search

3.1.6 Vista de genes

Hoy en día existen multitud de sitios que ofrecen información a través de interfaces bien definidas, tipo REST[24], siendo esta una gran posibilidad de reutilizar elementos ya existentes para enriquecer la experiencia de uso con la plataforma. La vista de genes está pensada para hacer uso de esta característica, dando información complementaria de los genes.

Cuando un usuario tiene una entidad abierta, sea un experimento o un *gene signature*, y desea ver información específica de un gen en alguna herramienta web bioinformática, puede integrarla fácilmente al cliente Bioplat, siempre y cuando el sitio soporte peticiones de recursos del estilo REST. Entonces, cuando seleccione el gen, podrá visualizar la información complementaria dentro de la plataforma.

La vista puede ser integrada a múltiples sitios web a través de las preferencias (ver sección Preferencias), siendo éste un punto de extensión en el cliente, que le permite al usuario la integración instantánea con sitios que le proveen información complementaria del gen que seleccione en la aplicación, ilustramos ejemplos de integración con NCBI, presentado información detallada del gen seleccionado en la Figura 3.1-29 y en la Figura 3.1-30, la integración con la aplicación STRING[25], en la cual se muestran los genes ''cercanos'' al gen seleccionado. La vista tiene sentido para cualquier entidad en el sistema que contenga genes en su composición, actualmente pueden ser los *gene signatures* o los experimentos Bioplat.

[24] Abreviatura de *Representational State Transfer*. Básicamente, define un modo standard de acceso a recursos web
[25] http://www.string-db.org/

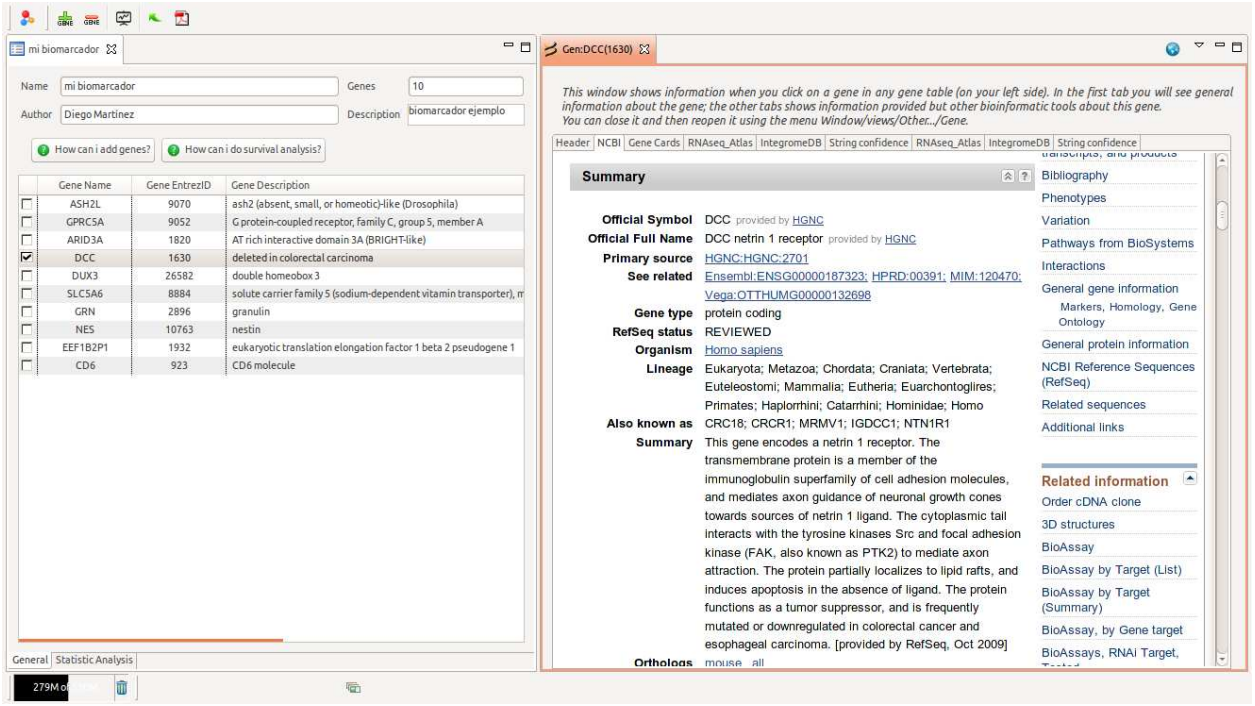


Figura 3.1-29 - Vista de genes: integración con NCBI

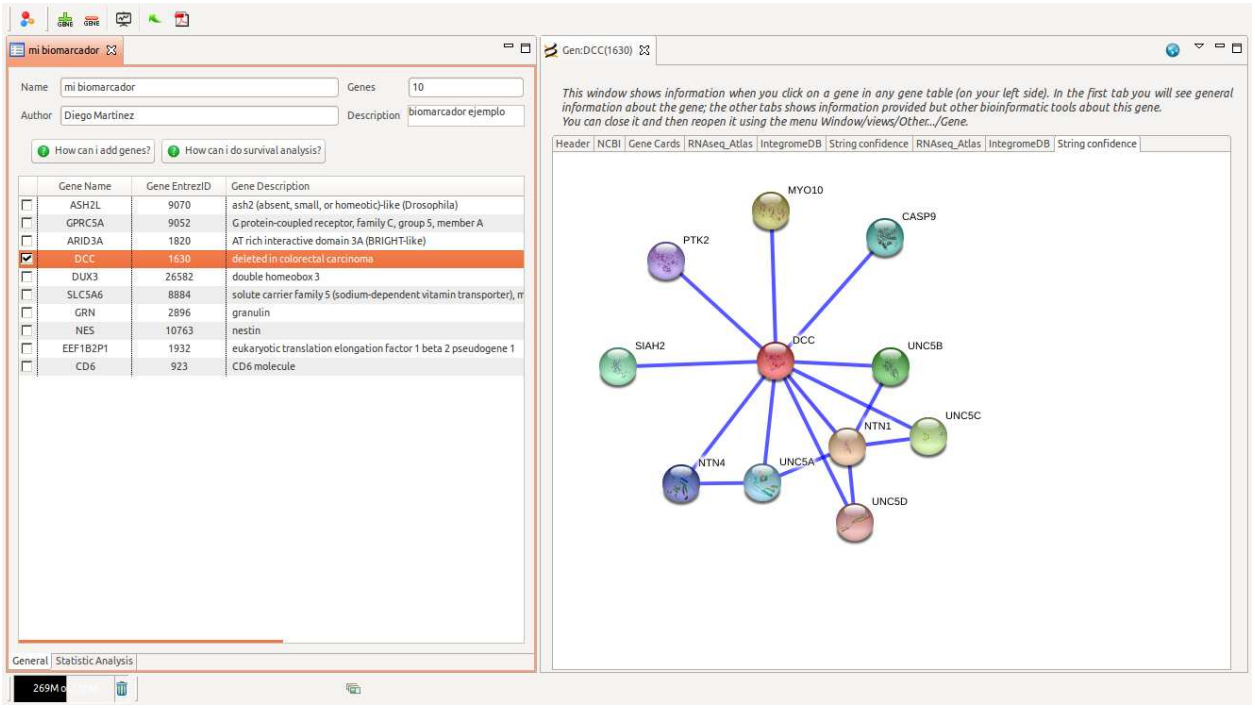


Figura 3.1-30 - Vista de genes: integración con web STRING

3.2 Solución tecnológica

En esta sección se profundizará en los resultados y soluciones técnicas desarrolladas para dar soporte a la construcción de la plataforma descrita en la sección anterior, La Plataforma desde el cliente.

Se analizarán primero las adecuaciones implementadas en el modelo, que dan soporte a diversas características que son necesarias para tener clientes más eficientes, un ejemplo de esto es el *feedback* que se da al usuario en operaciones de larga duración. Luego veremos las herramientas y *frameworks* básicos utilizados desde el cual se implementa el cliente Bioplat, para en última instancia ir hacia las soluciones propuestas que se construyen utilizando estas tecnologías de base, y que permiten la extensión del cliente de un modo eficiente.

3.2.1 Revisión de la arquitectura general de la plataforma

Con la incorporación de nuevos miembros al equipo integral de desarrollo de la plataforma, se pudieron planificar nuevas componentes, mejor definidas, tomando como base las experiencias del antiguo esquema. Una de ellas en el nuevo cliente motivo de este trabajo.

Si bien el esquema arquitectural de la Figura 2.1-1 introduce un desarrollo desacoplado, orientado a componentes/capas. La nueva arquitectura profundiza el desarrollo orientado en ese sentido, creando nuevas componentes que se comunican entre sí mediante interfaces que no expongan sus implementaciones internas, favoreciendo el bajo acoplamiento entre ellas. A modo de introducir la nueva arquitectura y sus implicancias, se dará una breve descripción del objetivo de cada componente profundizando luego en las que conciernen directamente al desarrollo del cliente visual y sus propuestas de solución.

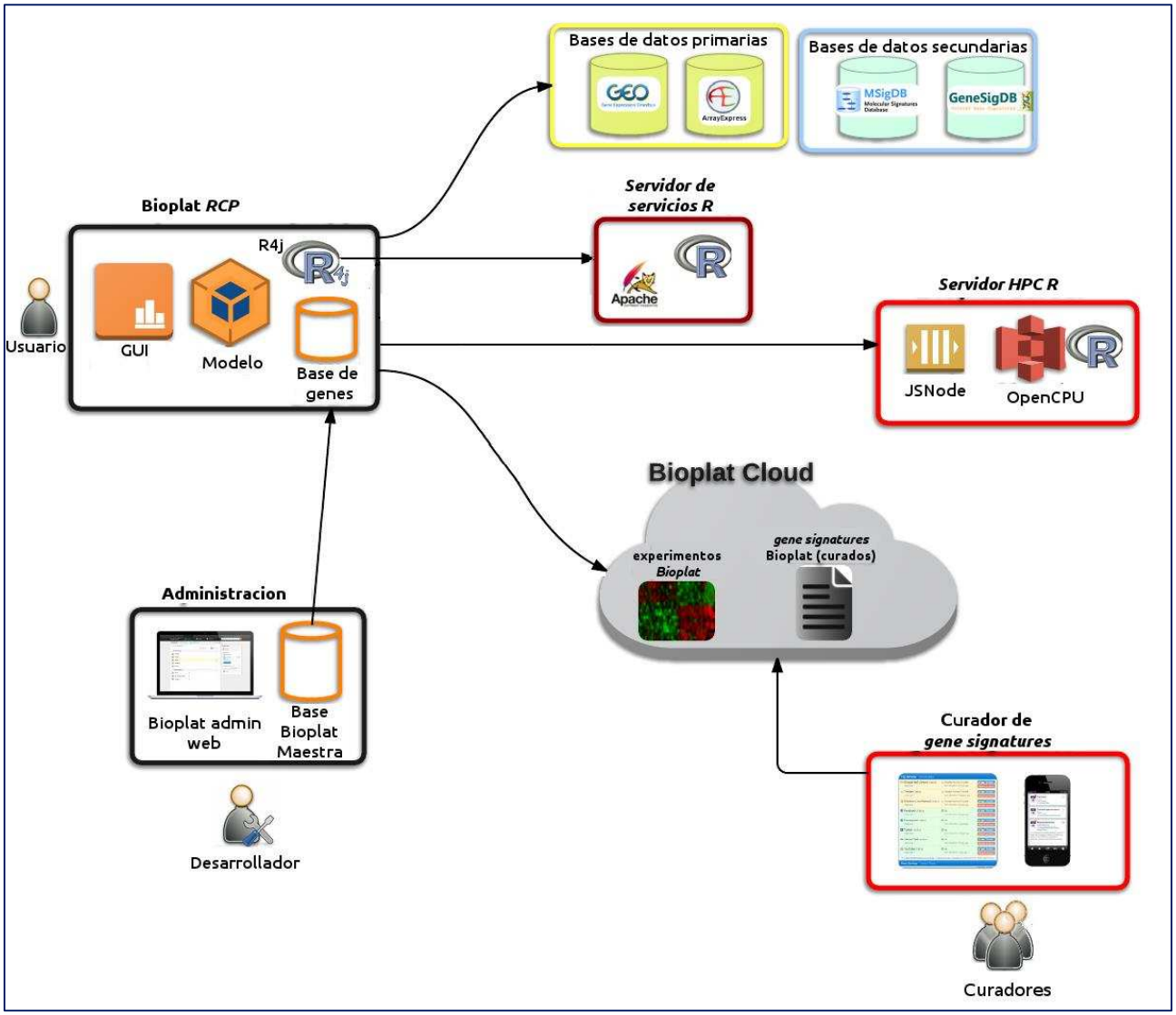


Figura 3.2-1 – Arquitectura futura general de Bioplat

En la Figura 3.2-1, vemos el esquema que tendrá la nueva arquitectura y a continuación se dará una breve descripción de los componentes que se muestran:

Bioplat RCP

Modelo. En el esquema lo nombramos como ''Modelo'' y forma parte del cliente. Define las entidades funcionales principales del sistema, distintos tipos de experimentos, *gene signatures*, genes. También define entidades y comportamiento comunes no funcionales como el soporte para *data binding* entre objetos, interfaz de monitor de progresos, etc. (ver sección Revisión del modelo de la plataforma);

Cliente Bioplat (GUI). Construye la interfaz gráfica para las entidades conceptuales que se definen en el Modelo de la plataforma. Se profundizará en el desarrollo de este cliente, objeto de esta tesina, en las secciones subsiguientes. (ver sección Cliente Bioplat)

R4j. Esta componente es desarrollada por el equipo, permite la ejecución de scripts R desde Java, transformando los objetos de un contexto al otro. Se comunica mediante HTTP al Servidor R. Esta capa de integración con R, define todos los objetos intervinientes en la comunicación con dicho servidor, y los cuales también consume el cliente gráfico para presentar los resultados.

Base de genes. El Modelo hace uso intensivo de la información de genes, entre otros datos, con lo cual se distribuye una base de datos con esta información para mejorar la performance general del sistema.

Servidor de servicios R. Este servidor publica servicios fundamentales para poder llevar a cabo cálculos sobre las entidades del sistema. Estos cálculos son más eficientemente procesados y expresados en el lenguaje de programación R. Este lenguaje está especialmente diseñado para cálculo estadístico y gráfico de funciones, y es ampliamente utilizado en el área de bioinformática. El cliente interactúa con este servidor a través de la capa R4j, comentada anteriormente, que está integrada en la infraestructura del cliente. La publicación de los servicios que provee se realiza a través del protocolo HTTP, y así proveer de un mecanismo relativamente conveniente para accederlo, es decir sortear *proxies* web, etc. (ver sección Servidor R como servicio)

Bases de datos primarias y secundarias. Bioplat utiliza estas fuentes de datos como eventual punto de entrada para confeccionar nuevos *gene signatures* a partir de otros publicados y almacenados en ellas. Un escenario de uso sería la construcción de un *gene signature* a partir de algoritmos heurísticos que analizan estos bancos de datos con algún criterio, como vimos en la sección Herramientas de generación/importación de *gene signatures*

Servidor HPC R. Esta componente está en proceso de desarrollo. Su objetivo es procesar los requerimientos u operaciones de larga duración que la aplicación puede disparar.

Dando la posibilidad de que estas sean ejecutadas asincrónicamente. Por ejemplo tendrá servicios para dar soporte a la optimización (ver sección Optimización de *gene signatures*)

Bioplat Cloud. Será un repositorio donde se albergarán los estudios parciales y publicados de cada usuario, de esta manera se podrán compartir estudios entre distintos usuarios utilizando un soporte dado por la plataforma. Actualmente en desarrollo en el equipo.

Curador de *gene signatures*. Esta componente permite curar todos aquellos datos de Bioplat *Cloud*. Ya sean datos privados que un usuario decide compartir con la comunidad, como datos que los curadores Bioplat decidan incorporar de otras herramientas bioinformáticas o de diferentes publicaciones de interés. Esta herramienta no está iniciada.

Administración. Esta herramienta mantendrá actualizada la base de datos local de la plataforma, incluyendo la actualización del genoma, de los genes y sus distintos identificadores alternativos. Esta componente se encuentra en desarrollo y actualmente se realiza a través de programas desarrollados para tal fin, que se corren periódicamente.

3.2.2 Servidor R como servicio

Vimos en la sección Arquitectura general, específicamente ilustrado en la Figura 2.1-1, que dentro del prototipo corría un servidor que daba soporte a las operaciones estadísticas y gráfico de funciones a través del lenguaje y entorno de programación R, especialmente diseñado y optimizado para esta tarea. Este *servidor* corría nativamente en el sistema operativo donde era instalado, incluyendo configuraciones propias en cada sistema, con fuerte incidencia en las librerías que el cliente pudiera tener preinstaladas, entre otras complicaciones. Toda esta configuración requería necesaria intervención y soporte de un especialista para la correcta instalación, y posterior correcto funcionamiento del producto, dificultando considerablemente la instalación y distribución del cliente en las PCs de los usuarios finales. Este hecho claramente atentaba contra la portabilidad y facilidad de distribución del cliente, parte de los objetivos de esta tesina.

Movido por las tendencias SaaS (*Software as a Service*) es que se decidió extraer el servidor del cliente, y publicar sus servicios a través del protocolo de aplicación web HTTP. Esto tiene ciertas ventajas propias de esta técnica de distribución de software como pueden ser: administración remota del servicio, la configuración y librerías requeridas se configuran centralizadamente y no en el cliente. Esto tiene implicancias positivas en nuestro caso particular, entre ellas se destacan las siguientes: la configuración del entorno de ejecución R, es demasiado compleja para dejarla en manos del usuario; las versiones de librerías, incluso las librerías propias que desarrolla el equipo especializado en R y se distribuyen como parte de la plataforma, y la configuración en general tienen que ser estrictamente controladas, ya que un cambio en estas versiones o una incorrecta configuración puede devolver incompatibilidades o errores en los cálculos que el cliente utiliza para presentar muchos de los resultados de las operaciones ejecutas haciendo uso de estos servicios, con lo que debe ser verificado y validado por los desarrolladores de la plataforma, que como se dijo, se hace centralizadamente donde corren los servicios.

Con este nuevo esquema, cabe mencionar que los servicios van a ser accedidos por todos los clientes Bioplat, teniendo que conservar, en lo posible, la performance observada cuando este corría localmente en las PCs de los usuarios. Este problema es atacado mediante la configuración de un *cluster* de servidores que proveen los servicios. Actualmente se utiliza Docker[26], que es un virtualizador de sistemas operativos, el cual facilita la construcción y manejo de dependencias para un aplicación, en nuestro caso nos permite configurar una imagen, es decir definir nuestra versión de R, su configuración, las librerías necesarias, etc. Y luego disparar la ejecución de estas imágenes, *containers* en términos de Docker, como nuevos nodos del cluster antes mencionado, logrando mantener la performance buscada en niveles satisfactorios. Luego se liberan estos *containers* si la demanda baja, devolviendo los recursos físicos de los servidores que estén involucrados.

De esta manera se le da, en gran medida, portabilidad al cliente y facilidad en su distribución.

[26] https://www.docker.com/

3.2.3 Revisión del modelo de la plataforma

El modelo de la plataforma se encarga como ya se mencionó de la definición de las entidades básicas de la plataforma, y de las operaciones que se pueden realizar con ella. Con la introducción del nuevo cliente, se introdujo también una nueva estrategia para el desarrollo del modelo, haciendo que este de soporte a nuevas características. Mencionaremos dos grandes rasgos a los que el modelo ahora da soporte.

En primer lugar se introducirá el soporte para la actualización de propiedades/aspectos del sistema. Para esto se requiere del uso de principios definidos en el *standard* Java Beans [4], como por ejemplo convención de nombres para la definición métodos, y que luego permitirá al cliente integrarse como se detallará en la sección *Widgets*. A continuación se comenta la estrategia de funcionamiento básica. Cada entidad básica que requiera este soporte, extenderá una clase base que proporcionará los mecanismos ahora definidos por el Modelo, para la correcta comunicación de cambios. La clase abstracta agregada al modelo se llama *AbstractEntity*. De esta manera cada entidad será responsable de indicar los aspectos que hayan cambiado internamente y que deban ser comunicados a los interesados, previamente suscriptos al esquema de notificación. Estos interesados pueden ser otros objetos del sistema, que realicen algún cálculo en función del cambio, el ejemplo por excelencia de esta situación es la actualización de la visualización en pantalla de la entidad, cuando esta cambia internamente por algún proceso. Esto desacopla eficientemente al modelo y su implementación interna de cómo y cuándo los interesados se enteran de cambios y sincronizan su estado, siendo estas características manejadas automáticamente.

En segundo lugar, y en otro orden de adaptaciones al Modelo, las operaciones que realiza la plataforma suelen ser extensas en el tiempo con lo cual se le dio al modelo el soporte para implementación de indicación de progreso, con tiempo estimado de terminación (cuando este puede ser calculado). El Modelo define una *interface* Java de monitoreo, con métodos que permiten indicar el progreso, indicar que terminó una tarea, etc. Esta *interface* se llama *Monitor*. La implementación la provee el cliente, siendo transparente para el Modelo cómo se informa visualmente el progreso de una operación.

Como puede verse el foco principal está puesto en el desacoplamiento del modelo de la plataforma de sus clientes. En este caso, y como empezaremos a ver en detalle en las secciones subsiguientes, el cliente se implementará utilizando la infraestructura de Eclipse que es el que define los lineamientos generales de interfaz gráfica y el modelo dará el soporte mencionado permitiendo integrar estos lineamientos.

3.2.4 Cliente Bioplat

A lo largo de esta sección, analizaremos la implementación del cliente que da soporte a la plataforma. Comenzaremos describiendo las tecnologías básicas utilizadas para llegar luego, a la implementación de características distintivas del cliente.

3.2.4.1 Tecnologías básicas

La selección de herramientas utilizadas resulta ser una decisión muy importante y debe hacerse en etapas tempranas del desarrollo. Para tomar estas decisiones se tuvieron en cuenta y se analizaron un conjunto de variables, por ejemplo entornos donde se usará la aplicación, documentación disponible, automatización de procesos, etc.

A continuación presentaremos las herramientas seleccionadas para el desarrollo del cliente, remarcando las principales características por las que estas fueron escogidas. Para una vez introducidas, ponerlas en relación al cliente Bioplat y el aporte que hacen a este.

Java

Java[28] es el lenguaje de programación principal elegido para hacer la implementación del cliente. Este es un lenguaje robusto, con una gran comunidad de desarrolladores que continuamente proveen mejoras en componentes de *software* reusables y que agilizan considerablemente la programación. Existen multitudes de bibliotecas libres que resuelven problemas que comúnmente se presentan a la hora de desarrollar una aplicación como por ejemplo, interacción con protocolos como HTTP, bases de datos, entre muchas otras integraciones con herramientas existentes en el mercado.

Java es un lenguaje que se compila a un formato propio conocido como ''*bytecode*'', y que luego es interpretado y ejecutado sobre otro programa escrito en código nativo del sistema operativo/hardware destino, este programa se conoce como máquina virtual. Así Java garantiza la portabilidad de las aplicaciones en él desarrolladas,

[28] http://www.oracle.com/technetwork/java/langenv-140151.html

siempre que exista la máquina virtual mencionada. Esta estrategia de diseño es la responsable de que Java pueda ser programado/compilado una vez y ejecutado en cualquier sistema operativo que corra su máquina virtual.

Es un lenguaje de programación en continuo crecimiento, ya que su sintaxis se expande versión a versión para incluir nuevas características, a la vez que garantiza cierta compatibilidad con versiones anteriores del lenguaje y su máquina virtual, aunque a veces este aspecto es criticado por presentar muchas veces una limitación o complicación para la implementación de nuevos aspectos del lenguaje.

Es un lenguaje de programación orientado a objetos, con que el desarrollo utilizará las estrategias y buenas prácticas propuestas por este paradigma de programación.

OSGi

OSGi[30] es acrónimo de *Open Service Gateway Initiative*. Describe por un lado el marco para dar soporte al desarrollo de un sistema modularizado y una plataforma de publicación de comportamiento, a la que llama Servicios, y provee un modelo desacoplado de comunicación entre dichos módulos. Cada uno de estos módulos se lo llama componente o *bundle*. [5]

De esta manera, provee un mecanismo natural para desarrollo orientado a componentes, y por otro una plataforma de administración de servicios dinámica que cada componente puede publicar o consumir. La concepción de dinamismo tiene que ver con la posibilidad de que los servicios, y en general los componentes, pueden ser detenidos, eventualmente actualizados, y arrancados nuevamente, dando la posibilidad de hacer cambios en los módulos sin la necesidad de detener el sistema completo.

Los mecanismos de modularización no son provistos a nivel de lenguaje por Java, es decir no son implementados nativamente en él. Se pueden ver como una extensión de los modificadores de acceso nativos que sí existen en Java, pudiendo así administrar la visibilidad de paquetes/clases entre módulos. [5]

[30] https://www.osgi.org/

A continuación exponemos el uso del archivo descriptor definido por el *standard* OSGi, donde se lo llama MANIFEST.MF. Se dará una breve descripción y uso de las principales propiedades utilizadas en el cliente Bioplat. Indicando comentarios sobre alguno de sus beneficios.

```
1.  Manifest-Version: 1.0
2.  Bundle-ManifestVersion: 2
3.  Bundle-Name: Bioplat Core
4.  Bundle-SymbolicName: edu.unlp.medicine.bioplat.core;singleton:=true
5.  Bundle-Version: 1.0.1.qualifier
6.  Require-Bundle: org.eclipse.ui;visibility:=reexport,
7.   org.eclipse.core.runtime;visibility:=reexport,
8.   org.eclipse.core.databinding;visibility:=reexport,
9.   org.eclipse.core.databinding.beans;visibility:=reexport,
10.  org.eclipse.core.databinding.property;visibility:=reexport,
11.  org.eclipse.jface.databinding;visibility:=reexport,
12.  org.eclipse.core.databinding.observable;visibility:=reexport,
13.  org.eclipse.ui.forms;visibility:=reexport,
14.  com.ibm.icu;visibility:=reexport,
15.  org.eclipse.ui.intro,
16.  edu.medicine.bioplat.base.lib;bundle-version="1.2.0"
17. Bundle-ActivationPolicy: lazy
18. Bundle-RequiredExecutionEnvironment: JavaSE-1.6
19. Bundle-ClassPath: .
20. Bundle-Localization: OSGI-INF/l10n/bundle
21. Bundle-Activator: edu.unlp.medicine.bioplat.core.Activator
22. Export-Package: edu.unlp.medicine.bioplat.core,
23.  edu.unlp.medicine.bioplat.rcp.core.handlers,
24.  edu.unlp.medicine.bioplat.rcp.core.selections,
25.  edu.unlp.medicine.bioplat.rcp.editor,
26.  edu.unlp.medicine.bioplat.rcp.ui.entities.actions,
27.  edu.unlp.medicine.bioplat.rcp.ui.entities.editors.contributors,
28.  edu.unlp.medicine.bioplat.rcp.ui.entities.preferences,
29.  edu.unlp.medicine.bioplat.rcp.ui.entities.wizards,
30.  edu.unlp.medicine.bioplat.rcp.ui.entities.wizards.databinding,
31.  edu.unlp.medicine.bioplat.rcp.ui.utils,
32.  edu.unlp.medicine.bioplat.rcp.ui.utils.accesors,
33.  edu.unlp.medicine.bioplat.rcp.ui.utils.async,
34.  edu.unlp.medicine.bioplat.rcp.ui.utils.databinding.validators,
35.  edu.unlp.medicine.bioplat.rcp.ui.utils.extension.points,
36.  edu.unlp.medicine.bioplat.rcp.ui.utils.preferences,
37.  edu.unlp.medicine.bioplat.rcp.ui.utils.tables,
38.  edu.unlp.medicine.bioplat.rcp.ui.views.messages,
39.  edu.unlp.medicine.bioplat.rcp.utils,
40.  edu.unlp.medicine.bioplat.rcp.utils.wizards,
41.  edu.unlp.medicine.bioplat.rcp.widgets,
42.  edu.unlp.medicine.bioplat.rcp.widgets.listeners,
43.  edu.unlp.medicine.bioplat.rcp.widgets.wizards
```

Código 3.2—1 - Ejemplo de archivo descriptor Manifest.MF

Donde:

Bundle-SymbolicName: indica el nombre identificador del *bundle*. El modificador singleton indica que habrá una única instancia del *bundle* esto es importante ya que habilita el uso de puntos de extensión en el mundo Eclipse (a los cuales se volverá más adelante en las siguientes secciones)

Bundle-version: versión del *bundle*

Require-bundle: Aquí se indican las dependencias. Estas se especifican citando su Bundle-SymbolicName y admiten distintos modificadores para cada dependencia individual. Por ejemplo, *''visibility=reexport''* indica que las dependencias se ven transitivamente a cualquier otro *plugin* que vea al que estamos definiendo. Se puede indicar la dependencia a alguna versión o rango de versiones de las que depender, para asegurar compatibilidad;

Bundle-activationPolicy: esta propiedad indica el momento en que se activa el *bundle* cuando la aplicación arranca. Es deseable, y en general perfectamente posible, que los *bundles* se activen ''perezosamente'' *(lazy),* esto es cuando alguna clase, interfaz, u otro recurso del componente es requerido, lo que no es necesariamente cuando arranca la aplicación; este rasgo permite mantener la performance de arranque y uso de la aplicación a la vez que va creciendo en nuevos componentes/*bundles,* y estos se activan a demandan.

Bundle-Classpath: cada *bundle* administra su propio *classpath*. En esta propiedad suele indicarse ''.'' para incluir todos los recursos que componente el *bundle*.

Bundle-Localization: se indica el directorio donde están los archivos necesarios para soporte de internacionalización (idioma) de la aplicación.

Bundle-activator: indica la clase que define el comportamiento cuando el *bundle* es activado, y el comportamiento correspondiente para cuando es detenido. Esto es importante si el *bundle* ofrece, registra, servicios a otros, ya que permite hacer los ajustes necesarios ante una eventual baja del *bundle* por ejemplo para su actualización, y posterior *rearranque* sin necesidad de bajar toda la aplicación.

Export-package: indica a nivel de paquete Java, que es visible fuera del bundle. Esta propiedad es particularmente importante ya que extiende los modificadores de visibilidad definidos por el lenguaje Java, por ejemplo permitiendo a una clase o *interface* ser pública, pero no ser visible fuera del *bundle*. En el ejemplo del cliente bioplat, se exportan utilidades comunes y genéricas, pero no sus implementaciones internas como

pueden ser clases *helpers*, que claramente no tienen sentido fuera del *bundle* y no forman parte de la API del módulo.

Maven

El desarrollo de todo el núcleo de la plataforma que se venía implementando, paralelamente con el prototipo, por citar algunas componentes Modelo, R4j, ya venían utilizando *Maven*[32] para la administración del proyecto, es decir usando las estrategias propuestas por esta herramienta. Para el desarrollo del cliente se decidió también utilizar esta herramienta.

Así se incorpora *Maven* para lograr un mejor manejo de los distintos componentes que hacen al cliente, y mejorar la integración con el resto de la plataforma. *Maven* administra tanto aspectos del ciclo de desarrollo de un proyecto, como también sus ciclos de *testing automático*, empaquetamiento y publicación de módulos, suministrando en todas estas etapas distintas parametrizaciones que permiten adaptar requerimientos propios de cada proyecto particular. Volveremos a las características que nos provee *Maven* en este sentido y cómo estas se introdujeron al desarrollo del cliente, en la sección *Toolkit UI*.

Eclipse

Eclipse[34] es una comunidad de software libre que provee distintas aplicaciones, que ayudan al desarrollo de software. Entre estas se incluye el popular entorno de programación llamado Eclipse IDE, pero también un variado conjunto de aplicaciones que van desde herramientas para desarrollo web, manipulación de datos, etc. Todas estas aplicaciones se construyen a partir de la misma base arquitectural, la que se llama Eclipse RCP (*Rich Client Platform*). [6]

Eclipse RCP, entonces, define una plataforma genérica sobre la cual permite desarrollar "clientes ricos". El concepto cliente rico fue introducido en los años noventa con los emergentes entornos de desarrollo que facilitaban la programación de

[32] https://maven.apache.org/
[34] https://www.eclipse.org/

aplicaciones con componentes visuales atractivos y potentes, que daban alta calidad y alta confortabilidad a los usuarios, entre ellos podemos citar a *Visual Basic* y *Delphi*. La metáfora o expresión de ''riqueza'' viene dada en contraposición a las aplicaciones *simples* de esa época orientadas a una "línea de comandos". [6]

Eclipse RCP entonces, sienta las bases sobre las cuales desarrollar una aplicación rica en conceptos visuales, esto incluye integración con sistemas operativos modernos, haciendo uso de las capacidades visuales de este nativamente como veremos en profundidad en la sección *Toolkit UI*. Pero Eclipse va un paso más allá en este sentido dando un marco para la estructuración visual de una aplicación definiendo distintos modos de mostrar las entidades del sistema, dependiendo de su naturaleza. Podemos encontrar entidades complejas de visualizar, por ejemplo tienen propiedades, tienen relaciones con otras entidades, puede también ser necesario visualizar distintos conjuntos de datos a la vez, etc. Para facilitar la visualización de esta información Eclipse define e implementa dos conceptos, editores y vistas. En los editores se muestran las entidades centrales de la aplicación y luego, para información que sirve de modo complementario a este, se utilizan las vistas. Así, el entorno de trabajo del usuario puede organizarse visualizando de modo central una entidad en su editor y una colección de vistas "alrededor" que complementen el trabajo que se realiza sobre este. Esto da un marco de trabajo muy flexible y personalizable para el usuario.

La plataforma Eclipse en sí fue implementada teniendo en mente su extensibilidad. Así definió todo el protocolo para llevar esto a cabo. Por un lado Eclipse, a través de un proyecto llamado *Equinox*[35], implementa la especificación *OSGi* que vimos anteriormente, de modo que permite utilizar todas las características descritas en esa sección. Rebautiza los *bundles* de la especificación original, como *plugins*. Usaremos indistintamente el término a menos que se requiera alguna aclaración [6].

En el caso del cliente Bioplat, por ejemplo se puede pensar en servicios de reporte de mensajes, un proveedor de genes, etc. Estos servicios se implementan en los distintos plugins, así Eclipse, apoyado en *Equinox,* da también soporte a una administración dinámica de los plugins, permitiendo activarlos o desactivarlos. Decimos que da soporte ya que, en general no es automático que todo continúe funcionando, sino que es necesario manejar cómo se comportarán los plugins que dependan de alguna

[35] http://www.eclipse.org/equinox/

funcionalidad provista por el *plugin* a desactivar, en general es perfectamente posible manejando estos casos [6]. Esta característica es muy interesante ya que permite que por ejemplo ante un *bug* en la implementación de un servicio, este se pueda corregir, generando un nuevo *plugin* que reemplace al que presenta el defecto.

Por otro lado, Eclipse RCP define un marco propio de extensiones, decisivamente importante como soporte para el posible enriquecimiento de una aplicación de este estilo de manera consistente. En Eclipse estos puntos son conocidos justamente como Puntos de extensión. Los Puntos de extensión pueden incluir implementaciones o ser completamente declarativos.

El funcionamiento de un punto de extensión plantea la idea de que cada *plugin* aporte a la plataforma según otros plugins así lo hayan definido. Se puede trazar una analogía, a modo de graficar este esquema de funcionamiento, con los tomacorrientes y los enchufes de electrodomésticos, donde los primeros definen una interfaz, un voltaje, etc. Luego puede enchufarse un artefacto o no si cumple con esa interfaz. Así podemos ver a los plugins como los que definen estos tomacorrientes, luego otros plugins, como artefactos, se enchufan suministrando alguna funcionalidad. Esto beneficia el bajo acoplamiento entre *plugins*, pudiendo estos interactuar sin necesidad de tener dependencias de implementación, solo con una interfaz bien definida. Como corolario de esta estrategia, se desprende que un *plugin* que define su punto de extensión, puede tener varios plugins que lo quieran contribuir y dependiendo del punto de extensión, este puede procesar todos o tomar en cuenta solo uno. En la plataforma, a modo de ejemplo, un punto de extensión son las operaciones que se pueden realizar sobre un experimento Bioplat. En el cual la idea es que distintos plugins contribuyan a un mismo punto, extendiendo la operatoria que se pueda realizar con la entidad. Esto refuerza la idea de desacoplamiento, pero también es un punto muy interesante para la abstracción a la hora de extender la plataforma. Profundizaremos este aspecto en la sección Framework de desarrollo

Habiendo comentado estas dos características básicas provistas por Eclipse RCP, primero *Equinox* como una implementación de OSGi para la administración dinámica de los plugins y segundo la arquitectura de puntos de extensión para definir el modo en que un *plugin* contribuye nueva funcionalidad, vemos que en conjunto, un concepto complementándose al otro, nos provee de una arquitectura sumamente potente para la

extensibilidad dinámica del cliente. Una de las grandes ventajas de esta estrategia es que se desconoce quiénes suministran las contribuciones, dando inmediatamente un marco natural de extensión.

Al comienzo de la sección se dijo que Eclipse RCP es una comunidad que desarrolla distintas herramientas reusables, ahora sabemos que estas herramientas se definen siguiendo las reglas de puntos de extensión, que se definen, implementan y agrupan en plugins. Algunos de estos puntos de extensión son provistos por la arquitectura básica de Eclipse RCP, por ejemplo para la definición de un producto Eclipse, es decir la aplicación y su configuración se define una extensión para ese punto, y otros pueden ser agregados, a través de la inclusión de plugins adicionales, para enriquecer el abanico de funcionalidades o *frameworks* disponibles para extender una aplicación, como ejemplo de estos plugins, tenemos al sistema de ayuda contextual, el *framework* de soporte para data binding [6], entre una gran variedad de alternativas.

Se volverá más adelante a la arquitectura conceptual de una aplicación desarrollada sobre Eclipse RCP y sus características, pero antes diremos que este tipo de aplicaciones se ejecutan en el cliente, es decir, definen una aplicación de escritorio. La contrapondremos con una aplicación web y analizaremos sus diferencias.

3.2.4.2 ¿Por qué una aplicación de escritorio?

Una de las grandes decisiones que se tuvieron que tomar para encarar el proceso de reemplazo del prototipo fue resolver la disyuntiva entre desarrollar una aplicación de escritorio o una aplicación web. Como ya se comentó, la elección se perfiló hacia un desarrollo en Escritorio o *Desktop*. A continuación veremos las distintas variables que se tuvieron en cuenta para tomar este camino.

Como viene siendo la lógica del texto, se mencionarán a un nivel general los distintos aspectos que se pueden llegar a tener en cuenta, para luego sí, ponderarlos en función de nuestro cliente Bioplat.

Hoy día, tanto aplicaciones de escritorio como *webs*, son muy potentes, con gran riqueza tanto visual como funcional. Aun así, siguen existiendo diferencias o limitaciones sustanciales a la hora de encarar el desarrollo de la interfaz de cierto tipo de aplicaciones. Aunque suene obvio, esta interfaz no es ni más ni menos con la que los usuarios interactuarán con el sistema y lo que en definitiva será cara visible de la plataforma, con lo que es de suma importancia tomar una buena decisión en este punto.

Las aplicaciones de escritorio son aquellas que se ejecutan, básicamente, de manera directa y nativa en el sistema operativo. Este tipo de aplicaciones tienen un largo recorrido en la industria de software. Históricamente, es el modo en que se distribuyen las aplicaciones. Detallaremos brevemente sus ventajas y desventajas. Entre las primeras caben mencionar:

Análisis de datos complejos (Big Data): Como las aplicaciones de escritorio se ejecutan en la pc del usuario, tiene la posibilidad de disponer de todos los recursos locales sin restricciones. Así es particularmente beneficioso para el análisis y presentación de datos complejos (potencia en gráficos, almacenamiento, procesamiento, etc.)

Posibilidad de correr offline. Esta es una ventaja, si bien en un mundo sumamente interconectado por Internet, seguramente este no sea 100% un punto positivo ya que de una manera u otra puede afectar al completo funcionamiento de la aplicación. Aun así es un punto a tener en cuenta a la hora del diseño de la aplicación y analizar qué beneficios se puede obtener de este aspecto.

Citamos algunas desventajas:

Actualización a demanda: este tipo de aplicaciones en general, requieren que el usuario sea el encargado de actualizar el software. Si bien, de ser estrictamente necesario, el sistema se puede programar para que chequee actualizaciones y obligue o no al usuario a realizarla para poder seguir operando.

Distribución e instalación: La distribución suele ser compleja, por dependencias con distintos componentes propios del sistema operativo, por ejemplo tener algún *framework o runtime* preinstalado, etc. Esto suele obligar al usuario a tener ciertos conocimientos que terminan siendo engorrosos y molestos, ya que de lo que se trata es que directamente el usuario pueda operar con la aplicación, sin más. Esto claramente atenta contra la portabilidad, no será el mismo *set* de dependencias en una PC que en otra con distinto sistema operativo o aplicaciones.

Mientras que en las aplicaciones web intervienen las siguientes particularidades. Estas aplicaciones no corren ni directa ni totalmente en las máquinas de los usuarios, sino que lo hacen en los servidores que la proveen y se ejecutan dentro de un browser de internet instalado en la máquina cliente. Los usuarios están habituados a este tipo de aplicaciones. Se tiene una portabilidad *ad hoc*, aunque pasa a depender implícitamente de los distintos browsers de internet (lo cual actualmente no representa un gran problema). La seguridad, o inseguridad inherente a Internet, se vuelve un tema a tener muy en cuenta, aquí vuelve a jugar un papel relevante el *browser*, debiendo estar correctamente actualizado para minimizar estos riesgos. La escalabilidad es relativamente fácil de proveer, ya que multiplicar los servidores puede y debería ser, transparente para los usuarios. Siempre se cuenta con la versión actualizada de la aplicación, la cual es relativamente sencilla de instalar, o por lo menos es, nuevamente, transparente para el usuario de la aplicación. Al tener un acceso relativamente limitado a los recursos del sistema, es difícil lograr una buena performance en la presentación de datos e información compleja, o de gran volumen (por ejemplo es casi inviable subir un archivo de gran volumen al servidor y que este se procese *in situ* en tiempos razonables)

Habiendo analizado varias de las implicancias generales de una aplicación de escritorio y una web. Procedemos a analizar las necesidades de nuestro cliente. El sistema requiere buena performance en el análisis de archivos de experimentos de gran volumen (del orden de gigabytes cada uno), así sería muy poco eficiente subir un experimento a través de la web y almacenarlo en el servidor web para su análisis, inclusive dicho análisis suele ser muy complejo computacionalmente, así es conveniente utilizar los recursos de almacenamiento y procesamiento presentes en el cliente. Naturalmente los datos tienen una complejidad en su presentación, hay grandes cantidades de puntos de información a mostrar, remitirse a la Figura 3.1-7 donde se muestra un experimento en el cliente para ejemplificar un caso habitual, y esto también debe ser lo más eficiente posible. Generalmente un usuario tiene descargado, de antemano, un conjunto de experimentos sobre los que viene trabajando, con lo que se refuerza el hecho ineficiente de subir un archivo al servidor, pero también se puede pensar en una ejecución offline de la aplicación, al menos parcial ya que como se ilustra en la Figura 3.2-1 con la arquitectura general de la plataforma se hacen uso de varios servicios disponibles a través de la red. El hecho de que el cliente utilice una base de datos embebida hace que la opción web no sea hoy en día viable, no al menos de la manera que la usa el cliente. En este punto del análisis parece ser necesario hacer una pausa y confirmar que la aplicación se realizará de

escritorio y no web. Pero cabe preguntarse cómo se podrían solventar las desventajas de esta elección, según lo planteado párrafos atrás en esta sección.

Dentro del análisis de lenguajes y *frameworks* a utilizar para desarrollar el cliente, Eclipse RCP tomó gran ventaja sobre otras alternativas, ya que direcciona soluciones a los problemas planteados, a través de herramientas integradas en la plataforma base. Analicemos los siguientes puntos:

Portabilidad: Java tiene a la portabilidad como uno de sus principales objetivos. Eclipse se ejecuta sobre la máquina virtual de Java, con lo que parte de la portabilidad está garantizada. Aun así hay elementos que por cuestiones de performance o reusabilidad de componentes (como se ejemplificará en secciones subsiguientes, particularmente con SWT) son provistos nativamente, es decir específicamente para el sistema operativo subyacente. Para estos elementos Eclipse también provee modos y herramientas para definir particularidades (configuración o librerías) de cada sistema operativo aisladamente, en el siguiente punto, sobre la distribución, se ejemplifica esta situación;

Distribución del producto e instalación: Eclipse se distribuye como un paquete totalmente autocontenido y sin necesidad de tener que ser instalado en el sistema (por ejemplo no requiere mantener información en componentes propios del sistema operativo). También admite una configuración y posterior ejecución aislada de otras aplicaciones preinstaladas por el usuario evitando la entrada en conflicto con estas.

Actualización de software: claramente los archivos residen fuera de nuestro alcance. Eclipse propone un esquema de versionado de *plugins* tal que puede chequear contra un servidor, definido especialmente para este fin, y si existe una nueva versión y proceder a la actualización, incluso podría ser automática y desatendida.

Recursos del cliente (PC): con respecto a este punto es interesante notar que las PCs modernas tiene un alto poder de cómputo, esto incluye cada vez más memoria, múltiples procesadores, discos sólidos, etc. El cliente al ser *Desktop*, tiene acceso a esta potencia, de la cual puede hacer uso para preprocesamiento de la información separando la que es relevante de la que no, y luego sí ser esta enviada a los servidores que realmente podrán realizar eficientemente el trabajo más complejo. Es menester recalcar

que el cliente, poniendo en plano ahora a la plataforma Bioplat, no reemplazará los complejos cálculos que son necesarios llevar a cabo para realizar ciertas operaciones como la optimización descripta secciones atrás (ver Optimización de *gene signatures*). Pero sí puede tomar de los experimentos que esté utilizando los genes necesarios según su *gene signature* y enviar un subconjunto de los datos efectivos a procesar, siendo este un preprocesamiento no muy complejo de hacer en el cliente.

3.2.4.3 Arquitectura de una Aplicación Eclipse RCP

Una aplicación desarrollada sobre Eclipse RCP se organiza a través de *plugins* que van enriqueciendo agregando funcionalidad. Esta es la capa de más alto nivel dentro de las distintas capas, que ilustramos en la Figura 3.2-2 y que describimos brevemente a continuación:

- *Plugins*

  Como mencionamos esta es la capa donde se agregan los *plugins* que enriquecen la plataforma. En esta capa es donde insertamos los *plugins* que implementa nuestro cliente Bioplat extendiendo las bases de Eclipse RCP. Interactúa con las capas Eclipse UI, JFace y SWT;

- *Eclipse UI*

  En esta componente se desarrollan las abstracciones conceptuales que presenta la plataforma Eclipse RCP, entre ellas editores para mostrar y editar entidades complejas en el sistema. Vistas utilizadas para mostrar información complementaria, soporte para ayuda contextualizada, etc. Interactúa principalmente con la capas Eclipse *Core* y JFace;

- *Eclipse Core*

  Esta capa define el comportamiento básico de la plataforma Eclipse que no tiene que ver con aspectos visuales, entre ellos podemos describir la estructura de una aplicación (el modelo de clases), *Logging*

básico de la aplicación, estructuras de información de puntos de extensión, etc. Esta capa interactúa con Equinox/OSGi;

- *Equinox/OSGi*

  Como ya se mencionó anteriormente, *Equinox* es la implementación de la especificación OSGi. Esta capa, entonces, implementa el soporte para acceder y administrar los servicios que los distintos *plugins* pueden proveerse entre sí y el dinamismo en la administración de dichos *plugins*. Tiene interacción obviamente con la JRE;

- *JFace*

  Este *framework* provee implementaciones de componentes visuales de alto nivel, como por ejemplo Visores de grillas, *Wizards*, entre otros. Volveremos sobre alguna de estas componentes ya que el cliente Bioplat hace algunos aportes en este sentido. Interactúa con *SWT*, proponiendo un modo conveniente de acceder a ciertos aspectos definidos en esa capa;

- *SWT*

  Este *framework* define los componentes más básicos que pueden encontrarse en cualquier cliente gráfico, como ser botones, *inputs* para entrada de texto, diálogos a su nivel más básico (conocidos como *Shells*), etc. Esta componente/capa interactúa con la capa JRE y con la de recursos nativos propios del y para el sistema operativo (sobre esto último, comentaremos detalles en la sección Toolkit UI);

- JRE

  Esta capa es la implementación más básica del lenguaje de programación Java. Aquí se implementan todos los detalles del lenguaje de programación que permiten su ejecución. La implementación de esta capa está regida por una especificación que estandariza las distintas implementaciones. Representa a la máquina virtual del *runtime* de Java.

- Sistema operativo (SWT)

  Esta capa es accedida desde la que denominamos SWT y provee tanto código Java específico para distintos sistemas operativos, como librerías específicas (por ejemplo librerías .so en Linux) que son accedidas a través de JNI (*Java Native Interface*), como se comentará con mayor detalle más adelante. También puede incluir cualquier otro recurso que sea específico del sistema operativo;

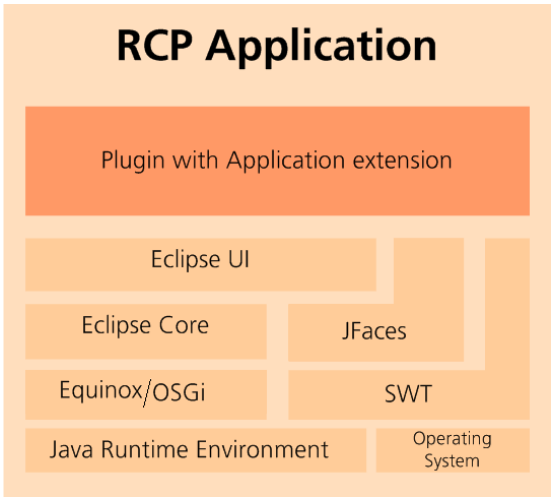


Figura 3.2-2 - Stack de una aplicación Eclipse RCP

3.2.4.4 Toolkit UI

Los usuarios esperan ver interfaces gráficas atractivas, modernas, que se integren naturalmente al sistema operativo, que respeten los aspectos y usos que definen los distintos sistemas operativos, estos aspectos van desde la forma de los botones, inputs de texto, hasta atajos de teclado y comportamiento del *mouse* (*drag&drop*) que el usuario ya viene acostumbrado a usar por las aplicaciones nativas. Así son deseables interfaces de usuario visualmente ricas, en el sentido de que se provean todas estas características eficientemente.

El desarrollo de librerías gráficas en Java históricamente persiguió dos objetivos, por un lado no perder la portabilidad propuesta por el lenguaje, y por el otro la integración como aplicación nativa en el sistema operativo. El primero de los objetivos

llevaba a tener que implementar desde cero los componentes visuales, de ahora en más *widgets*, perdiendo performance y no logrando un aspecto totalmente nativo de los mismos, haciendo incluso muy costosa su implementación, y viceversa si se encaraba el problema desde la perspectiva opuesta, es decir, buscando ''natividad'' y así performance se cedía portabilidad. Así es que luego de un recorrido por distintas estrategias de desarrollo de estas librerías de *widgets* que cumplían con una de las dos características buscadas, se arribó a SWT (*Standard Widget Toolkit*). Con el diseño de SWT se logró conseguir estas dos características sin perder las características mencionadas de eficiencia o riqueza visual como les pasaba a otras librerías gráficas, como por ejemplo AWT, y en menor medida a Swing, que da cierto soporte a cada sistema operativo [7]). Brevemente la implementación de SWT, provista como un subproyecto de Eclipse RCP, hace uso intensivo de una característica importante de Java que es su integración con el lenguaje de programación C para proveer implementaciones nativas utilizando JNI *(Java Native Interface)*. El funcionamiento básico de este mecanismo es que en Java puro se define el contrato (firma o *signature*) de una función, pero su implementación se realiza en C, que puede interactuar y lidiar con los detalles de cada sistema operativo, luego en la compilación se asocian, permitiendo su funcionamiento en conjunto; los detalles del mecanismo de implementación en código nativo exceden el alcance de este texto, se puede remitir a [4] para más detalles. Así, SWT define por un lado clases y métodos generales, normalizados, es decir independientes de los sistemas operativos e implementados puramente en Java, luego se proveen distintas implementaciones también en Java para cada sistema operativo de dichas clases pero mapeando las funciones de grano más fino a funciones nativas implementadas como ya se mencionó, utilizando el lenguaje de programación C. De este modo, se consigue una implementación portable (suministrando para cada sistema operativo su implementación nativa correspondiente y complementaria) y muy eficiente ya que se delega el comportamiento a invocaciones del sistema operativo, a la vez que la implementación de "nivel intermedio" se hace en Java, pudiendo realizar *debugging* ante errores más fácilmente o aislar problemas para el tratamiento particular en cada sistema operativo. [8]

Una vez definida esta estrategia de implementación base. Se fueron desarrollando otras capas de funcionalidad, como vimos en la sección Arquitectura de una Aplicación Eclipse RCP, aquí aparece un nuevo *framework* visual que agrega estructuras de más alto nivel sobre *SWT*, pero sin ocultarlo, es decir lo complementa. Este *framework* se llama *JFace* y define nuevos componentes visuales, extendiendo y utilizando

los de SWT, entre estos podemos encontrar cuadros de diálogo, mecanismos para el reporte de avance de operaciones de larga duración, visores más ricos para grillas, etc. *JFace,* adicionalmente, también provee el modelo de *data binding*[37] utilizado en el cliente Bioplat, dando soporte para permitir la sincronización de distintas propiedades de distintos objetos, muy provechoso por ejemplo para mantener sincronizada la vista, el aspecto visual de un dato, y su modelo. [6]

Eclipse agrega a esta estructura su ''*Workbench mod*el'', que suma organización a las componentes definidas por *JFace*, agregando sus propias entidades, entre ellas, editores, vistas, extensibilidad basada en contribuciones, acciones, etc.

Habiendo hecho esta introducción volvemos al cliente desarrollado, uniendo los conceptos propios funcionales con los comentados en los últimos párrafos.

Se utilizan editores para la visualización y edición general de los *gene signatures* y experimentos Bioplat. Así se define el marco general de trabajo para estas entidades, para ejemplificar este hecho diremos que un editor al estar activo agrega su propio menú de operaciones, y habilita según corresponda un subconjunto de estas operaciones en una barra de herramientas o *toolbar* para un acceso más rápido, ejemplo de esto lo vemos en la Figura 3.1-3.

Las vistas se utilizan, primeramente, para mostrar información complementaria. Así en el cliente se definieron dos vistas principales aunque con distinto nivel de importancia. Una meramente informativa de lo que va haciendo el sistema, donde se detallan también los errores y los mismos pueden ser reportados. Los mensajes en esta vista se cargan a través de un mecanismo programático provisto por la arquitectura del cliente Bioplat (vemos un ejemplo de uso en el Código 3.2—3). Y la otra es la vista de genes, que muestra información complementaria de un gen seleccionado en una entidad, integrándose con distintos sitios como ya se mencionó anteriormente en el texto (ver sección Vista de genes)

Cómo se vimos en la figura Figura 3.2-2, Eclipse define una *stack*, o conjunto de capas interrelacionadas, para desarrollar interfaces visuales potentes y nativas, dando un aspecto y comportamiento (*look & feel*) muy beneficioso. Pero Eclipse lejos de quedarse

[37] https://wiki.eclipse.org/JFace_Data_Binding_Introduction

en aspectos visuales potentes, define todo el marco de trabajo desde el desarrollo, utilizando Eclipse SDK IDE que es en sí mismo una aplicación construida sobre su propia infraestructura con el agregado de todos los *plugins* necesarios para la edición de código, compilación, ejecución, empaquetamiento de la aplicación y hasta su posterior publicación como producto. Así Eclipse como arquitectura donde cimentar los desarrollos propios es vista esencialmente como un *middleware* que permite enfocar el desarrollo de la aplicación en la implementación propia de las componentes del dominio, abstrayéndose de toda la infraestructura necesaria para desarrollar una aplicación de esta complejidad. [6]

Siguiendo en línea con este aspecto, más ligado al ciclo de vida del proyecto, es decir su empaquetamiento, publicación, etc. es que utilizamos *Maven* para facilitar y automatizar ciertas tareas que esta herramienta tiene como objetivo, entre ellas hacer el todo el ciclo de desarrollo más simple y, justamente, automatizado, esto incluye desde especificación de dependencias del proyecto, integración y configuración automática con el IDE, empaquetamiento, hasta publicación del cliente Bioplat.

La administración de los *plugins* Eclipse se realiza a través de Maven y el proyecto *Tycho*[38], que provee extensiones para que *Maven* "interactúe" con Eclipse. Estas extensiones permiten simplificar considerablemente el manejo de los artefactos que componen a una aplicación Eclipse RCP, como son los mencionados *plugins o bundles* y sus dependencias, los repositorios de publicaciones, entre otros. Cabe mencionar que el Modelo de la aplicación es una dependencia más para el Cliente Bioplat RCP. *Maven Tycho* arma el *bundle* del Modelo[39] automáticamente y se maneja como una componente más del cliente, del cual luego dependen los demás *plugins* que implementan la vista de genes, los editores de gene signatures, experimentos, y lineamientos básicos de los aspectos tanto visuales como estructurales, como veremos en la sección Framework de desarrollo. Estos plugins propios del cliente, también se administran utilizando Tycho.

Es importante notar que a pesar de delegar el manejo de los artefactos Eclipse a *Maven Tycho*, sigue siendo posible utilizar por completo el entorno de trabajo Eclipse,

[38] https://eclipse.org/tycho/

[39] *Bundle* en términos de OSGI, ya que esta componente no es un *plugin* Eclipse. Aun así permite una excelente integración con el resto de los plugins.

llamado *Plug-in Development Environment* (PDE)[40], aprovechando todas las herramientas que provee: administración de *plugins*, testeo, configuración, entre otras. PDE también trae soporte para desarrollo de componentes OSGi, recordemos que Eclipse, a través de Equinox implementa e integra los conceptos ya vistos de esta especificación.

El cliente Bioplat está definido como un producto Eclipse RCP, que es donde se centraliza y define todo lo que lo conforma:

- imágenes para íconos, *splash screen*, etc;
- modo de iniciar la aplicación con parámetros según el sistema operativo;
- plugins y/o *eclipse-features* (conjunto de plugins interrelacionados, bajo un mismo nombre/concepto) que componen la aplicación;
- configuración propia para cada sistema operativo (por ejemplo parámetros de contexto/ambiente especiales)

Otro aspecto importante del PDE, es el empaquetamiento de la aplicación, tomando como entrada la definición del producto mencionado. Da soporte para que el cliente se puede empaquetar y distribuir para los distintos sistemas operativos. Es importante recalcar nuevamente que Maven a través de Tycho permite integrar perfectamente con estas herramientas incluso dando soporte para su automatización.

[40] https://www.eclipse.org/pde/

3.2.4.5 *Framework* de desarrollo

Vale la pena volver a recordar en este punto que la idea, y uno de los objetivos, del cliente es que pueda ser extendido fácilmente. Seguiremos recorriendo el uso de los puntos de extensión definidos por Eclipse, para dar soporte al enriquecimiento del cliente y así de la plataforma, sin embargo esta facilidad no estaría completa si ante una actualización de la plataforma base, por ejemplo actualizando la arquitectura de Eclipse, se tuvieran que reimplementar (migrando o adaptado) todas las extensiones que se pudieran haber realizado. Por esta razón, del mismo modo que se diseñaron puntos de extensión, protocolos a respetar para extender de manera correcta el cliente, también es necesario abstraer de la mejor manera posible la creación de widgets, o componentes visuales, y que todas las operaciones se vieran respetando ciertos usos, intentando lograr que el desarrollo en el cliente Bioplat, mantuviera una coherencia en los lineamientos definidos.

Operaciones dinámicas

Las entidades definidas en la plataforma, tal como los experimentos Bioplat, como los *gene signatures,* son objeto de aplicación de operaciones. La plataforma ofrece un conjunto de estas operaciones como vimos tanto en Operaciones básicas *para gene signatures,* como para los experimentos en una sección similar para sus Operaciones básicas. Si bien estas operaciones son provistas como parte del desarrollo de la plataforma, es posible y es uno de los objetivos de este trabajo, proponer una solución que permita a otros desarrolladores implementar operaciones propias, según las necesidades específicas que se tenga. Para dar este soporte, se utilizaron y definieron puntos de extensión de Eclipse, los cuales fueron introducidos anteriormente.

Eclipse define sus propios puntos de extensión, y el modo de definir nuevos puntos de extensión. Los puntos de extensión son un conjunto de definiciones declarativas en un archivo descriptor llamado *plugin.xml*, y la confección de código, si bien esto último es opcional, generalmente es necesario proveer un desarrollo propio a

través de un conjunto de clases implementadas bajo ciertas normas (por ejemplo implementando una *interface* Java, suministrada para este fin).

Hecha esta introducción complementaria, se pasará a analizar la extensión de operaciones para entidades del sistema, de esta manera, también, se analizará el funcionamiento específico de los puntos de extensión, ilustrando sus características adaptadas al cliente Bioplat.

Para comenzar, se define la declaración del punto de extensión, como se Ilustra en el extracto de Código 3.2—2.

```
1.  <extension point="edu.unlp.medicine.bioplat.rcp.editor.operation.contribution">
2.    <actionContribution
3.          caption="Add gene 1"
4.          class="edu.unlp.medicine.bioplat.rcp.ui.lab.acions.AddGene1"
5.          groupId="edit"
6.          image="resources/icons/addGene1.png"
7.          onMenu="false"
8.          onToolbar="true">
9.      <selection class="edu.unlp.medicine.entity.biomarker.Biomarker" />
10.   </actionContribution>
11. </extension>
```

Código 3.2—2 - Extracto de plugin.xml

Del estracto del archivo descriptor de extensiones *plugin.xml*, se pueden hacer los siguientes comentarios:

- En la línea 1 se referencia al punto extensión sobre el cual se quiere hacer la contribución. Dentro de este punto de extensión se admiten los elementos actionContribution, que es el tag que definimos para configurar nuestro punto de extensión
- El tag <actionContribution>, admite los siguientes atributos:
  - *caption*: Indica el nombre a mostrar de la acción;
  - *class*: Indica el nombre completo de la clase que implementará la acción. Para que esta clase sea válida debe implementar la *interface* ActionContribution<AbstractEntity> o bien de manera conveniente extender la clase AbstractActionContribution<AbstractEntity> ambas provistas como parte de la arquitectura del cliente;

- *groupId*: indica el id del grupo donde se agrupará la acción (en la Figura 3.1-3, se muestra un ejemplo visual de agrupamiento de acciones)
- *image*: Indica la imagen con la cual se identificará a la acción en el cliente;
- *onMenu* y *onToolbar*: repectivamente indica la presencia o no en el menú principal de la applicación y/o en el área de herramientas
- Por último dentro de este elemento, podemos agregar el elemento <*selection*>. Este tiene como único atributo la indicación de la clase sobre la cual la operación se activará, esto es ante la presencia un *gene signature* seleccionado o activo, según este ejemplo, la operación aparecerá disponible de aplicar.

Esta extensión define y requiere de una clase que implemente el comportamiento deseado, como se observa en el atributo *class* del elemento <*actionContribution*> y se muestra a continuación:

```
1.  package edu.unlp.medicine.bioplat.rcp.ui.lab.acions;
2.
3.  import edu.unlp.medicine.bioplat.rcp.ui.entities.editors.contributors.\
    AbstractActionContribution;
4.  import edu.unlp.medicine.bioplat.rcp.ui.views.messages.Message;
5.  import edu.unlp.medicine.bioplat.rcp.ui.views.messages.MessageManager;
6.
7.  import edu.unlp.medicine.domainLogic.framework.MetaPlat;
8.  import edu.unlp.medicine.entity.biomarker.Biomarker;
9.  import edu.unlp.medicine.entity.gene.Gene;
10.
11. public class AddGene1 extends AbstractActionContribution<Biomarker>{
12.
13.   @Override
14.   public void run() {
15.
16.     Biomarker biomarker = model();
17.     Gene gene1 = findGene("1");
18.     biomarker.addGene(gene1);
19.
20.     MessageManager.INSTANCE
21.         .add(Message.info(gene1 + " added to " + biomarker));
22.   }
23.
24.   private Gene findGene(String id) {
25.     return MetaPlat.getInstance().findGene(id);
26.   }
27. }
```

Código 3.2—3 - Clase para agregar el gen 1

Algunas anotaciones sobre la clase AddGene1, ilustrando detalles sobre lo provisto por la arquitectura disponible en el cliente, algunas de sus características y su modo de interacción con algunas componentes del sistema:

1. La clase que implementa la operación extiende una clase abstracta provista por la arquitectura básica del cliente Bioplat. Esa es la forma conveniente de crear la extensión, también puede implementar una *interface* correspondiente, pero es posible dar un comportamiento por *default*, suministrando algunas implementaciones predefinidas;

2. Al extender de la clase provista, heredamos comportamiento útil y necesario para la implementación que buscamos. Así solo es obligatorio implementar el método *run()* que es el que realizará la acción cuando corresponda. En la clase de ejemplo, que vemos en Código 3.2—3, en la línea 16 se muestra la invocación al método *model()*, el cual representará al objeto del modelo que se esté editando. Cabe recordar que estas operaciones se habilitan cuando hay un editor activo y este tiene detrás su modelo, su entidad del sistema. Siguiendo el ejemplo se muestra que en este caso representa a un *Biomarcador*;

3. Se puede utilizar toda la API provista por el Modelo. En el ejemplo se accede al objeto *MetaPlat* para buscar un gen (línea 25 en Código 3.2—3), el que luego es agregado al biomarcador, a través de los mensajes disponibles en esta entidad (línea 18 en Código 3.2—3). Es importante remarcar aquí que con la infraestructura de *databinding* se mantiene sincronizado el modelo con la vista [6], con lo que ese agregado del gen al biomarcador es reflejado automática y transparentemente en la o las correspondientes vistas (widgets visuales) que hagan referencia a ese biomarcador. En la sección Widgets volveremos sobre la implementación de esta sincronización;

4. Por último, se muestra cómo se puede interactuar con la vista de mensajes del sistema para dar *feedback* de la operación o su resultado, como se aprecia en el ejemplo, en las líneas 20 y 21. (A modo de ejemplo ilustramos la vista de mensajes en la Figura 3.1-7)

Como venimos viendo en esta sección la definición de la extensión de nuestro punto de extensión es simplemente proporcionar su declaración a través de una porción de *xml*. Esta sección de *tags xml* está perfectamente definida utilizando un archivo *schema*[41], en el cual se describe formalmente su estructura. Y es obviamente definido por el proveedor del punto de extensión, indicando los atributos que va a ser necesario definir, para hacer efectiva la extensión. Gracias a la definición del *schema*, Eclipse puede chequear que la definición sea correcta al momento en que esta es justamente defina, por ejemplo puede chequear que si un atributo espera una clase, esta exista; o que, de la misma manera, si tiene que implementar una *interface* particular, la clase que se configura efectivamente lo haga. De esta manera se permite validar estáticamente la configuración. En este mismo sentido es posible chequear otro tipo de datos para atributos, como puede ser valores booleanos, numéricos.

Lo explicado hasta aquí en esta sección representa todo lo que tiene que ver con la declaración y eventual implementación puntual que el punto de extensión requiera. Siendo esto todo lo necesario para extender las operaciones de una entidad. Cabe mencionar que todas las operaciones definidas tanto para Experimentos Bioplat como para *Gene signatures*, referenciadas en las secciones correspondientes, se realizaron, se contribuyeron, haciendo uso de este punto de extensión.

Implementación del punto de extensión

A continuación, se expone la implementación del punto de extensión *''edu.medicine.bioplat.rcp.editor.operation.contribution''*. Es decir la implementación que busca las definiciones entre los *plugins* del cliente, las procesa y crea las operaciones para las entidades según corresponda. Para esto Eclipse provee una API que permite acceder a todas las declaraciones de puntos existentes, permitiendo filtrar en particular las que nos interese procesar. Conviene hacer notar que esta es la implementación del procesamiento del punto de extensión, cuando alguien interesado en extender el cliente, quiera hacer una contribución a través de este punto, le será totalmente transparente como es el procesamiento más interno.

[41] http://www.w3schools.com/xml/schema_intro.asp

```
1.  /**
2.   * Lee los extension points que se hayan registrado y los procesa
3.   */
4.  private List<ActionContribution<T>> readActionsByExtensionPoint() {
5.    final List<ActionContribution<T>> actions = Lists.newArrayList();
6.    ExtensionPointLoader.create("edu.medicine.bioplat.rcp.editor.operation.contribution")
7.        .load(new DefaultExtensionLoaderRunnable() {
8.          @Override
9.          protected void runOn(IConfigurationElement configurationElement)
10.                                                                  throws Exception {
11.           if (shouldAdd(configurationElement))
12.             actions.add(createAndConfigureContribution(configurationElement));
13.         }
14.
15.         private boolean shouldAdd(IConfigurationElement celement)
16.                                                       throws ClassNotFoundException {
17.           // tiene 1 o ningún elemento. es opcional,
18.           // en caso de no estar seteada va para cualquier entidad
19.           IConfigurationElement[] selectionHolder = celement.getChildren("selection");
20.           String selectionClassName = (ok(selectionHolder) ?
21.                       extractClassName(selectionHolder):defaultSelectionClassName());
22.
23.           final Class<?> configuredSelectionClass = Class.forName(selectionClassName);
24.           return configuredSelectionClass.isAssignableFrom(getSelectionType());
25.         }
26.
27.         private String extractClassName(IConfigurationElement[] selectionHolder) {
28.           return selectionHolder[0].getAttribute("class");
29.         }
30.
31.         private boolean ok(IConfigurationElement[] selectionHolder) {
32.           return selectionHolder.length != 0;
33.         }
34.
35.         private String defaultSelectionClassName() {
36.           return Object.class.getName();
37.         }
38.
39.         private ActionContribution<T> createAndConfigureContribution(
40.             IConfigurationElement celement) throws CoreException {
41.           // TODO evitar warning
42.           return ((ActionContribution<T>) celement.createExecutableExtension("class"))
43.             .modelProvider(AbstractEditorActionBarContributor.this)
44.             .caption(celement.getAttribute("caption"))
45.             .image(Activator.imageDescriptorFromPlugin(celement.getAttribute("image")))
46.             .onMenu(Boolean.valueOf(celement.getAttribute("onMenu")))
47.             .onToolbar(Boolean.valueOf(celement.getAttribute("onToolbar")))
48.             .group(celement.getAttribute("groupId"));
49.         }
50.
51.       });
52.   return actions;
53. }
```

Código 3.2—4 - Procesamiento de puntos de extensión

Algunos apuntes de la función presentada en el Código 3.2—4, que procesa efectivamente el punto de extensión que mencionamos:

1. La función se encarga de procesar todos los puntos de extensión presentes en todos los plugins del cliente;

2. Vemos que la búsqueda de las definiciones de nuestro punto de extensión se hace a través de la API definida por Eclipse RCP, vemos el uso de la clase *ExtensionPointLoader* en la línea 6. Creando una nueva instancia para nuestro punto de extensión, y posteriormente haciendo la invocación del método load en la línea 7. Este método espera un objeto de tipo *ExtensionLoaderRunnable* que define finalmente como procesar la contribución al punto de extensión, en nuestro ejemplo utilizamos una clase abstracta que provee comportamiento común, y requiere implementar el método *runOn*, que crea los objetos que hacen a las operaciones en base a cada declaración de extensión;

3. La implementación del método runOn entonces, define si corresponde crear la operación, y la crea o no. La determinación de si la debe agregar depende del *tag* <*selection*>, a modo de ejemplo del uso de este *tag* remitirse a la extracto de Código 3.2—2;

4. En la línea 39 definimos el método `createAndConfigureContribution`, que recibe una instancia de la configuración de la extensión que nos interesa (según punto 2). En la línea 42 de este método creamos un ActionContribution mediante la invocación al método `createExecutableExtension`, que crea una instancia de la class definida en la extensión. Notar que no sabemos la implementación concreta, pero sí sabemos que esta clase implementa ActionContribution ya que esto se indicó en la definición que hicimos del punto de extensión (uso de *schema* para validar, como se mencionó en la primera parte de esta misma sección);

5. Una vez tenemos la instancia de ActionContribution, procedemos a configurarla obteniendo la información desde la declaración de la extensión. Esto se ve en el código desde las líneas 43 a 48.

Preferencias

En esta sección mencionaremos los usos *ad hoc* que se hacen de algunas extensiones provistas por Eclipse RCP. Entre estas extensiones mencionaremos la posibilidad de indicar preferencias, y dentro de ella su utilización para la integración de sitios de terceros que se integrarán a la vista de genes (ver sección Vista de genes). El cliente ofrece la configuración personalizada de ciertas preferencias del sistema. Entre ellas:

- Configuración de *proxies* para acceso a Internet: Es común el caso en que el cliente ejecuta la aplicación detrás de un *proxy web*, desde aquí se da la posibilidad de que lo tome automáticamente desde el sistema o que sea configurado manualmente; recordamos que desde el cliente tenemos accesos a través de internet, como por ejemplo la vista de genes (ver sección Vista de genes) y los accesos a los servicios de R (ver sección Servidor R como servicio)

- Configuración propia de visualización de las entidades *gene signatures* y experimentos *Bioplat*. Por ejemplo, dado el gran volumen de *samples* y genes presentes en un experimento, es posible que el usuario seleccione cuántos desea que el cliente cargue para visualizar, es importante recalcar que esto impacta solo en lo que tiene que ver con la visualización, en el modelo se opera con el conjunto total de datos;

- Urls extensibles y parametrizables que integran al cliente a través de la vista de genes. Se ilustra en la Figura 3.2-3. Este punto de extensión es provisto por el cliente, para permitir la integración con sitios externos que sirven de complemento a la información que se visualiza en el cliente. Las mencionadas *urls* permiten la parametrización del gen que se desea consultar, el mismo es provisto por la selección en el editor de la entidad correspondiente (ver sección Vista de genes)

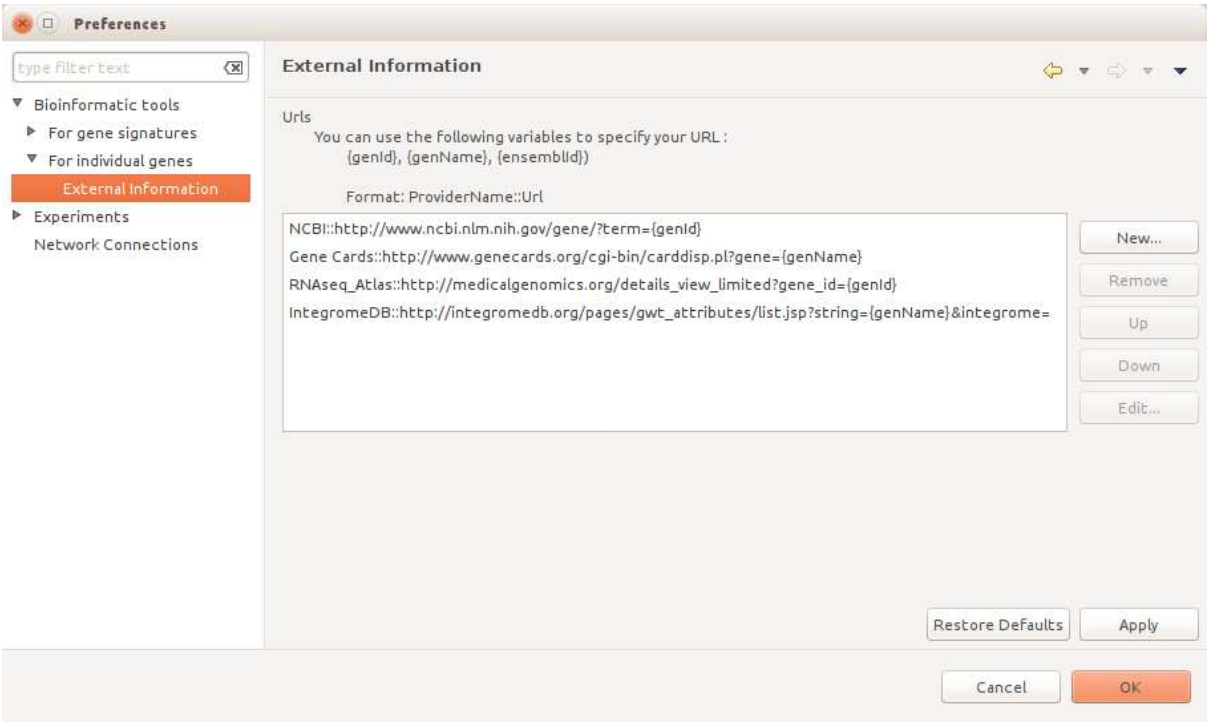


Figura 3.2-3 - Configuración de urls de integración para Vista de genes

Para indicar estas preferencias Eclipse RCP define su propio punto de extensión, llamado *''org.eclipse.ui.preferencePages''*, donde se debe suministrar nuestra implementación de la página de preferencias, luego Eclipse se ocupa de mostrarla dentro del diálogo correspondiente, es decir de procesarlo. Es de notar la potencia y flexibilidad en la definición de extensiones a través de este mecanismo, indicando que estas pueden servir para cualquier índole teniendo siempre una base consistente.

Widgets

Remitiéndonos a las figuras expuestas a lo largo de la sección La Plataforma desde el cliente, vemos un uso intensivo de campos simples de introducción de texto, listas de selección, grillas, etc. con lo cual se proveen utilidades que permiten una interacción de alto nivel, donde agrupamos comportamientos comunes en cualquier desarrollo de componentes visuales (poner un texto precediendo un campo de entrada de texto indicando su título, ordenar y alinear los widgets, etc.)

Para lograr estos objetivos se crearon clases que ''decoran'', es decir se ponen como intermediarias de, las componentes propias de *JFace/SWT* y se crearon clases ''*Factories*'' que se utilizan para crear estos decoradores y así ocultar los objetos propios. Hacia el desarrollador se devuelve un objeto intermediario que justamente abstrae y oculta los detalles más internos. Entre estos detalles internos se incluye estado, pero también la configuración *standard* de aspectos visuales, la configuración de *databinding* [6], registro de *listeners* de eventos para comportamientos internos (ejemplo de esto es que al eliminar los objetos, se eliminen también los elementos que este haya creado, liberando recursos; la ''desregistración'' de los *bindings* registrados como soporte al *databinding*). Este objeto intermediario, al que llamaremos Widget, será la API con la cual el desarrollador interactuará para manipular los objetos visuales del cliente. Esta API fue desarrollada siguiendo la modalidad *fluent*[42]. Intentando definir, finalmente, un DSL de configuración y manipulación de los *widgets* en el cliente Bioplat.

Ilustramos en el Gráfico UML 1 el diagrama de secuencia para la creación de un *widget* de entrada de texto. La clase *Widgets* funciona como ''*Factory method*'' [9] de los distintos *widgets*. La clase *CText* es el decorador de *org.eclipse.swt.widgets.Text*, que es el *widget* real de SWT. Notar la relación de composición entre los objetos. El objeto *dataBindingContextManager,* que vemos en el mismo diagrama, administra el contexto de *databindings* a nivel global, es un *singleton* [9]

[42] http://martinfowler.com/bliki/FluentInterface.html

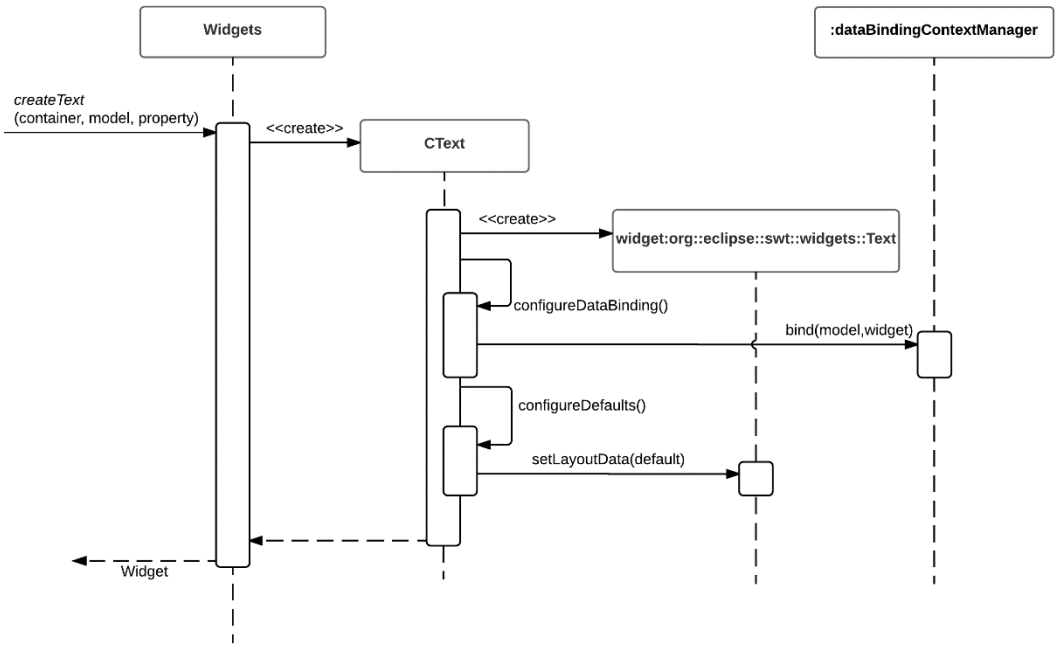


Gráfico UML 1 - Diagrama de secuencia de creación de widget

Extracto de código de ejemplo de uso de la API para crear e interactuar con los widgets visuales dentro del cliente:

```
1. Widgets.createText(container, aBiomarker, "name")
2.         .setLayoutData(aLayoutData)
3.         .readOnly();
```

Código 3.2—5 - Creación simple de widget

Donde, *container* es el objeto que contiene a los widgets (un objeto *Composite* en SWT); *aLayoutData* es como se dispondrá el objeto visual sobre su *container* (por ejemplo si estará centrado o alineado a otro widget, etc). readOnly() hace al widget de solo lectura. Notar el encadenamiento de llamadas, dadas por las *interfaces* fluídas antes mencionadas. Cabe destacar que al interactuar de esta manera, el desarrollador se abstrae de la creación y configuración de los objetos, pudiendo estos cambiar internamente sin afectar el código cliente.

Otra componente visual de la que se hace uso intensivo en la aplicación son las grillas. Estas muestran la información tabulada y permite hacer operaciones sobre los datos que representa. En el cliente estas grillas, muestran gran volumen de información

con lo cual fue necesario optimizarlas. Una de las optimizaciones fue hacer que se muestren de forma paginada de manera que las mismas se van cargando a medida que son requeridas, haciendo un uso a eficiente de la memoria.

Para el diseño de esta extensión, se siguieron los lineamientos planteados en el diseño de los widgets más básicos comentados anteriormente en esta misma sección. Esto incluye nuevamente una API fluída para dar mejor dinamismo al desarrollo y una abstracción de los widgets visuales utilizados a ''bajo nivel''. También soporta *databinding* para la sincronización de la vista con una colección de datos. En el recuadro Código 3.2—6 se muestra un extracto de código a modo de ejemplo de la creación de un componente visual grilla

```
1. final TableBuilder tableBuilder = TableBuilder.create(container);
2.
3. tableBuilder
4.    .model(biomarker, "genes")
5.    .showSelectionColumn()
6.    .addColumn(
7.                ColumnBuilder.create()
8.                             .title(Messages.GENE_NAME)
9.                             .centered()
10.                            .property("name"))
11.   .addColumn(
12.               ColumnBuilder.create()
13.                            .property("description")
14.                            .title(Messages.GENE_DESCRIPTION)
15.                            .rightAligned()
16.                            .width(800));
17.
18. TableReference table = tableBuilder.build();
19.
20. table.addSelectionChangeListener(this);
21.
22.
```

Código 3.2—6 - Código de creación de grillas

Algunos comentarios sobre las características del código.

1. Se utiliza un objeto que administrará la construcción de la grilla una vez esta se haya terminado de configurar, este objeto es representado por *TableBuilder*;

2. Al *tableBuilder* se le agregan las columnas a construir, el modelo a observar y en qué propiedad, esta propiedad debe representar lógicamente a una colección de objetos;

3. La columnas observan las propiedades de cada uno de los elementos de la colección que se configuró en el *tableBuilder*;

4. Es de destacar el encadenamiento de llamadas a los métodos que van configurando cada uno de los objetos. Estos no tiene un orden prefijado y en su mayoría cuentan con valores por *default* o se configuran en función de los datos que mostrará, por ejemplo, si se va a mostrar un número este se alineará automáticamente a la derecha;

5. Una vez configurados todos los parámetros para la creación, se arma la grilla con el método *build()*. Este método retorna una abstracción de la grilla, que permite manipularla agregándole por ejemplo un *listener,* para tener notificaciones de cambio de elemento seleccionado, retornar los objetos seleccionados, etc. Todos métodos que tienen sentido luego de que la grilla está instanciada;

6. Por último se menciona que puede pasar que como caso excepcional se requiera acceder a la grilla básica de *SWT*, ya que se puede querer implementar un comportamiento que no está soportado por la arquitectura que se provee, en ese momento, desde el cliente. Así se permite accederla. Sin embargo la idea es proveer un conjunto de objetos que simplifiquen la interacción con componentes básicas y presenten una alternativa más conveniente que la alternativa de "bajo nivel". Esta estrategia es la misma que adopta *JFace* con respecto a *SWT*.

Wizards

Habiendo detallado algunas de las utilidades que provee la arquitectura en el cliente, se darán a continuación otras características presentes en el cliente para una apropiada extensión.

La operación que se describió en el ejemplo Código 3.2—3, páginas atrás no requirió mayores complejidades, pero en general es necesario presentar al usuario pasos (diálogos) donde va configurando parámetros según las necesidades de la operación que vaya a realizar, por ejemplo si es necesario seleccionar un experimento, el cliente deberá proporcionar un diálogo acorde a tal fin, y si luego de seleccionado es necesario aplicar alguna operación complementaria como filtro se abrirá otro diálogo para esta tarea. Estas interfaces son más complejas claramente, ya que incluyen además validaciones de consistencias de configuración, habilitación de paso siguiente, si es posible terminar la configuración y disparar la operación propiamente dicha, etc. Esto forma un conjunto de diálogos ordenados. Este esquema de diálogos interrelacionados se conoce como *Wizards*. Eclipse, a través de JFace, provee una implementación *standard* muy potente que fue extendida para adaptarse a las necesidades propias y comunes que requiere este cliente.

A modo de *framework* se extendió la estructura de clases definida por Eclipse para alcanzar los siguientes objetivos:

- *facilidad de uso*

  Se construyeron abstracciones para simplificar los parámetros que pide el modelo de *wizards* standard

- *comportamiento extendido*

  Todos los *wizard* comparten un comportamiento común básico al que se le da soporte (por ejemplo la creación del diálogo que contendrá las páginas del *wizard*). Además se definió un marco de trabajo genérico para proveer validaciones sobre los datos ingresados, sobre el estado del *wizard*, etc. Para esto se hizo uso, nuevamente, del componente *databinding* de manera de proveer estas funcionalidades a todos los wizards. [6]

  Otro comportamiento requerido por el cliente es el cálculo de pasos como resultado de pasos previos, que no es provisto explícitamente por el *framework* standard de wizards de Eclipse.

- Ocultamiento de implementación

  No es necesario llegar a los detalles de implementación que muchas veces complican la configuración e implementación de un *wizard*. Estos son

ocultados o en muchos casos adaptados para un mejor entendimiento del desarrollador que implemente uno, a través de la definición de interfaces de más alto nivel de abstracción.

La idea es despegarse de la implementación de Eclipse/*JFace*, y proveer un conjunto de objetos más conciso, mejor adaptado a los lineamientos definidos para interactuar con el cliente Bioplat. Esto permite independizar a los objetos utilizados por Eclipse para la construcción de *Wizards* en este caso, de los objetos que utiliza un desarrollador en el cliente. Así una actualización del cliente a nivel de Eclipse es mejor controlada. El modelo básico de esta implementación se ilustra en el diagrama de clases del Gráfico UML 2.

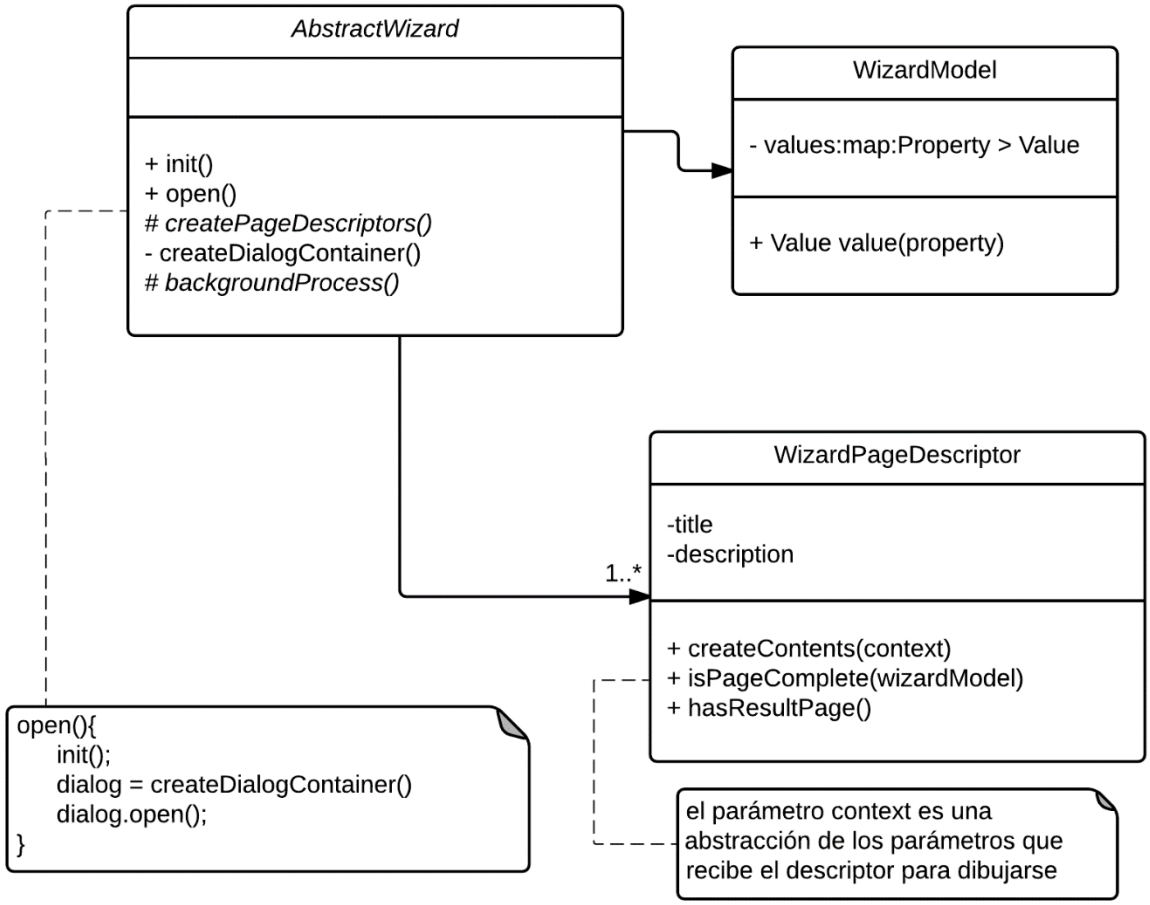


Gráfico UML 2 - Modelo de clases simplificado para Wizards

El diagrama de secuencia ilustrado en el *Gráfico UML* 3 muestra el intercambio de mensajes para la creación de un *wizard*, y la posterior creación de sus páginas. Es un diagrama simplificado, donde indicamos únicamente la dinámica de creación y apertura.

En el diagrama vemos que como parte de la inicialización del *Wizard,* se crean los *descriptors, (una colección de WizardPageDescripor)* a través de la invocación al método *createPageDescriptors*, el cual se deja sin implementación (abstracto en Java) en AbstractWizard para que cada clase concreta proporcione su propia implementación, es decir, su propio conjunto de páginas en este caso sus descriptores. Una vez estos se crearon, se dispara la creación de los *widgets* visuales que representan a estas páginas, las cuales son extensiones hechas en la plataforma. Una vez creado el diálogo que contiene las páginas del *wizard*, el usuario comienza a interactuar con ellas, a medida que lo va haciendo, es decir va configurando los distintos parámetros, el *framework* va disparando eventos que llegan las páginas del *wizard*, que a su vez los va delegando en los objetos *descriptors, creados y mencionados anteriormente,* que con un contexto dado saben determinar si la página está completa, pudiendo pasar a la siguiente o incluso se puede terminar la configuración via *wizard* y, finalmente, disparar la operación.

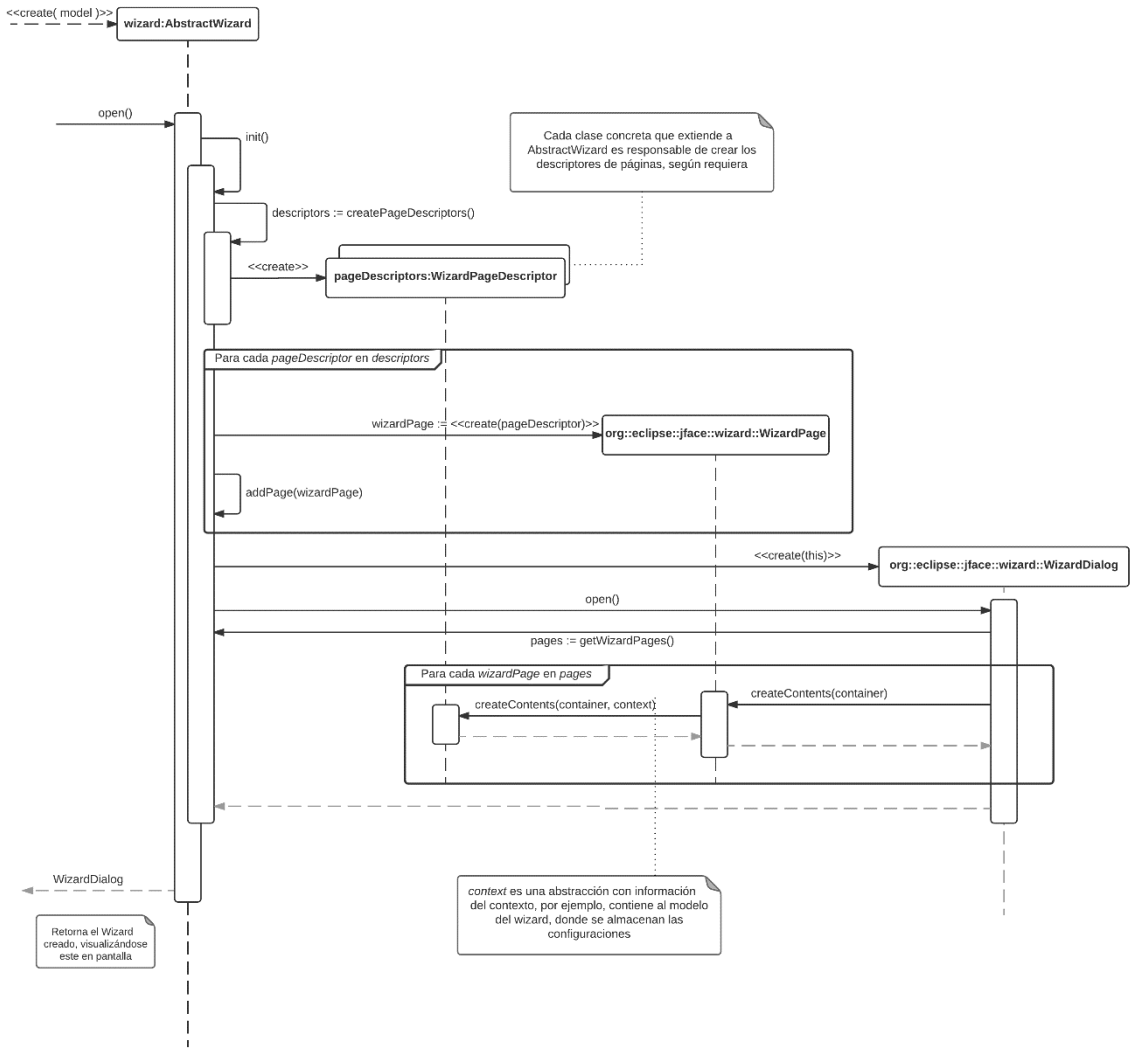


Gráfico UML 3 - Diagrama de secuencia de creación de wizards (simplificado)

Esta clase *AbstractWizard* implementa el patrón de diseño ''*Template Method*'' [9] también para el método *backgroundProcess()*, es decir la implementación de este método lo completarán las clases concretas que extiendan nuestro *AbstractWizard*. Este método es el que realiza la operación una vez que están todos los parámetros necesarios para operar, correctamente configurados. En este punto cabe recordar que en general las operaciones en el cliente suelen ser de mediana o larga duración con lo cual esté método en realidad está decorado con un monitor/indicador de progreso, un ejemplo de esta situación se muestra en la Figura 3.1-28. Esta funcionalidad es transparente al desarrollador, la obtiene como un comportamiento por *default* provisto por el cliente y eficiente para el usuario que puede enviar la operación a un segundo plano, pudiendo observar su progreso como se indica en un ejemplo en la Figura 3.2-4 en una vista

específica para tal fin, a su vez que puede realizar otra tarea mientras la primera se ejecuta. Una vez culminada la operación enviada a segundo plano, el usuario puede recuperar los datos accediendo a la vista de progresos de operaciones mencionada anteriormente.

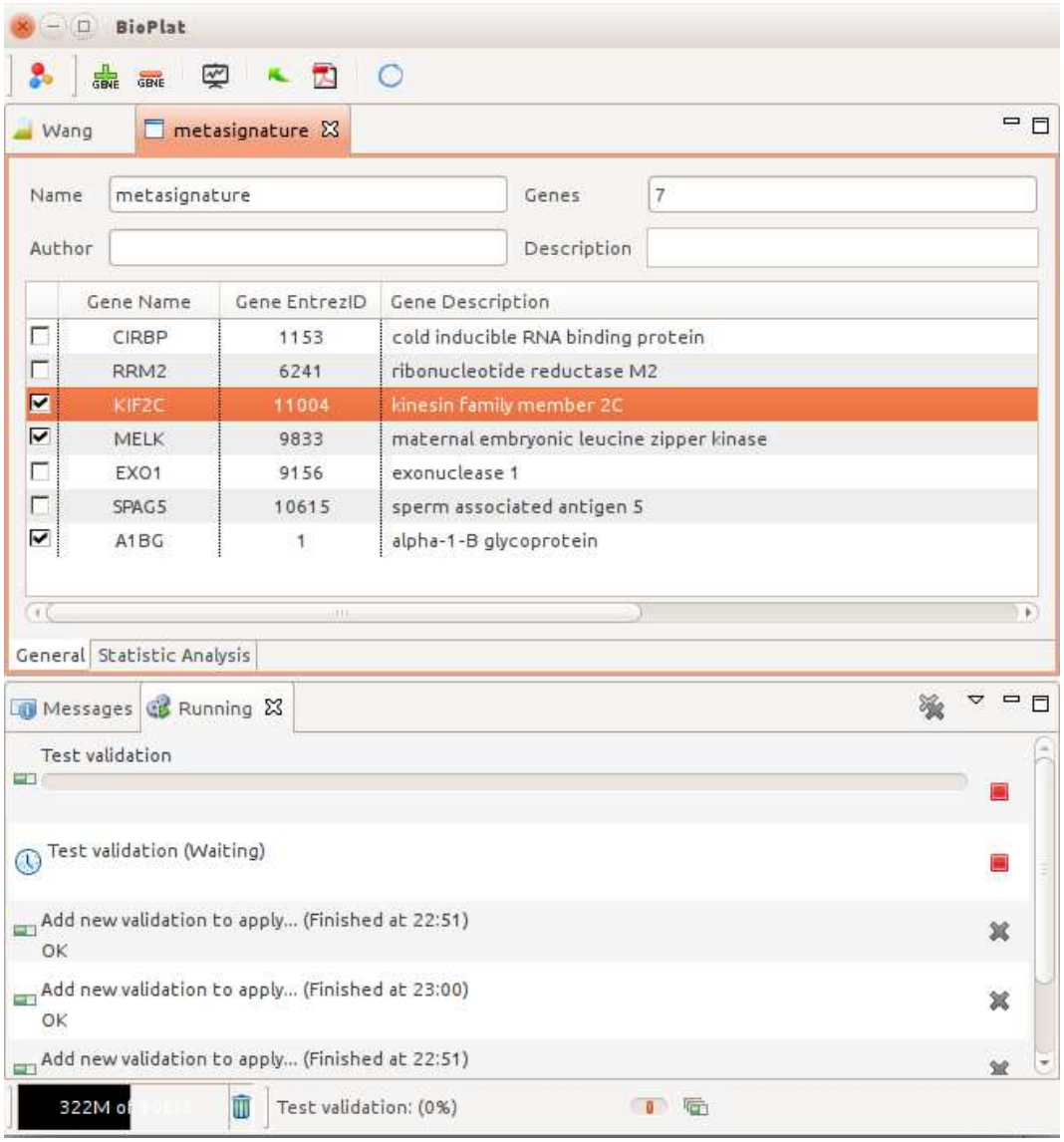


Figura 3.2-4- Vista de operaciones en ejecución en segundo plano

3.3 Publicaciones y subsidios

Este cliente fue presentado en la revista *Bioinformatics*[45], y actualmente se encuentra en proceso de presentación un nuevo artículo con todos los nuevos avances implementados de la plataforma.

El grupo de trabajo de la plataforma ha obtenido becas por parte del del Instituto Nacional del Cáncer[46] y de CAETI[47], y para continuar con las investigaciones y desarrollo de la plataforma.

Esta tesina fue seleccionada por la Agencia Nacional de Promoción Científica y Tecnológica, a través del FONSOFT[48] en su programa de ''Becas jóvenes Profesionales TIC'', para otorgar la beca para la finalización de los estudios de grado.

[45] http://bioinformatics.oxfordjournals.org/content/30/12/1782.abstract?sid=be507b21-ab1d-431b-81b2-69c2f1d197ca
[46] http://www.msal.gob.ar/inc/
[47] http://caeti.uai.edu.ar/
[48] http://www.agencia.mincyt.gob.ar/frontend/agencia/fondo/fonsoft

4 Trabajos futuros

4.1 Futuro del cliente

El cliente ya se encuentra en uso, y como todo *software* no está libre de errores, así eventualmente se van ajustando detalles a medida que el usuario los encuentra y los reporta. Aun así, quedan grandes aspectos en los que avanzar, entre los más importantes resaltamos los siguientes puntos:

- Introducción del sistema propuesto por Eclipse para la automatización de actualizaciones en el cliente Bioplat. Eclipse define un subproyecto de *Equinox* llamado p2, que utilizando las características de dinamicidad de OSGi, permite el *upgrade* de *bundles*, o *plugins*, en el cliente;

- Automatización del empaquetamiento del cliente Bioplat para las distintas plataformas donde es soportado. Actualmente se realiza manualmente desde el entorno de desarrollo Eclipse IDE;

- Integración del sistema de ayuda contextual provisto por Eclipse, para dar mejor soporte y asistencia al usuario en el uso del cliente;

- Mantener actualizada la arquitectura básica utilizada y que forman las dependencias del cliente Bioplat, tanto de Eclipse como de librerías de terceros, y el propio modelo, principalmente este último que es al que el cliente da soporte.

4.2 Futuro integral de la plataforma

En lo que tiene que ver con nuevos desarrollos en las demás componentes que forman parte de la plataforma. Resaltamos los siguientes puntos:

- Bioplat *cloud*: El usuario será capaz de publicar en la plataforma un *gene signatures* en la ''nube'' para poder utilizarlo en análisis posteriores, pero también para poder compartirlo con el resto de la comunidad Bioplat (una vez superado el proceso de cura llevado adelante por curadores Bioplat). Notar que esto implica un nuevo modo de llegar a un *gene signature*, que es importándolo desde esta nueva fuente;

- Servidor R para operaciones de larga duración. El usuario podría encolar una operación y dejarla ejecutando asincrónicamente en servidores HPC, recibiendo un aviso una vez esta termine y así poder obtener los resultados disponibles en la plataforma;

- Gestión remota, o en ''la nube'', de archivos de experimentos, de grandes volúmenes de información, del orden de gigabytes. Para realizar nuevas operaciones en los servidores de R;

- Administración remota: permite actualizar la base de datos Bioplat a través de una aplicación. Permitirá la actualización del genoma, de *gene signatures*;

- Divulgación de la plataforma tanto en ambientes académicos, como en laboratorios farmacéuticos.

5 Conclusiones

Se pudieron por un lado, encontrar soluciones informáticas a problemas técnicos reales. Y por el otro, lograr la interacción con el dominio de la biología genética, absorbiendo conocimientos en un área muy interesante a la vez que compleja, encontrando muy buenos resultados en el trabajo conjunto con especialistas en esas áreas, fundamentales de esta tesina.

Se logró construir un producto simple de distribuir, pudiendo ser ejecutado en los sistemas operativos más populares y sin requerir conocimientos técnicos por parte de los usuarios que quieran utilizar la plataforma a través del cliente Bioplat, logrando gran portabilidad. Además el cliente, al ser autocontenido, no interfiere con librerías preinstaladas del usuario.

Se enriqueció la experiencia de uso con el cliente, y así con la plataforma, tanto aprovechando características visuales propias de asistencia en los procesos complejos que se realizan, como haciendo uso de las posibilidades de integración con distintos sitios web o herramientas, con las cuales el usuario pudiera estar familiarizado y habituado a utilizar.

El cliente puede ser extendido por otros usuarios o desarrolladores que quieran enriquecer la plataforma. Se definió un modo adecuado de realizar las extensiones, mediante distintas técnicas de abstracción, que facilitan la interacción con las entidades de la plataforma.

Bibliografía

[1] M. Butti, Metodología analítica e integradora para la generación de biomarcadores de pacientes con cáncer de mama sobre la base de perfiles de expresión génica, 2013.

[2] D. Alonso, El desafío del cangrejo, 2011.

[3] G. Jiménez-Sánchez y A. Hidalgo Mirando, «Bases genómicas del cáncer de mama: avances hacia la medicina personalizada.,» *Salud Publica Mex 2009;51 supl 2:S197-S207,* vol. 51, nº 2, pp. 197-207, 2009.

[4] C. Hortsmann y G. Cornell, Core Java. Volume II Fundamentals. Ninth Edition, Prentice Hall, 2004.

[5] S. Mcculloch, R. S. Hall, K. Pauls y D. Savage, OSGi in action: Creating Modular Application in Java, Dreamtech Press/Wiley, 2011.

[6] J. McAffer, J.-M. Lemieux y C. Aniszczyk, Eclipse Rich Client Platform. Second Edition, The Eclipse series.

[7] C. Hortsmann y G. Cornell, Core Java. Volume I Fundamentals. Ninth Edition, Prentice Hall, 2004.

[8] M. Scarpino, S. Holder, S. Ng y L. Mihalkovic, SWT JFace in Action - GUI design with Eclipse 3.0, Manning, 2004.

[9] E. Gamma, R. Helm, R. Johnson y J. Vlissides, Design Patterns: Elements of Reusable Objetc-Oriented Software, Addison Wesley Professional Computing Series.

Listas de referencias

Lista de figuras

Lista de códigos fuentes

Lista de gráficos UML

Apéndice A: Glosario

ADN: Abreviatura de Ácido desoxirribonucleico. Biopolímero, o grupo molecular, que constituye el material genético de las células; contiene la información para la síntesis de proteínas.

ARNm (mensajero): Es el duplicado de la información de un determinad gen.

Bases de datos primarias: Repositorio de estudios con datos de expresión de genes en muestras

Bases de datos secundarias: bases de datos que contienen la información extraída de las bases de datos primarias, pero previamente curadas (procesadas).

Bioinformática: uso de las tecnologías de la información (informática) en biología y biotecnología, de gran importancia en el secuenciamiento del Genoma Humano. Es una de las especialidades de la biotecnología de mayor desarrollo es los últimos años.

Biomarcador: Es un marcador molecular, que puede tener utilidad diagnóstica, pronóstica, o predictiva de la enfermedad en estudio.

Experimento Bioplat: Contiene por un lado la expresión génica de para distintas muestras organizado por gen. Opcionalmente tiene datos agregados de observaciones clínicas de las muestras.

Famacogenómica: diseño de fármacos (medicamentos) específicos en función de las secuencias genéticas de grupos humanos. Objetivo tener medicamentos ''a medida'', más eficaces y sin efectos secundarios.

Fenotipo: Es la expresión del genotipo en función de un determinado ambiente o contexto.

Gen: El gen es la unidad básica de información del ADN

Gene signature: Es un biomarcador hipotético, en etapa de estudio.

Genoma: es la secuencia de bases presentes en su ADN. Las bases son 4 posibles que las identificaremos con las letras A, C, G o T, por los nombres Adenina, Citosina, Guanina y Timina respectivamente

Genotipo: Se refiere a la información genética que posee un organismo en particular, en forma de ADN.

Hitopatología: Rama de la Patología, que trata un diagnóstico a través del estudio de los tejidos

In silico: es una expresión que significa ''hecho por computadora o vía simulación computacional''.

Mitosis: División celular que conduce a la formación de dos células hijas de la primera.

Neoplasia: Crecimiento autónomo de un grupo de células, que desatiende a las pautas de comportamiento que impone el organismo. Este crecimiento sin control se origina en mutaciones genéticas acumulativas que dañan la capacidad de las células para responder señales regulatorias

Neoplasia: Crecimiento autónomo de un grupo de células, que desatiende a las pautas de comportamiento que impone el organismo. Este crecimiento sin control se origina en mutaciones genéticas acumulativas que dañan la capacidad de las células para responder a señales regulatorias.

Oncogenómica: El estudio de la relación entre el genoma de un individuo y el cáncer

Patología: Rama de la medicina encargada del estudio de las enfermedades en seres humanos

Transcripción: Síntesis de ARN mensajero. Proceso por el cual la información de un gen es ''replicada'' para luego en base a esta, sintetizar proteínas o enzimas.

Apéndice B: *Paper* publicado en *Bioinfomatics*

Bioinformatics Advance Access published March 10, 2014

BIOINFORMATICS APPLICATIONS NOTE 2014, pages 1–3
doi:10.1093/bioinformatics/btu111

Gene expression

Advance Access publication February 25, 2014

BioPlat: a software for human cancer biomarker discovery

Matias D. Butti, Hernan Chanfreau, Diego Martinez, Diego García, Ezequiel Lacunza and Martin C. Abba*

Basic and Applied Immunological Research Center, Faculty of Medical Sciences, National University of La Plata, 1900 La Plata, Argentina

Associate Editor: Janet Kelso

ABSTRACT

Summary: Development of effective tools such as oligo-microarrays and next-generation sequencing methods for monitoring gene expression on a large scale has resulted in the discovery of gene signatures with prognostic/predictive value in various malignant neoplastic diseases. However, with the exponential growth of gene expression databases, biologists are faced with the challenge of extracting useful information from these repositories. Here, we present a software package, BioPlat (Biomarkers Platform), which allows biologists to identify novel prognostic and predictive cancer biomarkers based on the data mining of gene expression signatures and gene expression profiling databases. BioPlat has been designed as an easy-to-use and flexible desktop software application, which provides a set of analytical tools related to data extraction, preprocessing, filtering, gene expression signature calculation, *in silico* validation, feature selection and annotation that leverage the integration and reuse of gene expression signatures in the context of follow-up data.

Availability and implementation: BioPlat is a platform-independent software implemented in Java and supported on GNU/Linux and MS Windows, which is freely available for download at http://www.cancergenomics.net.

Contact: mcabba@gmail.com

Supplementary information: Supplementary data are available at *Bioinformatics* online.

Received on September 3, 2013; revised on January 20, 2014; accepted on February 19, 2014

1 INTRODUCTION

Human cancer transcriptomes have been extensively profiled over the past decade, allowing the identification of different cancer molecular subtypes and the development of prognostic and predictive gene expression signatures. Gene expression signatures have been curated from the literature and collected into publicly available databases such as MSigDB 3.0 and GeneSigDB 4.0, with >10 000 gene signatures related to human diseases (Culhane *et al.*, 2012; Liberzon *et al.*, 2011). On the other hand, databases such as GEO (Gene Expression Omnibus) and AE (ArrayExpress) are the primary repositories for the raw data from functional genomic studies (Barrett *et al.*, 2013; Rustici *et al.*, 2013). A central focus on translational cancer research is that patient diagnosis and prognosis can be improved by stratification of patients based on functional genomics data beside relevant follow-up data. Therefore, we developed BioPlat (Biomarkers Platform), a user-friendly bioinformatics resource, which provides a set of analytical tools for the *in silico* identification of novel prognostic and predictive cancer biomarkers. It encompasses all the stages of data mining and analysis of the vast number of gene expression signatures and profiling data, including data extraction, preprocessing, filtering, *in silico* validation and feature selection.

Currently, there exists a myriad of applications and software focused on the integration and analysis of oncogenomics and clinicopathological data (e.g. Oncomine, ITTACA, PAPAyA, IntOGene, Cancer Genomics Browser, MeV, GenePattern). However, the salient feature that makes BioPlat unique is that it allows the direct and easy integration of gene expression signatures and gene expression profiling data to further perform survival analysis. Moreover, BioPlat implements features that are not included in other biomarker discovery tools. These features include statistic methods to measure the performance of prediction models and also algorithms to perform feature selection. Although several of these statistical tests and algorithms are mostly available in R packages, our software provides a unified framework that facilitates the use of these components.

2 METHODS

2.1 System implementation

BioPlat is a desktop client application implemented in Java and based on Rich Client Platform and Standard Widget Toolkit. It uses object-oriented programming, a local H2 in-memory database and Hibernate to perform the object-relational mapping. The Java-embedded database contains all the required annotation data for mapping probes and gene identifiers (Entrez ID and Ensembl ID) to gene symbols and related information. Statistical and data mining analyses are performed with the R statistical package and Bioconductor (Gentleman *et al.*, 2004). Briefly, we use several R/Bioconductor packages (e.g. frma, inSilicoDb, fpc, affy, genefu, limma, survival, survcomp) and functions (e.g. dist, hclust, cutree) for data retrieval, preprocessing, management and analysis. The integration between Java and R was achieved using an R bridge developed by us (named R4J) based on Rserve. BioPlat was designed using extension points for allowing users with programming knowledge to incorporate new behavior and algorithms easily.

2.2 Features and algorithms implementation

BioPlat consists of an integrated set of tools that allow the access and analysis of data deposited in gene expression signatures and gene expression profiling repositories (Fig. 1 and Supplementary Data S1). Gene expression signatures can be queried and filtered by tumor localization,

*To whom correspondence should be addressed.

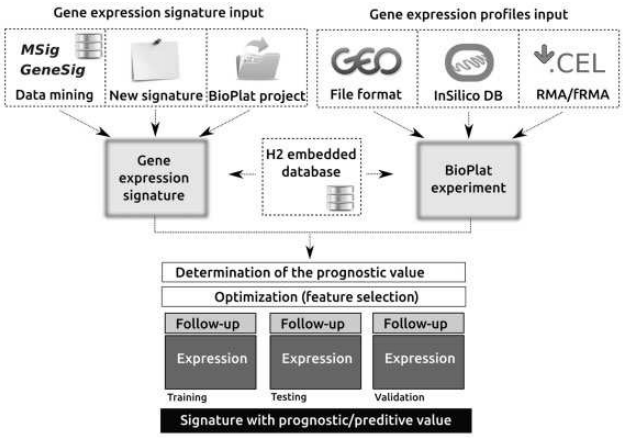


Fig. 1. Diagrammatic workflow of BioPlat. The first two steps in BioPlat are the gene signature creation and the import of experiments containing follow-up and gene expression profiling data. BioPlat provides several processes for the gene list generation such as access and data mining of gene signature databases. Gene expression data, provided by experiments incorporated in the platform, are used to cluster the samples of the experiments based on K-means method. This clustering will let the platform validate the statistical significance level of the biomarker using the follow-up data of the same experiment in terms of C-index, Log-rank test and Kaplan–Meier survival analysis

tissue types or signature identifiers directly from the two main curated repositories: MSigDB (Liberzon *et al.*, 2011) and GeneSigDB (Culhane *et al.*, 2012). BioPlat allows the creation/edition of gene lists by using any of the identifiers included in the embedded database (e.g. Gene name, Entrez, Ensembl and Probe IDs). In addition, BioPlat integrates information and tools provided by other well-known online resources such as DAVID (Database for Annotation, Visualization and Integrated Discovery), STRING (Search Tool for the Retrieval of Interacting Genes/Proteins), Enrichr, Expression Atlas, RNA-seq Atlas, Gene Cards and others.

Gene expression profiles (BioPlat experiments) are used for the validation and optimization steps of gene signatures previously defined. A BioPlat experiment contains follow-up data and gene expression profiles of samples. BioPlat experiments can be programmatically queried and imported from InSilico database (Taminau *et al.*, 2011), local GEO-formatted files or preprocessing Affymetrix CEL files with RMA/fRMA (frozen robust multiarray analysis) algorithms. Although the platform is prepared for incorporating experiment mining processes as a new way to propose candidate gene lists, the main use of the experiments in BioPlat is to estimate the prognostic/predictive value of a gene expression signature. In the validation process, given a pair of gene signature experiments, the results shown are the annotated data matrix with the patient's cluster, the concordance index (C-index), the Log-Rank test *P*-value and the Kaplan–Meier curves.

Because of the large number of features usually present in the gene expression signature analyzed, signature optimization is a key step that was considered in the development of the software. We focus on selecting compact feature subsets while maximizing prediction accuracy for biomarker discovery to reduce complexity. BioPlat provides two optimization processes—'blind search' and particle swarm optimization (PSO)—to search a better candidate gene signature in the solution space of the signature to be optimized. The implementation of the PSO algorithm in BioPlat was based on the PSO binary version previously described (Kennedy and Eberhart, 1997). Briefly, PSO is a machine learning metaheuristic whose aim is to reduce the solution space for finding the optimum gene signature without going over the whole space based on a metric of quality associated with the outcome (e.g. C-index). Considering that the metric of performance is a critical point for the algorithms and that new metrics are being worked on, the comparison strategy was designed in the platform to be easily replaced without affecting the core of the algorithm. Moreover, to avoid the overfitting of the found solution on the training experiment used for running the heuristic logics, any optimizer allows configuring not only a training experiment but also a testing experiment and a validation one.

3 USAGE EXAMPLE

We wanted to provide a systematic analysis of breast cancer gene expression signatures, in an effort to identify breast cancer biomarkers with prognostic value. The integration of 655 breast cancer signatures obtained from MSig and GeneSig databases, using the Metasignature wizard, revealed a set of 140 genes that were the most commonly deregulated transcripts among breast cancer studies (Supplementary Data S2A). K-means clustering of the metasignature in three independent publicly available datasets (GSE25066 n = 508, NKI dataset n = 295 and a compiled dataset of 737 carcinomas derived from GSE2034, GSE3494 and GSE11121) identified two main clusters of breast carcinomas that differed in their relapse-free survival (Supplementary Data S2B).

Feature selection analysis of the 140 gene metasignature in the GSE25066 study (training dataset) using PSO-based process allows reducing the candidate gene signature to 60 genes (Supplementary Data S2C and S3) with improved prognostic performances as reflected by having the highest C-index and the lowest nominal log-rank *P*-values for relapse-free survival. Interestingly, cluster 1 was highly associated with luminal A/B breast cancer intrinsic subtypes, whereas cluster 2 was associated with basal-like breast carcinomas. Gene enrichment analysis revealed two functional modules significantly affected: one related with the response to steroid hormone (upmodulated in cluster 1) and another related with the cell cycle signaling pathway (upmodulated in cluster 2) (Supplementary Data S2A–E). Finally, we compared the prognostic performance of the 60-gene signature with the PAM50 and genomic grade Index signatures on the GSE25066 dataset. This analysis demonstrated that the 60-gene signature outperformed the genomic grade Index, having similar prognostic value that the PAM50 signature in predicting the relapse-free survival of patients with early-stage breast cancers (Supplementary Data S4).

4 CONCLUSION

BioPlat facilitates the integration, analysis, validation and feature selection of gene signatures derived from different databases in the context of follow-up data obtained from publicly available gene expression profiling repositories.

Funding: National Agency of Scientific and Technological Promotion (PICT-0275) and The National Cancer Institute of Argentina.

Conflict of Interest: none declared.

REFERENCES

Barrett,T. *et al.* (2013) NCBI GEO: archive for functional genomics data sets—update. *Nucleic Acids Res.*, **41**, D991–D995.

Cullhane,A.C. *et al.* (2012) GeneSigDB: a manually curated database and resource for analysis of gene expression signatures. *Nucleic Acid Res.*, **40**, D1060–D1066.

Gentleman,R.C. *et al.* (2004) Bioconductor: open software development for computational biology and bioinformatics. *Genome Biol.*, **5**, R80.

Kennedy,J. and Eberhart,R. (1995) Particle swarm optimization. *IEEE Neural Netw. Proc.*, **4**, 1942–1948.

Liberzon,A. *et al.* (2011) Molecular signatures database (MSigDB) 3.0. *Bioinformatics*, **27**, 1739–1740.

Rustici,G. *et al.* (2013) ArrayExpress update—trends in database growth and links to data analysis tools. *Nucleic Acids Res.*, **41**, D987–D990.

Taminau,J. *et al.* (2011) InSilicoDb: an R/Bioconductor package for accessing human Affymetrix expert-curated datasets from GEO. *Bioinformatics*, **27**, 3204–3205.