UNIVERSITAT POLITÈCNICA DE VALÈNCIA


Escola Tècnica Superior d'Enginyeria Informàtica

Universitat Politècnica de València

# Entorno de negociación automática bilateral

**Sistemas Multiagente**

José Ángel González Barba – jogonba2@dsic.upv.es

# Índice

# 1. Introducción

## 1.1 Proyecto

Se trata de un entorno de negociación automática bilateral, desarrollado para la asignatura de Sistemas Multiagente, que facilita el desarrollo de agentes negociadores. Se proporcionan funciones de utilidad, estrategias de concesión, criterios de aceptación, métodos de generación de ofertas y modelos de aprendizaje automático para predecir la respuesta del contrario ante una oferta.

Además, es posible implementar otros tipos de estrategias mediante la inserción de código en los scripts del proyecto. Las estructuras de datos, los parámetros que recibe cada estrategia y la forma de implementar estrategias se discuten en apartados posteriores. Con ello, los scripts mencionados son:

- **Offers.py**: es el script donde se definen las estrategias de generación de ofertas. Con el entorno se proporcionan varias estrategias, a destacar:
    - **Métodos basados en optimización de la función de utilidad**:
        - **Hyperopt**: hace uso de árboles de estimadores de Parzen (**TPE**) para optimizar hiperparámetros [1]. La implementación dada no permite manejar conocimiento del oponente (no se ha implementado por falta de tiempo).
        - **Scipy**: se emplean los optimizadores de la librería **Scipy** [2] que permiten manejar restricciones (en apartados posteriores se muestra la necesidad de ello) para optimizar la función de utilidad. La implementación proporcionada no está completa pero puede servir como base para continuar.
        - **Algoritmo genético**: adaptación del algoritmo genético, implementado en una asignatura anterior, aplicado a regresión simbólica. En este caso, se emplea para optimizar la función de utilidad y solo permite manejar valores enteros y categóricos. No garantiza que la utilidad de la oferta generada supere el valor de concesión **s** en un step determinado, para esto es necesario ajustar parámetros (dependientes del problema). Esta implementación permite utilizar conocimiento del oponente.
    - **Otros métodos**:
        - **Aleatorio**: genera ofertas con valores aleatorios para cada atributo, dentro de los intervalos permitidos. La implementación proporcionada también permite manejar conocimiento del oponente.

- **Utility.py**: en este script se definen las funciones de utilidad que permiten modelar las preferencias de los agentes. Con el entorno solo se proporciona la implementación de una función de utilidad lineal, que es el modelo de preferencia multiatributo más común, pero no es capaz de modelar dependencias complejas entre atributos.

- **Concession.py**: se definen las estrategias de concesión a utilizar por los agentes. Estas estrategias determinan cómo, cuánto y cuándo concede un agente (modifica su valor de aspiración). Las estrategias de concesión implementadas son:

    - **Concesión temporal**
    - **Toma y daca**
        - **Relativo**
        - **Absoluto**
        - **Promediado**
    - **Concesión aleatoria**
    - **No concesión**

- **Utils.py**: se emplea para añadir todo el código auxiliar necesario para definir las estrategias e.g. cargar información del oponente o preprocesarla.

- **Acceptance.py**: en este script se definen los criterios de aceptación, empleados para determinar cuándo un agente acepta una oferta determinada.

- **Classifier.py**: en el script se añaden clasificadores que permiten resolver problemas de clasificación o regresion dependientes de información del oponente. Principalmente, en el entorno proporcionado, se utilizan durante la generación de ofertas para determinar si el agente oponente aceptará o no una oferta, pero se pueden utilizar en cualquier parte desarrollada por el usuario p.e. en la definición de criterios de aceptación. Se proporcionan los clasificadores del *framework* **scikit-learn** [3]:

    - **Multilayer perceptron**
    - **Naïve Bayes (Gaussian)**
    - **Naïve Bayes (Multinomial)**
    - **Naïve Bayes (Bernoulli)**
    - **SVM**
    - **AdaBoost**
    - **Random Forest**
    - **Decision Trees**
    - **Nearest Centroid**
    - **K Nearest Neighbours**

Los parámetros de los clasificadores no se pueden modificar desde la definición de los agentes (queda pendiente la modificación).

- **Statistics.py**: en este script se definen las estadísticas utilizadas para evaluar a los agentes tras la ejecución de un torneo. Se utilizan los estadísticos proporcionados por **numpy** [4].

    - **Media artimética**
    - **Mínimo**

    - **Máximo**
    - **Desviación típica**
    - **Primer percentil**
    - **Segundo percentil**
    - **Tercer percentil**

- **Agent.py**: es la clase que representa los agentes, para ello, carga la definición de sus parámetros (información, dominio y parámetros de estrategias) desde los ficheros de agentes **json**. En ella se especifican los métodos para generar y recibir ofertas, actualizar el nivel de aspiración, getters y setters de información del agente etc. En principio, no es necesario modificar este fichero.

- **Exchange.py**: es el script que controla el bucle principal de la negociación. Maneja parámetros como el número de ofertas a emitir en cada *step*, el *path* donde se encuentra la información de los agentes, el *path* dónde almacenar conocimiento sobre los agentes o la forma de mostrar el resultado de la negociación (gráfica y por consola). En principio, tampoco es necesario modificar este script.

- **Graphics.py**: contiene los métodos para graficar el proceso de negociación. Es posible representar todas las ofertas simultáneamente o de forma interactiva siguiendo el orden en el que fueron realizadas. En cualquier caso, la representación de ofertas se realiza al final de la negociación para no ralentizar el proceso.

- **Messages.py**: en este script se definen métodos para generar cualquier cadena a mostrar por la aplicación, principalmente, resultados de los procesos e información de la negociación y de las ofertas. Se agrupan todas las cadenas en una clase para facilitar una posible traducción de la herramienta.

- **Main.py**: es el punto de entrada de la aplicación, carga el fichero de configuración de la herramienta e inicia el proceso de negociación en función de los parámetros de configuración.

Por otro lado, con respecto a la definición de agentes, estos se definen mediante ficheros **json** (según la configuración por defecto, en el directorio **Bots**), donde se indica información del agente, el dominio en el que pretende negociar y los parámetros de las estrategias empleadas por el agente. Se pueden consultar los tres agentes de ejemplo en la carpeta **Bots**.

Por último, durante el desarrollo de esta memoria, se pretende introducir al lector al uso de la herramienta desarrollada, es por ello por lo que primero se comentarán detalles de la arquitectura y por último se llevará a cabo una guía de cómo desarrollar un agente y evaluarlo mediante la negociación con otros agentes negociadores.

## 1.2 Dependencias

El entorno desarrollado requiere ciertas dependencias para poder funcionar. Más concretamente, dichas dependencias son:

- **Numpy**: es utilizado para manipular arrays, estadísticos para torneos y otras funciones numéricas.

- **Scipy**: principalmente empleado para los optimizadores usados durante la generación de ofertas y otras funciones numéricas.

- **Matplotlib**: librería gráfica utilizada para graficar el proceso de negociación.

- **Scikit-learn**: librería utilizada para proporcionar clasificadores de ejemplo en el script **Classifier.py**.

- **Deap**: es un *framework* de computación evolutiva utilizado en la implementación del algoritmo genético de generación de ofertas.

- **Hyperopt**: librería para optimización en espacios de búsqueda complejos formados por valores multidimensionales de tipo real, discreto o condicional. Es utilizada en la generación de ofertas **Hyperopt**.

Para instalar las dependencias mencionadas, se proporciona el script **setup.py**, que permite realizar la instalación mediante el gestor de paquetes **pip** (lo instala automáticamente si el usuario no dispone de él) si se le indica el parámetro **install** (**python setup.py install**). Si la ejecución finaliza por algún error, es recomendable instalar la plataforma **Anaconda** [5] y utilizarla para instalar a mano los paquetes anteriores (**conda install <package>**).

## 1.3 Modos de ejecución

En este apartado se discuten los modos de ejecución y los parámetros que se le pueden pasar a la herramienta (mediante el fichero **config.json**) para modificar ciertos aspectos del proceso de negociación, independientes de los agentes negociadores.

En primer lugar, si abrimos el fichero **config.json** en la raíz del proyecto, se puede observar información sobre el proyecto en el campo **project_info**, este campo únicamente sirve para controlar la versión actual de la herramienta y almacenar información sobre el autor. Modificar este campo no afecta al proceso de negociación y únicamente puede variar la salida por pantalla.

En segundo lugar, encontramos el campo **agents**. En este campo se especifican los agentes que participarán en el proceso de negociación, si solo se consideran dos agentes, será un escenario de negociación bilateral, mientras que si se añaden más agentes, se llevará a cabo un torneo de negociaciones bilaterales entre todo par de agentes especificados.

Por otro lado, el campo **knowledge** permite controlar cómo va a manejar la herramienta, información de los agentes negociadores, para que, posteriormente, los agentes puedan utilizarla durante su ejecución. Concretamente, los atributos del campo **knowledge** son:

- **corpus_path**: este atributo permite especificar la ruta donde los agentes van a almacenar y cargar información sobre los oponentes (en caso de que la usen). La obtención de información durante el proceso de negociación la controla el núcleo (**Exhange.py**) para eliminar dicha tarea de la parte de los agentes y facilitar el trabajo a los desarrolladores. En apartados posteriores se comenta cómo y qué información almacenan los agentes sobre sus oponentes.

- **save_corpus**: mediante este atributo le indicamos a la herramienta si durante la negociación se va a almacenar conocimiento del oponente o no. El directorio donde se almacenará dicha información viene especificado por el parámetro **corpus_path** anterior, y el fichero con información sobre un determinado agente, toma como nombre el nombre del agente.

El último campo que encontramos en el fichero **config.json** es **params**. Este campo permite indicar algunos de los parámetros que utilizará la herramienta para modificar aspectos del proceso de negociación. Entre estos parámetros encontramos:

- **n_offers**: permite modificar el número de ofertas que se envían en cada paso de la negociación. Por defecto, se envía una única oferta. Si el parámetro es $> 1$, al recibir un conjunto de ofertas, se aceptará la mejor de todas las recibidas.

- **show_offers**: permite mostrar por pantalla, al final del proceso de negociación, las ofertas que han sido enviadas por cada agente en cada paso de negociación.

- **first_random**: si su valor es True, el primero en iniciar la negociación será uno de los dos agentes escogidos de forma aleatoria. Si no, el agente que inicia la negociación es el agente que primero se carga desde el fichero **json**.

- **graphic_process**: se le indica a la herramienta si se debe graficar, al final del proceso, el resultado de la negociación (valores de utilidad de las ofertas propuestas por cada agente, representado en una gráfica 2D, donde los ejes x e y son el nivel de aspiración de los agentes uno y dos, respectivamente.

- **graphic_interactive**: indica la forma de graficar el resultado de la negociación. Hay dos formas implementadas, si el parámetro es False, todas las ofertas se representan de forma estática. Sin embargo, si el valor del parámetro es True, se representarán las ofertas de forma secuencial en el orden en que fueron enviadas.

Un ejemplo de fichero **config.json** válido se muestra en la Figura 1.

```
{

  "project_info" : {

    "version" : "0.4",
    "author" : "JAGB"
  },

  "agents" : {

    "Tyrion" : "./Bots/Tyrion.json",
    "JohnSnow" : "./Bots/JohnSnow.json"

  },

  "knowledge" : {

      "corpus_path":"./",
      "save_corpus" : true
  },

  "params" : {

      "n_offers" : 1,
      "show_offers" : true,
      "first_random" : false,
      "graphic_process" : true,
      "graphic_interactive" : false
  }

}
```

Figura 1. Ejemplo de fichero **config.json** válido.

# 2. Agentes

## 2.1 Definición de agentes

En el presente apartado, se comenta la estructura de un agente, es decir, las diferentes partes que componen el fichero **json** con la definición de dicho agente, en concreto: información sobre el agente, el dominio en el que negociará y los parámetros de las estrategias seguidas por el mismo. También se expondrá la clase Agente, que actúa como un contenedor de estrategias parametrizadas por los atributos de un agente definido en **json**. Por último, se mostrarán y comentarán los agentes de ejemplo proporcionados con la herramienta.

### 2.1.1 Información

El primer campo de un fichero de agente **json**, **agent_info**, permite almacenar información asociada al agente. La mayoría de esta información no es usada por la herramienta, pero sería interesante emplearla como conocimiento del oponente para algunas estrategias. Actualmente, la información considerada es:

- **Name**: permite identificar a los agentes, es utilizado en los *logs*, para almacenar ficheros con conocimiento sobre un agente etc.
- **Gender**: permite indicar el género del agente. Este campo no es utilizado actualmente por la herramienta.
- **Country**: permite indicar el país del agente. Este campo no es utilizado actualmente por la herramienta.
- **Birthday**: permite indicar la fecha de creación del agente. Este campo no es utilizado actualmente por la herramienta.
- **Reputation**: permite indicar la reputación del agente. Este campo no es utilizado actualmente por la herramienta.
- **Description**: contiene una definición del agente. Este campo no es utilizado actualmente por la herramienta.
- **Domain**: contiene el nombre del dominio en el que un agente puede negociar. El campo no es utilizado actualmente por la herramienta, pero puede ser útil para identificar agentes interesados en dominios similares.

- **Using oponent knowledge**: este atributo permite definir si el agente utilizará conocimiento del oponente o no. En este caso, sí es considerado por la herramienta, y solo es posible asignarle un valor True si se dispone de conocimiento del oponente (el método para utilizar conocimiento del oponente se discute en secciones posteriores).

Un ejemplo de información del agente se muestra en la Figura 2.

```
"agent_info" : {
    "name": "Tyrion",
    "gender": "male",
    "country": "spain",
    "birthday": "16/08/91",
    "reputation": 0.3,
    "description" : "tyrion, interested in ...",
    "using_oponent_knowledge" : false,
    "domain" : "car sale"
}
```

Figura 2. Información del agente en el fichero **json**.

Ninguno de los campos, a excepción de **using_oponent_knowledge**, son obligatorios, aunque sí es recomendable definirlos siempre (en especial el nombre) porque se pueden utilizar en ciertas situaciones, como por ejemplo al emplear conocimiento del oponente. Sí que es obligatorio definir el campo **agent_info** aunque no se especifiquen algunos atributos en él. También será obligatorio definir los demás campos asociados con el dominio y con las estrategias básicas de los agentes, esto se discute en los dos siguientes apartados.

### 2.1.2 Dominio

El dominio en el que participa un agente es una parte imprescindible en el proceso de negociación y en el caso de la herramienta, se considera como información asociada al agente ya que determina con qué atributos va a negociar éste. Por tanto, si se quiere que un agente participe en **n** dominios distintos, es necesario definir **n** ficheros de agente, cambiando los campos asociados al dominio de negociación.

Dentro del fichero **json** de un agente, encontramos tres campos, de obligatoria definición, relativos al dominio, más detalladamente, estos campos son:

- **attributes**: el atributo es un diccionario que contiene los atributos por los que negociará el agente y el tipo de dichos atributos. Un ejemplo de dicho atributo se muestra en la Figura 3.

```
"attributes" : {
    "price" : "integer",
    "color" : "categorical",
    "abs"   : "categorical"
}
```

Figura 3. Campo **attributes** del fichero **json**.

- **weights_attr**: mediante este atributo se definen las preferencias de los agentes, utilizadas por la función de utilidad (se discutirán dichas funciones en apartados posteriores). En la Figura 4 se muestra un ejemplo de este atributo, donde se observa que la suma de los pesos de los atributos debe ser 1.

```
"weights_attr" : {
   "price" : 0.6,
   "color" : 0.2,
   "abs"    : 0.2
},
```

Figura 4. Campo **weights_attr** del fichero **json**.

- **values_attr**: controla el valor que toman los atributos dentro de ciertos intervalos. A su vez, está compuesto por un diccionario con tres valores: **integer**, **float** y **categorical** donde se definen los atributos de tipo entero, real y categórico respectivamente. Dichas definiciones de atributos están formadas por dos campos más:

    - **properties**: definen información sobre el dominio de los tipos. Si el tipo es **integer** o **float**, se debe especificar el valor mínimo y el máximo. Si por el contrario, el tipo es categórico, se deben enumerar las posibles elecciones (**choices**). En este ultimo caso, es recomendable utilizar valores enteros como posibles elecciones (sobretodo si se va a utilizar conocimiento del oponente), pero también es posible utilizar cadenas en algunas ocasiones (no se han testeado). En la Figura 5 se muestra un ejemplo de cada tipo (**integer** y **categorical**).

```
"values_attr" : {
 "integer"     : {
    "price" : {
      "properties" : {
         "max" : 3000,
         "min" : 0
       },
              ...
    },
      ...
  },
   ...
}
```

```
"values_attr" : {
  "categorical" : {
    "color" : {
      "properties" : {
        "choices" : [0, 1, 2]
       },
            ...
    },
      ...
  },
   ...
}
```

Figura 5. Campo **properties** del campo **values_attr** para los tipos de datos.

- **conditions**: en este campo se especifican los intervalos o elecciones categóricas que determinan el valor de un atributo si la cantidad de este se encuentra dentro de dichos intervalos o coincide con las elecciones especificadas.

  En el caso de atributos numéricos (**integer** y **float**)**,** es necesario especificar intervalos continuos (el valor máximo del intervalo $i$ debe coincidir con el valor mínimo del intervalo $i + 1$). Además, el valor mínimo del primer intervalo debe coincidir con el valor especificado en el campo **min** de **properties** y el valor máximo del último intervalo debe coincidir con el valor del campo **max** de **properties**.

  Así, las condiciones sobre atributos numéricos, se especifican de la forma mostrada en la Figura 6. Donde cada intervalo se define como un par **<identificador> : [$\mathbf{min}_i$, $\mathbf{max}_i$, utilidad]**.

```
"conditions" : {
    "cond_1" : [0, 1000, 1],
    "cond_2" : [1000, 2000, 0.5],
    "cond_3" : [2000, 7500, 0.3],
    "cond_4" : [7500, Infinity, 0]
}
```

Figura 6. Definición de intervalos sobre atributos numéricos.

  Si el atributo es categórico, las condiciones se definen como un par **<identificador> : [elección, utilidad]**. Tal como se muestra en la figura 7.

```
"conditions" : {
    "cond_1" : [0, 0.4],
    "cond_2" : [1, 0.1],
    "cond_3" : [2, 1]
}
```

Figura 7. Definición de condiciones sobre atributos categóricos.

Con todo ello, un ejemplo completo de especificación de un dominio, en este caso de venta de coches, se muestra en la Figura 8, donde se pueden identificar las diferentes partes ya comentadas.

```
"attributes" : {"price" : "integer", "color": "categorical", "abs":
"categorical"},

"weights_attr" : {
   "price" : 0.8,
   "color" : 0.1,
   "abs"   : 0.1
},
```

```
"values_attr" : {
    "integer"      : {
        "price" : {
            "properties" : {
                "max" : 6000,
                "min" : 0
             },
             "conditions" : {
                "cond_1" : [0, 1000, 0.1],
                "cond_2" : [1000, 5000, 0.7],
                "cond_3" : [5000, Infinity, 1]
             }
        }
   },

   "float"        : {},

   "categorical" : {
        "color" : {
            "properties" : {
                "choices" : [0, 1, 2]
            },
            "conditions" : {
                "cond_1" : [0, 0.1],
                "cond_2" : [1, 0.5],
                "cond_3" : [2, 1]
            }
        },
        "abs" : {
            "properties" : {
                "choices" : [0, 1]
            },
            "conditions" : {
                "cond_1" : [1, 1],
                "cond_2" : [0 , 0]
            }
        }
   }
}
```

Figura 8. Especificación completa del dominio dentro del fichero **json** de definición del agente.

### 2.1.3 Parámetros de las estrategias

# 3. Guía rápida

## 3.1. Instalación

En primer lugar, es necesario instalar las dependencias de la aplicación para poder ejecutarla. Como ya se comentó en el apartado del capítulo uno dedicado a la instalación, podemos hacer uso del script **setup.py** (**python setup.py install**) proporcionado por la herramienta, la salida de la ejecución debe ser como se muestra en la figura X.

```
~$ python setup.py install

Installing pip ...
Installing numpy ...
Installing scipy ...
Installing matplotlib ...
Installing scikit-learn ...
Installing deap ...
Installing hyperopt ...
All installed.
```

Figura X. Instalación de las dependencias.

Si la ejecución finaliza con algún error, como ya se comentó, es recomendable instalar la plataforma **Anaconda** e instalar los paquetes necesarios desde ella. Sin embargo, el objetivo de la guía rápida es marcar los pasos necesarios para llegar a ejecutar la herramienta con los ejemplos proporcionados, por lo que no se ha detallado una instalación de esta manera y se asume que, en este punto, el lector ya dispone de todas las dependencias requeridas por el entorno.

## 3.2. Primeras negociaciones

Una vez se ha completado la instalación de las dependencias, ya es posible ejecutar en el entorno una negociación de prueba entre los agentes de ejemplo proporcionados. Así, como primera prueba pondremos a los agentes **Tyrion** (**Bots/TyrionPrimeraPrueba.json**) y **John Snow** (**Bots/JohnSnowPrimeraPrueba.json**) a negociar en el dominio de venta de coches. Para más información sobre la definición de los agentes y cómo diseñar nuevas estrategias de negociación para estos, consultar los capítulos dos y tres.

Con respecto a los ficheros de definición de agentes, como es la primera negociación entre ellos, los agentes no podrán utilizar conocimiento del oponente, por lo que el valor del campo **using_oponent_knowledge** del fichero **json** de los agentes debe ser **false**.

Por otro lado, en la configuración (**config.json**) empleada en esta prueba se han especificado los agentes que participarán en la negociación, se ha indicado que se quiere que los agentes guarden información del oponente para utilizarla en pruebas siguientes (**save_corpus=True**), además, se emitirán tres ofertas en cada paso de negociación (**n_offers=3**) y se mostrará, al final

de la negociación, una representación gráfica estática del proceso (**graphic_process=True** y **graphic_interactive=False**) y el *log* con las ofertas emitidas en cada paso. Así, el fichero **config.json** queda de la siguiente manera (Figura X):

```
{

  "project_info" : {
    "version" : "0.4",
    "author" : "JAGB"
  },

  "agents" : {
    "Tyrion" : "./Bots/TyrionPrimeraPrueba.json",
    "JohnSnow" : "./Bots/JohnSnowPrimeraPrueba.json"

  },

  "knowledge" : {
      "corpus_path":"./",
      "save_corpus" : true
  },

  "params" : {
    "n_offers" : 1,
    "show_offers" : true,
    "first_random" : false,
    "graphic_process" : true,
    "graphic_interactive" : false
  }

}
```

Figura X. Fichero **config.json** empleado en la primera prueba

Con esto, basta con ejecutar **Main.py** (**python Main.py**) para iniciar un proceso de negociación bilateral entre los agentes mencionados. Tras ejecutar la aplicación obtenemos la salida por consola mostrada en la Figura X y la representación gráfica de la Figura X.

```
~$ python Main.py


*************
* Bilateral *
*************


Step  52 ...

* Deal accepted by dealer JohnSnowPrimeraPrueba at step
52 *

  · Benefits dealer JohnSnowPrimeraPrueba: 0.760000
```

```
 · Benefits dealer TyrionPrimeraPrueba: 0.700000
 · Concession dealer JohnSnowPrimeraPrueba: 0.500000
 · Concession dealer TyrionPrimeraPrueba: 0.581675
 · Offer: {'color': 2, 'price': 1528.0, 'abs': 1}


* Offers *

- Step 0 from TyrionPrimeraPrueba :

        {'color': 2, 'price': 164.0, 'abs': 1}
        {'color': 2, 'price': 135.0, 'abs': 1}
        {'color': 2, 'price': 699.0, 'abs': 1}

- Step 1 from JohnSnowPrimeraPrueba :

        {'color': 2, 'price': 2550.0, 'abs': 1}
        {'color': 2, 'price': 2694.0, 'abs': 1}
        {'color': 1, 'price': 3356.0, 'abs': 1}

                        ...

- Step 52 from TyrionPrimeraPrueba :

        {'color': 2, 'price': 1116.0, 'abs': 1}
        {'color': 2, 'price': 1201.0, 'abs': 1}
        {'color': 2, 'price': 1528.0, 'abs': 1}
```

Figura X. Salida de la primera prueba.

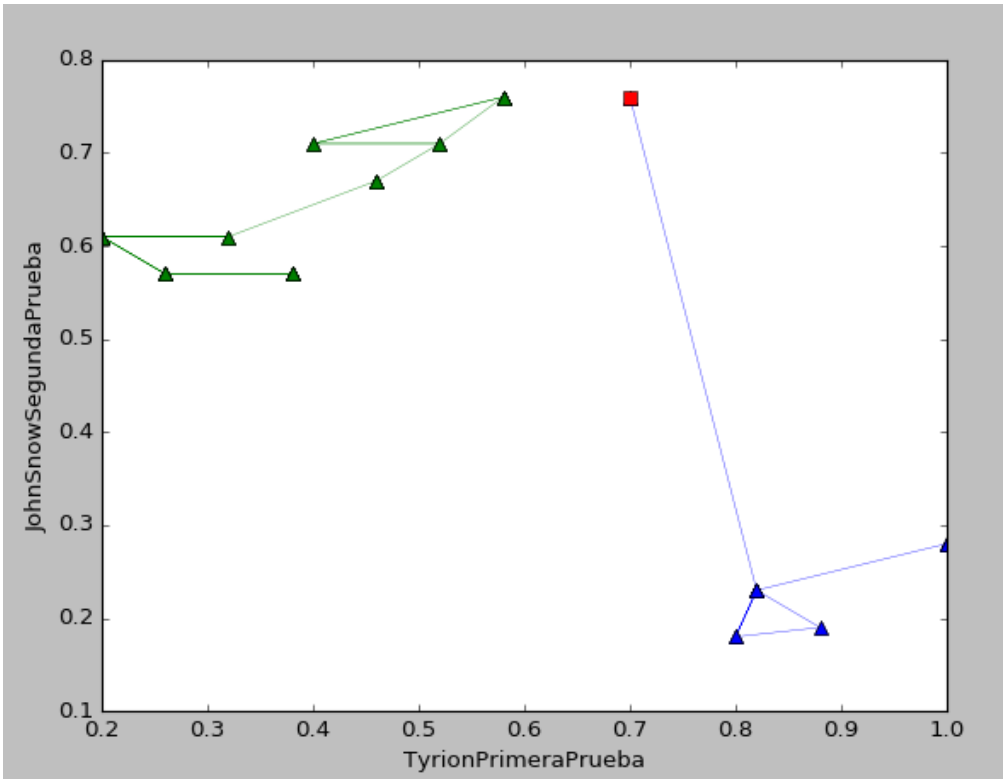


Figura X. Representación gráfica del proceso de negociación en la primera prueba.

Como se puede observar, se imprime por pantalla la etapa en la que se ha alcanzado el fin de la negociación indicando, si se alcanza un acuerdo, para cada agente, su nivel de aspiración y el beneficio que le proporciona la oferta aceptada. También se imprime la oferta aceptada, que debe coincidir con alguna de las ofertas emitidas en el último paso de negociación.

En este ejemplo no se ha utilizado conocimiento del oponente porque los agentes no disponían información de los oponentes, sin embargo, al haber especificado el parámetro **save_corpus** en la ejecución anterior, se han generado los ficheros **TyrionPrimeraPrueba.dump** y **JohnSnowPrimeraPrueba.dump** que contienen conocimiento sobre el agente que da nombre al fichero (para más información sobre estos ficheros consultar el apartado dedicado al **Aprendizaje** en el capítulo tres). Por este motivo, se muestra a continuación un proceso de negociación donde los dos agentes usan conocimiento del adversario para generar ofertas.

Para ello, podemos mantener el parámetro de **config.json**, **save_corpus** a **True** para continuar almacenando conocimiento del oponente (se añade en binario al fichero) y basta con cambiar el parámetro **using_oponent_knowledge** a **True**, en los ficheros de definición **json** de los agentes, para que empiecen a utilizar conocimiento del oponente. Es conveniente notar, que por defecto, solo algunas de las estrategias de generación de ofertas utilizan conocimiento del oponente (**random_offer** y **genetic_offer**), aunque sería simple implementarlo también con la estrategia **hyperopt** o en nuevas estrategias definidas por el usuario.

Por este motivo, para esta prueba, cambiamos, en los ficheros **json** de definición de agentes, el método de generación de ofertas (**offer_type**) de **JohnSnowPrimeraPrueba** y **TyrionPrimeraPrueba** a **random_offer**. Además, sin contar con los parámetros modificados, **using_oponent_knowledge** y **offer_type**, se mantiene la configuración de la aplicación y las definiciones de agentes empleadas en la prueba anterior, por lo que se continúan enviando tres ofertas en cada paso de negociación, se imprime el *log* de las ofertas emitidas y se representa gráficamente el proceso de negociación. Así, la salida por consola se muestra en la Figura X y la representación gráfica en la Figura X.

```
*************
* Bilateral *
*************


Step 19 ...

* Deal accepted by dealer JohnSnowPrimeraPrueba at step
19 *

  · Benefits dealer JohnSnowPrimeraPrueba: 0.760000
  · Benefits dealer TyrionPrimeraPrueba: 0.700000
  · Concession dealer JohnSnowPrimeraPrueba: 0.532329
  · Concession dealer TyrionPrimeraPrueba: 0.658842
  · Offer: {'abs': 1, 'color': 2, 'price': 1363}



* Offers *
```

```
- Step 0 from JohnSnowPrimeraPrueba :

        {'abs': 1, 'color': 1, 'price': 4373}
        {'abs': 1, 'color': 1, 'price': 3351}
        {'abs': 1, 'color': 1, 'price': 2482}

- Step 1 from TyrionPrimeraPrueba :

        {'abs': 1, 'color': 2, 'price': 867}
        {'abs': 1, 'color': 2, 'price': 775}
        {'abs': 1, 'color': 2, 'price': 380}

                        ...

- Step 19 from TyrionPrimeraPrueba :

        {'abs': 1, 'color': 2, 'price': 1155}
        {'abs': 0, 'color': 0, 'price': 862}
        {'abs': 1, 'color': 2, 'price': 1363}
```

Figura X. Salida de la prueba con aprendizaje.

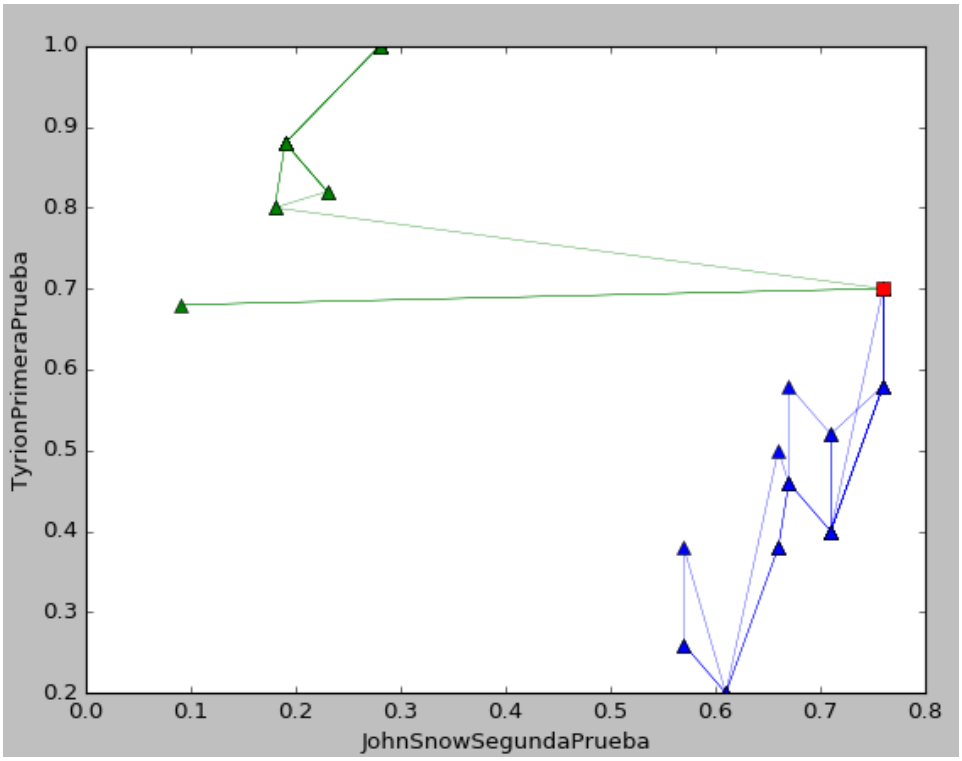


Figura X. Representación gráfica del proceso de negociación en la prueba con aprendizaje.

En este caso, se observa cómo, si ambos agentes emplean conocimiento del oponente se alcanza la misma solución que si no lo emplean (prueba anterior), pero en un menor número de pasos de negociación. Esto ocurre así porque el dominio es muy simple y, posiblemente, la

solución obtenida sea la mejor, por lo que no se puede mejorar, pero sí reducir el número de pasos para alcanzarla. Sin embargo, si el dominio es más complejo, es posible que utilizando aprendizaje se alcancen mejores soluciones.

## 3.3. Torneo

En el apartado anterior se ha llevado a cabo una negociación bilateral entre dos agentes, **TyrionPrimeraPrueba** y **JohnSnowPrimeraPrueba**, sin embargo, la herramienta también permite realizar torneos de negociaciones bilaterales entre todo par de agentes. Para ejecutar el entorno de esta manera, basta con indicar la ruta de más de dos agentes en el campo **agents** del fichero **config.json** que regular el comportamiento de la herramienta.

En este caso, indicamos que queremos realizar un torneo con tres agentes, **TyrionPrimeraPrueba**, **JohnSnowPrimeraPrueba** y **DanerisPrimeraPrueba** (con las mismas preferencias que **TyrionPrimeraPrueba**). Para esto, modificamos el campo **agents** para que quede de la forma mostrada en la Figura X.

```
                                ...
"agents" : {
    "Tyrion" : "./Bots/TyrionPrimeraPrueba.json",
    "JohnSnow" : "./Bots/JohnSnowPrimeraPrueba.json",
    "Daneris" : "./Bots/DanerisPrimeraPrueba.json"

},

                                ...
```

Figura X. Campo **agents** del fichero de configuración **config.json**

Además, en esta prueba solo se emitirá una oferta en cada paso de negociación (**n_offers=1** en **config.json**). Tras esto, se ejecuta la herramienta con **python Main.py** y se observa cómo se realiza una negociación bilateral entre cada par de agentes. Por tanto, si hay **n** pares de agentes, se irán mostrando uno a uno los resultados de las **n** negociaciones, de la misma forma que en las primeras pruebas del apartado anterior. Además, si se le ha especificado a la herramienta que genere una representación gráfica, esta se mostrará al finalizar cada una de las **n** negociaciones. Con ello, las 3 negociaciones que se dan en el torneo de ejemplo se muestran en las siguientes figuras.

```
**************
* Tournament *
**************


Round 0 : DanerisPrimeraPrueba - JohnSnowPrimeraPrueba
```

```
Step  1 ...

* Deal accepted by dealer Daneris at step 1 *

  · Benefits dealer DanerisPrimeraPrueba: 0.700000
  · Benefits dealer JohnSnowPrimeraPrueba: 0.760000
  · Concession dealer DanerisPrimeraPrueba: 0.600000
  · Concession dealer JohnSnowPrimeraPrueba: 0.700000
  · Offer: {'abs': 1, 'price': 1104.0, 'color': 2}


* Offers *

- Step 0 from DanerisPrimeraPrueba :

          {'abs': 0, 'price': 591, 'color': 1}

- Step 1 from JohnSnowPrimeraPrueba :

          {'abs': 1, 'price': 1104.0, 'color': 2}


Round 1 : DanerisPrimeraPrueba - TyrionPrimeraPrueba

Step  1 ...


* Deal accepted by dealer DanerisPrimeraPrueba at step 1
*

  · Benefits dealer DanerisPrimeraPrueba: 1.000000
  · Benefits dealer TyrionPrimeraPrueba: 1.000000
  · Concession dealer DanerisPrimeraPrueba: 0.600000
  · Concession dealer TyrionPrimeraPrueba: 0.900000
  · Offer: {'abs': 1, 'price': 272.0, 'color': 2}


* Offers *

- Step 0 from DanerisPrimeraPrueba :

          {'abs': 0, 'price': 461, 'color': 1}

- Step 1 from TyrionPrimeraPrueba :

          {'abs': 1, 'price': 272.0, 'color': 2}


Round 2 : JohnSnowPrimeraPrueba - TyrionPrimeraPrueba

Step  13 ...

* Deal accepted by dealer JohnSnowPrimeraPrueba at step
13 *

  · Benefits dealer JohnSnowPrimeraPrueba: 0.760000
  · Benefits dealer TyrionPrimeraPrueba: 0.700000
  · Concession dealer JohnSnowPrimeraPrueba: 0.594962
  · Concession dealer TyrionPrimeraPrueba: 0.581839
  · Offer: {'abs': 1, 'price': 1637.0, 'color': 2}
```

```
* Offers *
- Step 0 from JohnSnowPrimeraPrueba :
         {'abs': 1, 'price': 1326.0, 'color': 2}
- Step 1 from TyrionPrimeraPrueba :
         {'abs': 1, 'price': 657.0, 'color': 2}
                              ...
- Step 13 from TyrionPrimeraPrueba :
         {'abs': 1, 'price': 1637.0, 'color': 2}
                              ...
```

Figura X. Salida de la prueba con aprendizaje.

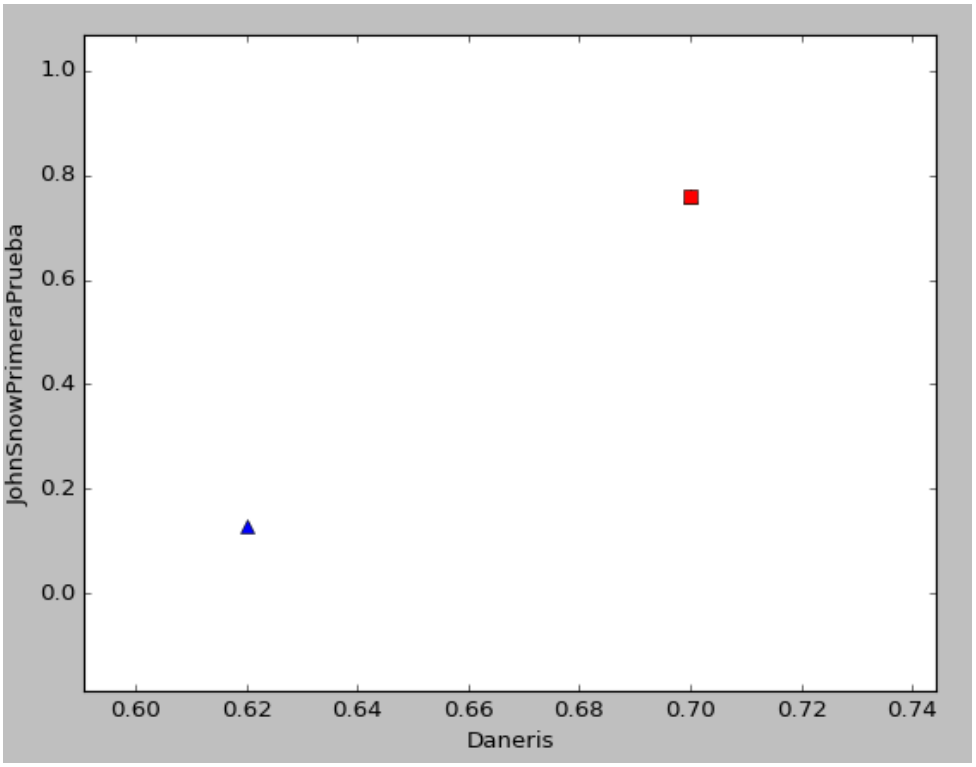


Figura X. Representación gráfica de la negociación entre **JohnSnow** y **Daneris**.

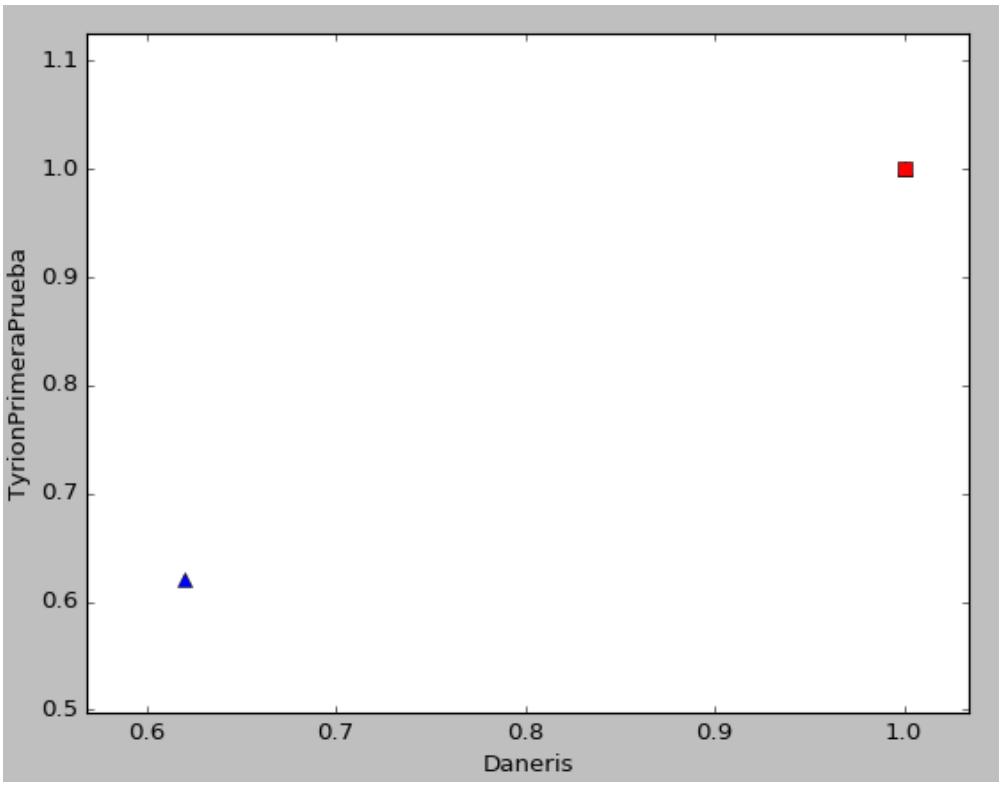


Figura X. Representación gráfica de la negociación entre **Tyrion** y **Daneris**.

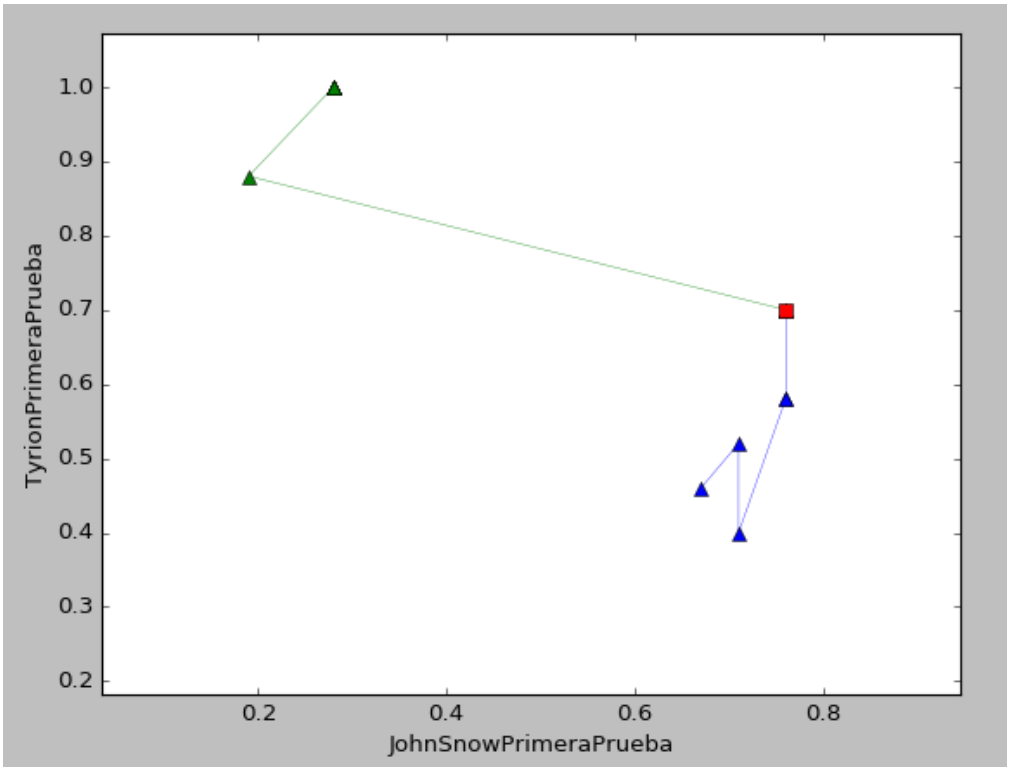


Figura X. Representación gráfica de la negociación entre **Tyrion** y **JohnSnow**.

Además de estas salidas, asociadas a negociaciones bilaterales entre dos agentes, se imprimen por consola varias medidas para evaluar la bondad de los agentes en global, durante todas las negociaciones del torneo en las que ha participado. En el entorno proporcionado, las únicas métricas son los estadísticos que proporciona **numpy** ya comentados en el primer apartado del primer capítulo, pero es posible añadir todos los que se requiera en el script **Statistics.py**. Siguiendo con nuestro ejemplo del torneo entre tres agentes, los resultados de las distintas métricas mostradas por pantalla son:

```
*** Tournament statistics ***

JohnSnowPrimeraPrueba :

        · mean : 0.76
        · min : 0.76
        · max : 0.76
        · std : 0.0
        · var : 0.0
        · first percentile : 0.76
        · second percentile : 0.76
        · third percentile : 0.76

DanerisPrimeraPrueba :

        · mean : 0.85
        · min : 0.7
        · max : 1.0
        · std : 0.15
        · var : 0.0225
        · first percentile : 0.703
        · second percentile : 0.706
        · third percentile : 0.709

TyrionPrimeraPrueba :

        · mean : 0.85
        · min : 0.7
        · max : 1.0
        · std : 0.15
        · var : 0.0225
        · first percentile : 0.703
        · second percentile : 0.706
                · third percentile : 0.709
```

Figura X. Salida de la prueba con aprendizaje.

Donde se observa que los agentes **DanerisPrimeraPrueba** y **TyrionPrimeraPrueba** se comportan mejor en el torneo que el agente **JohnSnowPrimeraPrueba**, principalmente debido a que los dos primeros agentes tienen las mismas preferencias y es fácil que alcancen acuerdos muy beneficiosos para ambos (valor de utilidad 1 para los dos, en este caso).

Esta es la última experimentación comentada, a pesar de que se han realizado muchas más, debido a la falta de tiempo para continuar el trabajo. Sin embargo, es conveniente destacar que todo lo que es posible usar en una negociación bilateral simple, también se puede emplear en torneos p.e. conocimiento sobre el oponente.

# 4. Bibliografía

[1] J. Bergstra, Rémi Bardenete, Yoshua Bengio, Balázs Kégl. Algorithms for hyper-parameter optimization. NIPS proceedings, 2011.

[2] Jones E, Oliphant E, Peterson P, *et al.* **SciPy: Open Source Scientific Tools for Python**, 2001-, http://www.scipy.org/ [Online; accessed 2016-12-30]

[3] Scikit-learn: Machine Learning in Python, Pedregosa *et al.*, JMLR 12, pp. 2825-2830, 2011.

[4] Travis E. Oliphant (2006) Guide to NumPy, Trelgol Publishing, USA

[5] *"Anaconda End User License Agreement". continuum.io. Continuum Analytics. Retrieved May 30, 2016*