



UNIVERSIDAD DE GRANADA

PROYECTO FIN DE DOBLE GRADO EN INGENIERÍA
INFORMÁTICA Y MATEMÁTICAS

Una estrategia para bolsa basada en algoritmos evolutivos y su implementación en una plataforma de trading

Miguel Ángel Torres López

supervisado por
Prof. Jose Manuel Zurita López

April 2, 2019

Contents

1	Fundamentos de Bolsa	3
1.1	Mercado de valores	3
1.2	El valor de las acciones	3
1.3	Corredores de bolsa o brokers	5
2	Plataformas de Trading	6
2.1	MetaTrader5	6
2.2	Backtrader	6
2.2.1	Instalación	6
2.2.2	Preparación del entorno	7
2.2.3	La primera estrategia	8
2.3	Otras plataformas	11
3	Conseguir datos históricos para realizar backtesting	12
3.1	Quandl	12
3.2	YAHOO! Finance	13
4	Algoritmos genéticos	14
4.1	Definición	14
4.1.1	Población	14
4.1.2	Función fitness	14
4.1.3	Cruce	14
4.1.4	Mutación	14
4.2	Pyvolution	14
5	Árbol de Decisión	16
5.1	Definición	16
5.1.1	Etiquetado	16
5.1.2	Conjunto de entrenamiento	17

1 Fundamentos de Bolsa

1.1 Mercado de valores

El mercado de valores o bolsa, es un tipo de mercado en el que se compran y venden productos bursátiles. Para este trabajo, prestaremos principal atención a las acciones, el producto bursátil más simple de un mercado de valores.

Una acción corresponde con una pequeña participación en la empresa a la que pertenezca la acción. De esta forma, inversores privados tienen la posibilidad de comprar porciones de empresas. Por ejemplo, si una empresa está dividida en 200 acciones, tras comprar 50 de estas, seríamos propietarios de una cuarta parte de la empresa. Esto puede reportarnos beneficios de distinta índole según las normas de la empresa. En ocasiones algunas empresas deciden repartir una determinada cantidad de dinero entre sus accionistas en relación a las ganancias obtenidas en un periodo (dividendo). Otras veces, ser poseedor de una gran parte de la empresa nos otorgará derechos dentro de la misma, como por ejemplo participar en las decisiones importantes que se tomen.

El mercado de valores español abre de 9:00 a 17:30 de lunes a viernes. Cabe destacar que hay una aleatoriedad de 30 segundos y que, además, después del cierre hay un periodo de subasta de 5 minutos para evitar manipulaciones de los precios. No vamos a entrar en cómo funciona este sistema.

1.2 El valor de las acciones

El valor de una acción no es fijo, va cambiando según la oferta y la demanda de las acciones de la empresa. Esto convierte a las acciones en un producto muy conveniente para especular.

El mecanismo de compra y venta es el que define el valor de una acción. Para comprar o vender, un inversor debe enviar a bolsa una orden de compra o venta, respectivamente. Tras enviar la orden, esta queda registrada y será ejecutada cuando sea posible. Aclaremos esto con un ejemplo sencillo:

Supongamos que existe una empresa que tiene en el mercado cuatro acciones. Cada una de ellas pertenece a un propietario distinto, llamémosles A, B, C y D.

A tiene una orden de venta de su acción a 4 euros y C tiene una orden de compra de una acción a 1 euro. En esta situación ninguno puede vender ni comprar y el precio está situado en 4 euros por acción. Ahora bien, B decide vender su acción por 3 euros. En el momento en el que realiza una orden de

venta, el precio de las acciones baja a 3 euros. Esto no quiere decir que A venda su acción a este precio. Para bajar de 4 a 3 euros debería cancelar la orden y realizar una nueva. En esta situación no se puede ejecutar ninguna orden en el mercado, como se puede ver en la figura 1.

Órdenes de compra	Precio (euros)	Órdenes de venta
1	5	
	4	1
	3	1
	2	
	1	

Figure 1: Situación de las ordenes

Tras ver que el precio de la acción a bajado de 4 a 3 euros, el inversor D decide comprar y sitúa una orden de compra a 3 euros de una acciones. Entonces, la orden de venta de C y la orden de compra de D pueden ejecutarse. Ahora D pasa a tener 2 acciones y el mercado se queda con las dos órdenes de A y B, provocando que el precio de la acción vuelva a subir a 4 euros.

Además de las órdenes vistas en el ejemplo, existen otras un poco más complejas:

- **Orden de mercado.** Se compran todas las acciones que se emitan en la orden al mejor precio posible. En caso de compra a los precios más bajos, en caso de venta a los precios más altos. Si B hubiera emitido una orden de mercado de una acción, habra comprado la acción de A a 4 euros.
- **Orden limitada.** Se compran todas las acciones que se emitan en la orden siempre y cuando su precio esté situado en el límite establecido. Si D hubiera emitido una orden limitada a 3 euros de dos acciones, hubiera comprado por 3 euros la acción de C pero no la de A, que está a 4 euros. La orden permanece vigente hasta que la suma de las acciones sea comprada.
- **Orden on Stop.** No se hace efectiva hasta que el precio de las acciones llega al precio de la orden. En ese momento, la orden se convierte en una orden de mercado.

- **Orden Stop-Limit.** Este tipo de orden no se hace efectiva hasta que el precio de las acciones llega al precio de la orden. En ese momento, la orden se convierte en una orden limitada. Es muy frecuente que las plataformas ofrezcan este tipo de orden, pues funciona como seguro para no perder todo el capital cuando las acciones caen de precio rápidamente y el inversor posee acciones de la empresa.

1.3 Corredores de bolsa o brokers

Para agilizar las transacciones en bolsa, no se permite que particulares envíen órdenes al mercado. Esta tarea se delega en los corredores de bolsa o *brokers*. La entidad de *broker* se encarga de captar inversores, colocar órdenes y presentar a compradores con vendedores. Para ser broker se debe aprobar un examen y demostrar que se tienen los conocimientos necesarios. Por este trabajo se lleva una comisión que todo inversor debe abonar.

Las comisiones pueden ser de dos tipos:

- **Comisiones fijas.** Son un valor fijo que cada broker cobra. Puede ser por transacción y por tiempo de servicio. Las comisiones fijas son beneficiosas para inversores con un gran capital.
- **Comisiones variables.** Son precios variables que dependen del pas de inversin, la cantidad invertida e incluso el tiempo. Las comisiones variables afectan en misma medida a inversores grandes y pequeños.

La mayoría de los corredores de bolsa tienen comisiones mixtas. No corresponde a este trabajo discutir más allá sobre este tema. De ahora en adelante asumiremos unas comisiones variables.

2 Plataformas de Trading

2.1 MetaTrader5

Podemos observar que es una plataforma ampliamente usada, por tanto es fácil encontrar ejemplos iniciales o ayuda técnica. Se nos oferta en la página web de la plataforma[4] una demo gratuita con acceso a casi todos los recursos.

La programación de las estrategias se divide en varios archivos con unas especificaciones concretas. No obstante, existe una tienda empujada en la plataforma para comprar y vender estrategias, índices o bibliotecas con otros usuarios. Cada producto está puntuada por los usuarios que la han usado.

A pesar de todas estas ventajas, no usaremos MetaTrader5 por ser de carácter privativo. Además, el lenguaje de programación de las estrategias es MQL5, un lenguaje propio de la plataforma que nos impide usar con facilidad bibliotecas externas.

2.2 Backtrader

Backtrader es un framework de libre licencia realizado en el lenguaje Python. La página web de la plataforma[3] contiene una amplia documentación en la que se explican todos los componentes del framework. Asimismo encontramos el método de instalación y un tutorial de iniciación.

El framework viene con indicadores ya programados, aunque te permite hacer más de forma manual. Para producir gráficos hay que instalar una librería adicional, esto no es problema porque viene integrado en la instalación como veremos.

2.2.1 Instalación

Pasamos ahora a su instalación. Necesitamos tener la versión 2.7 de Python, tal y como se indica en la documentación de backtrader[1] que podemos encontrar en su página web. Para comprobar la versión podemos usar el comando *python -Version*. En caso de no tener instalado Python o no tener la versión indicada, podemos instalarlo usando *sudo apt-get install python2.7*.

El framework backtrader está disponible, desde el código fuente en Github. No obstante, también esta disponible para el instalador de paquetes de Python pip. Si no está instalado pip, podemos hacerlo con el comando *sudo*

apt-get install python-pip.

Una vez llegados a este punto, podemos optar por una versión con posibilidad de generar gráficas o no. Para facilitar el análisis de resultados, instalaremos la versión con plotting con el comando *pip install backtrader[plotting]* como podemos ver en la figura 2. El mismo comando quitando la directiva *[plotting]* instala la versión sin gráficas.

```
miguel@miguelpc:~$ pip install backtrader[plotting]
Collecting backtrader[plotting]
  Downloading https://files.pythonhosted.org/packages/.../backtrader-1.9.7-py3-none-any.whl (419kB)
  100% |#####| 419kB 2.4MB/s
Collecting matplotlib; extra == "plotting" (from backtrader[plotting])
  Downloading https://files.pythonhosted.org/packages/.../matplotlib-3.1.0-py3-none-any.whl (12.6MB)
  100% |#####| 12.6MB 14MB/s
Collecting pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 (from matplotlib; extra == "plotting")
  Downloading https://files.pythonhosted.org/packages/.../pyparsing-2.4.6-py2.py3-none-any.whl (61kB)
  100% |#####| 61kB 6.6MB/s
Collecting backports.functools-lru-cache (from matplotlib; extra == "plotting")
  Downloading https://files.pythonhosted.org/packages/.../backports.functools-lru-cache-1.6.2-py2.py3-none-any.whl (102kB)
  100% |#####| 102kB 8.1MB/s
Collecting pytz (from matplotlib; extra == "plotting")
  Downloading https://files.pythonhosted.org/packages/.../pytz-2018.9-py2.py3-none-any.whl (512kB)
  100% |#####| 512kB 2.9MB/s
Collecting six>=1.10 (from matplotlib; extra == "plotting")
  Downloading https://files.pythonhosted.org/packages/.../six-1.11.0-py2.py3-none-any.whl (10kB)
  100% |#####| 10kB 4.4MB/s
Collecting python-dateutil>=2.1 (from matplotlib; extra == "plotting")
  Downloading https://files.pythonhosted.org/packages/.../python-dateutil-2.6.0-py2.py3-none-any.whl (215kB)
  100% |#####| 215kB 4.4MB/s
Collecting kiwisolver>=1.0.1 (from matplotlib; extra == "plotting")
  Downloading https://files.pythonhosted.org/packages/.../kiwisolver-1.0.1-py2.py3-none-any.whl (952kB)
  100% |#####| 952kB 1.7MB/s
Collecting cycler>=0.10 (from matplotlib; extra == "plotting")
  Downloading https://files.pythonhosted.org/packages/.../cycler-0.10.0-py2.py3-none-any.whl (12.1MB)
  100% |#####| 12.1MB 14MB/s
Collecting numpy>=1.7.1 (from matplotlib; extra == "plotting")
  Downloading https://files.pythonhosted.org/packages/.../numpy-1.16.0-py2.py3-none-any.whl (12.1MB)
  100% |#####| 12.1MB 14MB/s
Collecting setuptools (from kiwisolver>=1.0.1->matplotlib; extra == "plotting")
  Downloading https://files.pythonhosted.org/packages/.../setuptools-40.6.0-py2.py3-none-any.whl (573kB)
  100% |#####| 573kB 2.6MB/s
```

Figure 2: Resultado de instalar backtrader con plotting

Observamos que se han instalado otros paquetes adicionales. En concreto se ha instalado el paquete numpy, que más tarde nos será útil como herramienta matemática.

2.2.2 Preparación del entorno

Una de las estructuras de datos más comunes en backtrader son las líneas. Una línea no es más que un conjunto de pares ordenados por uno de los elementos de menor a mayor. Por ejemplo, al hablar de un valor de mercado tenemos 5 líneas: valor de apertura, valor de salida, valor máximo, valor mínimo y volumen de compra/venta.

Al acceder a cada línea hay que tener en cuenta que no usa la indexación de un vector en un lenguaje común. En su lugar, el índice 0 representa el

instante actual y para acceder a instantes anteriores usaremos índices negativos, por ejemplo:

```
self.sma = SimpleMovingAverage(...)
valor_actual = self.sma[0]
valor_instante_anterior = self.sma[-1]
```

Antes de ver como crear una estrategia, vamos a ver como podemos simular una prueba introduciendo unos datos y un presupuesto inicial. Para ello, en un archivo con formato py para que podamos ejecutar con Python escribimos el siguiente código:

```
from __future__ import (absolute_import, division, print_function, unicode_literals)

import datetime
import os.path
import sys
import backtrader as bt # Importar todas las herramientas de backtrader

if __name__ == '__main__':
    cerebro = bt.Cerebro()

    # Crear un paquete de datos
    data = bt.feeds.YahooFinanceCSVData(
        dataname='YAHOO', # Ruta absoluta donde se encuentran los datos descargados de YAHOO! Finance
    # Fecha inicial de los datos
        fromdate=datetime.datetime(2000, 1, 1),
    # Fecha final de los datos
        todate=datetime.datetime(2000, 12, 31),
        reverse=False)

    # Activar los datos en el cerebro
    cerebro.adddata(data)
    # Establecer dinero inicial
    cerebro.broker.setcash(100000.0)

    print('Starting Portfolio Value: %.2f' % cerebro.broker.getvalue())
    cerebro.run() # EJECUTAR BACKTESTING (AHORA MISMO, SIN NINGUNA ESTRATEGIA)
    print('Final Portfolio Value: %.2f' % cerebro.broker.getvalue())
```

Notemos que se trata de una archivo Python en el que solo hemos tenido que importar el paquete *backtrader*. Por tanto, en el caso de necesitar otros paquetes para realizar nuestra estrategia, solo tenemos que incorporarlos de forma habitual.

En este ejemplo, no le hemos especificado a la clase *cerebro* cuál va a ser la estrategia a seguir. Por este motivo, al hacer la simulación el presupuesto inicial y final no varían. Para ejecutar el backtesting simplemente se ejecuta el script de Python con el comando *python rutadelscript.py* en terminal. Nos dará un fallo, la ruta donde debería estar el fichero con los datos está vacía, pues no hemos descargado ningún archivo de datos. Este tema se abordará en la sección 3.

2.2.3 La primera estrategia

Antes de programar la primera estrategia de *trading*, vamos a ilustrar con una estrategia fútil cómo estructurar una clase para que backtrader pueda

trabajar con ella como tal. La estructura básica es una clase con un método `__init__` y otro método llamado `next`.

El primero puede usarse para instanciar y agrupar los datos que requiere nuestra estrategia. El segundo método es llamado en cada instante por backtrader. Consideramos que un instante es cada una de las marcas temporales que hay registradas en nuestros datos. Así pues, los datos recogidos en `__init__` son la referencia para llamar al método `next`. Cabe destacar que en cada llamada al segundo método solo son accesibles los datos con marcas temporales menores o iguales al instante correspondiente a la llamada.

Veamos un ejemplo:

```
# Crear una estrategia
class TestStrategy(bt.Strategy):

    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.date(0)
        print('%s, %s' % (dt.isoformat(), txt))

    def __init__(self):
        # Guarda una referencia de la línea de valores de cierre
        self.dataclose = self.datas[0].close

    def next(self):
        # Muestra por pantalla el valor de cierre
        self.log('Close, %.2f' % self.dataclose[0])
```

Al ejecutar esta estrategia, veremos el valor de cierre de cada instante y, si los datos lo facilitan, la marca temporal asociada a cada instante.

Para activar la estrategia es necesario indicar al *cerebro* la clase creada con la siguiente línea, que puede ser situada justo después de indicar el presupuesto inicial.

```
# Registrar estrategia
cerebro.addstrategy(TestStrategy)
```

La estrategia está completa, pero no realizamos ninguna compra ni venta. Para nuestro posterior estudio es imprescindible introducir estas acciones. Sin embargo, no entraremos en profundidad en todos los aspectos de la compraventa, esto se hará en la sección ???. Veamos el código de una nueva táctica de inversión:

```
class DoubleDownStrategy(bt.Strategy):

    def log(self, txt, dt=None):
        dt = dt or self.datas[0].datetime.date(0)
        print('%s, %s' % (dt.isoformat(), txt))

    def __init__(self):
        self.dataclose = self.datas[0].close

        # Para mantener las ordenes no ejecutadas
        self.order = None

    def notify_order(self, order):
        if order.status in [order.Submitted, order.Accepted]:
            return

        if order.status in [order.Completed]:
            if order.isbuy():
                self.log('BUY EXECUTED, %.2f' % order.executed.price)
            elif order.issell():
                self.log('SELL EXECUTED, %.2f' % order.executed.price)
```

```

        self.bar_executed = len(self)

    elif order.status in [order.Canceled, order.Margin, order.Rejected]:
        self.log('Order Canceled/Margin/Rejected')

    self.order = None

    def next(self):
        self.log('Close, %.2f' % self.dataclose[0])
        # Si hay una compraventa pendiente no puedo hacer otra
        if self.order:
            return

        # Si no tengo nada adquirido
        if not self.position:
            if self.dataclose[0] < self.dataclose[-1]:
                if self.dataclose[-1] < self.dataclose[-2]:
                    self.log('BUY CREATE, %.2f' % self.dataclose[0])
                    self.order = self.buy()

            else:
                # Ya hemos adquirido algo
                if len(self) >= (self.bar_executed + 5):
                    self.log('SELL CREATE, %.2f' % self.dataclose[0])
                    self.order = self.sell()

```

Aquí introducimos varios nuevos conceptos del framework. En primer lugar, las acciones *self.buy()* y *self.sell()* ejecutadas dentro del método *next* indican que queremos lanzar una orden de compra o de venta, respectivamente. Cuando no se especifica, backtrader compra o vende al precio de cierre del instante actual una acción. Esto no quiere decir que la orden se ejecute en ese instante, si no que, a partir de ese momento y si el precio del producto lo permite, se realizará. Notar que una vez lanzada una orden hay que esperar a que se complete o se cancele antes de lanzar otra.

Esta situación hace necesario incluir el método *notify_order*. Como su propio nombre sugiere, es llamado cuando una orden cambia de estado. En este ejemplo, cuando una orden es completada, mostramos por pantalla si es era de compra o venta y el precio de la misma. Las ordenes pueden ser canceladas o rechazadas. Un ejemplo de este hecho sería intentar comprar con un presupuesto insuficiente.

Por último, comentaremos la parte lógica de la estrategia. En cada instante, el método *next* evalúa alguno de los siguientes casos:

- **Existe una compraventa lanzada y no terminada.** En este caso debemos esperar. Notar que backtrader permite cancelar una orden, aunque no entraremos en este punto.
- **No hay una orden lanzada y tampoco tengo nada comprado.** Esto puede comprobarse con *self.position*, que devuelve negativo si no se tiene nada en posesión. En esta situación solo cabe esperar una orden de compra y para este ejemplo la lanzaremos si los últimos dos instantes han bajado su precio de cierre.

- **No hay una orden lanzada pero tengo algo comprado.** Para ilustrar una nueva herramienta, vamos a lanzar una orden de venta 5 instantes después de haber comprado. Para ello podemos comprobar la longitud de *self* con la función *len* de Python.

2.3 Otras plataformas

- **Tradestation** permite programar estrategias, pero no hacer *backtesting*. Además no es de acceso gratuito.
- **Cloud9trader** tiene una demo que permite programar estrategias y hacer *backtesting*. Está bien para probar estrategias sencillas basadas en indicadores ampliamente conocidos. No obstante, la programación hay que hacerla en la ventana del navegador, con un lenguaje propio y sin posibilidad de usar paquetes externos.
- **Plus500** es una plataforma de trading con escasas posibilidades de automatización. Tan solo permite algunas órdenes sencillas condicionales preprogramadas.
- **PyAlgoTrade** es un proyecto desarrollado en Python disponible en Github. Tiene posibilidad de *backtesting*, utilizar paquetes externos y, en el caso de estar haciendo *trading* con Bitcoins, comprar y vender estos. Los datos del mercado hay que conseguirlos de forma externa en formato CSV.

3 Conseguir datos históricos para realizar back-testing

En esta sección veremos como conseguir archivos de los valores históricos en bolsa de diferentes empresas. A parte de conseguir los ficheros, vamos a incorporar los datos a backtrader para su posterior uso.

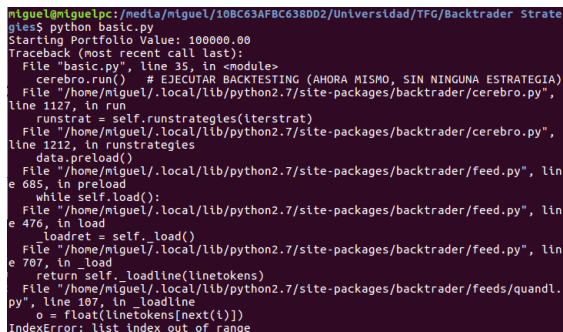
3.1 Quandl

Quandl es una empresa que reúne miles de paquetes de datos financieros de todo el mundo. Para acceder a esta información es necesario registrarse en su página web. Una vez tengamos acceso, es posible descargar muchos de los archivos en formato csv. Aunque quizás la opción más cómoda para este proyecto es utilizar la api que ofrece. De esta forma podemos realizar el acceso a los datos desde Python, sin necesidad de hacer una descarga previa a la ejecución del script.

Para usar la api seguiremos los pasos de instalación que se pueden encontrar en la documentación[2] de Quandl. Instalamos el paquete quandl con el gestor de paquetes pip. Para usar el paquete quandl, debemos indicar a Quandl la *api_key* que nos dan al inscribirnos en su página.

```
# Crear un paquete de datos con QUANDL
data = bt.feeds.Quandl(
    dataset='WFE',
    fromdate = datetime.datetime(2016,1,1),
    todate = datetime.datetime(2017,1,1),
    dataname='INDEXES.BMESPANISHEXCHANGESMADRID',
    buffered=True,
    apikey='')
```

No obstante, como puede verse en la figura 3 backtrader tiene algún error interno al usar esta api. Probablemente, como en este foro de la comunidad se apunta[5], se debe a una mala lectura de los vectores de datos recibidos.



```
miguel@miguelpc:/media/miguel/108C63AFBC638DD2/Universidad/TFG/Backtrader Strate
gies$ python basic.py
Starting Portfolio Value: 100000.00
Traceback (most recent call last):
  File "basic.py", line 35, in <module>
    cerebro.run() # EJECUTAR BACKTESTING (AHORA MISMO, SIN NINGUNA ESTRATEGIA)
  File "/home/miguel/.local/lib/python2.7/site-packages/backtrader/cerebro.py",
line 1127, in run
    runstrat = self.runstrategies(iterstrat)
  File "/home/miguel/.local/lib/python2.7/site-packages/backtrader/cerebro.py",
line 1212, in runstrategies
    data.preload()
  File "/home/miguel/.local/lib/python2.7/site-packages/backtrader/feed.py", lin
e 685, in preload
    while self.load():
  File "/home/miguel/.local/lib/python2.7/site-packages/backtrader/feed.py", lin
e 476, in load
    loadret = self._load()
  File "/home/miguel/.local/lib/python2.7/site-packages/backtrader/feed.py", lin
e 707, in _load
    return self._loadline(linetokens)
  File "/home/miguel/.local/lib/python2.7/site-packages/backtrader/feeds/quandl.
py", line 107, in _loadline
    o = float(linetokens[next(i)])
IndexError: list index out of range
```

Figure 3: Ejecución del script recolectando datos de Quandl.

3.2 YAHOO! Finance

Una de las opciones más completas para conseguir los datos es la página web de YAHOO! Finance (<https://finance.yahoo.com>). Basta con buscar el índice o la empresa sobre la que se quieren los datos, indicar las fechas y la frecuencia de muestreo y pinchar en el botón de descarga. Los datos se descargan en formato CSV. Este formato es ampliamente utilizado por la comunidad científica para almacenar grandes cantidades de datos.

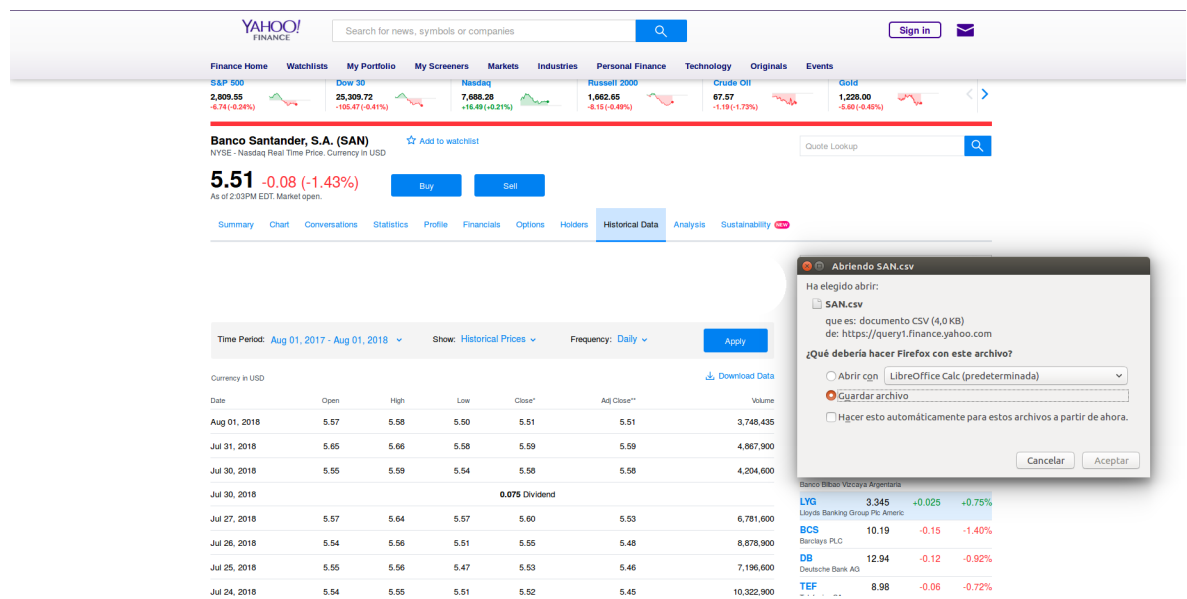


Figure 4: Descarga de los datos históricos de Banco Santander.

Por ejemplo, imaginemos que buscamos los valores históricos de Banco Santander desde el 1 de agosto de 2017 hasta la misma fecha del año siguiente. Para ello indicamos la empresa, nos vamos al apartado de *Historical Data* e indicamos las fechas. El resultado se muestra en la figura 4.

Para cargar estos datos en backtrader basta con indicar la ruta y el rango de fechas con el siguiente código:

```
# Crear un paquete de datos con YAHOO FINANCE
data = bt.feeds.YahooFinanceCSVData(
    dataname='Data/SAN.csv',
    fromdate=datetime.datetime(2017, 8, 1),
    todate=datetime.datetime(2018, 8, 1),
    reverse=False)

# Activar los datos en el cerebro
cerebro.adddata(data)
```

4 Algoritmos genéticos

4.1 Definición

4.1.1 Población

4.1.2 Función fitness

4.1.3 Cruce

4.1.4 Mutación

4.2 Pyvolution

Pyvolution es un paquete de Python de algoritmos genéticos. Aunque la última versión fue lanzada en 2012, es un paquete completo y sencillo de utilizar. En apenas 30 líneas y sin necesidad de dar muchas especificaciones se pueden realizar algoritmos completos. Una ejecución normal consta de varios parámetros donde se pueden controlar las generaciones, los individuos por generación, probabilidad y severidad de las mutaciones, cantidad individuos elitistas e incluso el tiempo máximo de ejecución.

Para ilustrar brevemente su uso, muestro uno de los ejemplos que vienen en la página de Github del paquete.

```
import math
from pyvolution.EvolutionManager import *
from pyvolution.GeneLibrary import *

"""
Queremos calcular una solución del siguiente sistema:
a + b + c - 17 = 0
a^2 + b^2 - 5 = 0
"""

def fitnessFunction(chromosome):
    """
    Dado un "cromosoma", esta es la función que calcula su puntuación
    La puntuación es una float mayor que 0.
    """

    #Accedemos a los cromosomas o valores a ajustar
    a = chromosome["a"]
    b = chromosome["b"]
    c = chromosome["c"]
    d = chromosome["d"]

    #Calculamos los valores que nos gustaría que fueran 0
    val1 = math.fabs(a + b + c - 17)
    val2 = math.fabs(math.pow(a, 2) + math.pow(b, 2) - 5)

    #La función distancia agrupa los valores para una mejor puntuación
    dist = math.sqrt(math.pow(val1, 2) + math.pow(val2, 2))

    if dist != 0:
        return 1 / dist # Cuanto menor sea la distancia mayor será la puntuación
    else:
        return None #Devolver None indica que los cromosomas han sido ajustados

#Configuramos el algoritmo genético
em = EvolutionManager(
    fitnessFunction,
    individualsPerGeneration=100,
```



```

        mutationRate=0.2,          #Probabilidad de mutacion
        maxGenerations=1000,
        stopAfterTime=10,         #Para simulacion tras 10 segundos
        elitism=2,                 #Mantener los 2 mejores de cada generacion
    )

    #Creamos una funcion de mutacion inversamente proporcional a la bondad del ajuste
    mutator = FloatInverseFit("mut", maxVal=0.01, startVal=1)

    #Indicamos que los puntos iniciales se toman siguiendo una distribucion normal
    #de media 0 y desviacion 100. Ademas marcamos la mutacion como la definida antes.
    atype = FloatGeneType("a", generatorAverage=0, generatorSTDEV=100, mutatorGene="mut")
    btype = FloatGeneType("b", generatorAverage=0, generatorSTDEV=100, mutatorGene="mut")
    ctype = FloatGeneType("c", generatorAverage=0, generatorSTDEV=100, mutatorGene="mut")

    #Registramos los parametros y la mutacion
    em.addGeneType(mutator)
    em.addGeneType(atype)
    em.addGeneType(btype)
    em.addGeneType(ctype)

    #Ejecutamos
    result = em.run()

```

5 Árbol de Decisión

Los árboles se utilizan en diversos campos de la informática y, en concreto, en la toma de decisiones y/o predicciones. A pesar de que son difíciles de representar y visualizar de forma global, resultan una herramienta muy útil para separar situaciones o datos según una serie de condiciones dadas previamente.

5.1 Definición

Un grafo simple es un grafo sin lazos ni aristas múltiples entre sus vértices. Un árbol es un grafo simple G tal que para cada dos vértices de G existe un único camino simple entre ellos.

Un árbol de decisión será, por tanto, un árbol compuesto por:

- Un vértice especial, llamado nodo raíz, que se considera el inicio del camino de decisión.
- Un conjunto de vértices, llamados nodos, conectados con al menos otros 2 vértices. En cada nodo hay una condición. La verificación de dicha condición determina que nodo es el siguiente en el camino de decisión.
- Un conjunto de vértices, llamados hojas o nodos terminales, conectados con un único vértice. En cada hoja hay una clasificación del dato que cumple todas las condiciones que tienen los nodos del camino desde la raíz hasta la hoja.

El objetivo de un árbol de decisión es, por tanto, clasificar datos de una determinada naturaleza según una serie de condiciones. La construcción de dicho árbol y la elección de las condiciones las veremos en los siguientes apartados.

5.1.1 Etiquetado

Para poder clasificar un dato, es necesario saber cuáles son las posibles clasificaciones o, como se suele nombrar, cuáles son las clases. Esta clase viene dada por la etiqueta o *tag*.

En el caso que nos ocupa, el *trading* en bolsa, no se tiene una clasificación natural. Crear estas clases de manera que representen un buen momento para comprar o vender será objeto de estudio en nuestro trabajo.

5.1.2 Conjunto de entrenamiento

Una vez que la estructura del árbol esta hecha es sencillo clasificar un dato. Basta con ir comprobando las condiciones de cada nodo y continuar el recorrido del grafo según los resultados de estas. Cuando una condición nos dirija a una hoja, encontraremos la clase predicha para ese dato concreto. Pero, cómo podemos crear el árbol?

Aquí entra en juego el conjunto de entrenamiento. Un conjunto de entrenamiento es un compendio de datos, de la misma naturaleza que los que queremos clasificar, de los que ya sabemos su clase real. Por tanto, a partir de este conjunto, deberemos inducir un árbol de decisión cuyas hojas clasifiquen bien los datos conocidos. De esta forma, si los datos con clase desconocida tienen una procedencia parecida a los datos del conjunto de entrenamiento, serán correctamente clasificados. O al menos esta es la intención.

References

- [1] Documentación de backtrader. 2015. (<https://www.backtrader.com/docu/index.html>).
- [2] Documentación de quandl. 2018. (<https://docs.quandl.com/docs/python-installation>).
- [3] Página web de backtrader. 2018. (<https://www.backtrader.com>).
- [4] Página web de metatrader5. 2018. (<https://www.metatrader5.com>).
- [5] Quand data feed - futures data. 2018. (<https://community.backtrader.com/topic/797/quandl-data-feed-futures-data>).