Traducido del inglés al español - www.onlinedoctranslator.com

- Manejador de ejecución -El ExecutionHandler simula una conexión a un corretaje. El trabajo del controlador es tomar OrderEvents de la Cola y ejecutarlos, ya sea a través de un enfoque simulado o una conexión real a un corredor de hígado. Una vez que se ejecutan las órdenes, el manejador crea FillEvents, que describen lo que realmente se operó, incluidas las tarifas, la comisión y el deslizamiento (si se modelaron).
- Prueba retrospectiva -Todos estos componentes están envueltos en un bucle de eventos que maneja correctamente todos los tipos de eventos, enrutándolos al componente apropiado.

A pesar de la cantidad de componentes, este es un modelo bastante básico de un motor comercial. Existe un margen significativo para la expansión, particularmente en lo que respecta a cómo se utiliza la Cartera. Además, los diferentes modelos de costos de transacción también podrían abstraerse en su propia jerarquía de clases.

14.2.1 Eventos

El primer componente que se discutirá es la jerarquía de clases de eventos. En esta infraestructura hay cuatro tipos de eventos que permiten la comunicación entre los componentes anteriores a través de una cola de eventos. Son MarketEvent, SignalEvent, OrderEvent y FillEvent.

Evento

La clase padre en la jerarquía se llama Evento. Es una clase base y no proporciona ninguna funcionalidad o interfaz específica. Dado que en muchas implementaciones los objetos Event probablemente desarrollarán una mayor complejidad, por lo tanto, se está "preparando para el futuro" mediante la creación de una jerarquía de clases.

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# evento.py

de__futuro__importarimprimir_funcion


claseEvento (objeto):
    """
    El evento es una clase base que proporciona una interfaz para todos los
    eventos posteriores (heredados), que desencadenarán más eventos en la
    infraestructura comercial.
    """
    pasar
```

MercadoEvento

Los MarketEvents se activan cuando el ciclo while externo del sistema de backtesting comienza un nuevo "latido". Ocurre cuando el objeto DataHandler recibe una nueva actualización de datos de mercado para cualquier símbolo que se esté rastreando actualmente. Se utiliza para activar el objeto Estrategia generando nuevas señales comerciales. El objeto de evento simplemente contiene una identificación de que es un evento de mercado, sin otra estructura.

```
# evento.py

claseEvento de mercado (evento):
    """
    Maneja el evento de recibir una nueva actualización del mercado con las barras
    correspondientes.
    """

    definitivamente__en si mismo):
```

```
        """
        Inicializa MarketEvent. """

        self.type = 'MERCADO'
```

SeñalEvento

El objeto Estrategia utiliza datos de mercado para crear nuevos SignalEvents. SignalEvent contiene un ID de estrategia, un símbolo de cotización, una marca de tiempo de cuándo se generó, una dirección (larga o corta) y un indicador de "fuerza" (esto es útil para las estrategias de reversión a la media). Los SignalEvents son utilizados por el objeto Portfolio como consejo sobre cómo operar.

```
# evento.py

clase SignalEvent(Evento):
    """
    Maneja el evento de enviar una señal desde un objeto de estrategia. Esto es
    recibido por un objeto de Portafolio y se actúa en consecuencia.
    """

    definitivamente __init__(self, id_estrategia, símbolo, fecha y hora, tipo_señal, fuerza):
        """
        Inicializa SignalEvent.

        Parámetros:
        id_estrategia: el identificador único de la estrategia que
            generó la señal.
        símbolo: el símbolo de cotización, por ejemplo, 'GOOG'.
        datetime: la marca de tiempo en la que se generó la señal. tipo_señal -
        'LARGO' o 'CORTO'.
        fuerza - Una "sugerencia" de factor de ajuste utilizada para escalar
            cantidad a nivel de cartera. Útil para estrategias de parejas.
        """

        self.tipo = 'SEÑAL'
        self.strategy_id = estrategia_id
        self.symbol = símbolo
        self.datetime = datetime self.signal_type
        = signal_type self.strength = fuerza
```

Evento de pedido

Cuando un objeto Portfolio recibe SignalEvents, los evalúa en el contexto más amplio de la cartera, en términos de riesgo y tamaño de la posición. En última instancia, esto conduce a OrderEvents que se enviarán a un ExecutionHandler.

OrderEvent es un poco más complejo que SignalEvent, ya que contiene un campo de cantidad además de las propiedades mencionadas anteriormente de SignalEvent. La cantidad está determinada por las restricciones de la cartera. Además, OrderEvent tiene un método print_order(), que se utiliza para enviar la información a la consola si es necesario.

```
# evento.py

clase OrderEvent(Evento):
    """
    Maneja el evento de enviar una Orden a un sistema de ejecución. La orden contiene
    un símbolo (por ejemplo, GOOG), un tipo (de mercado o límite),
```

```
    cantidad y una dirección. """

    definitivamente__init__(self, símbolo, order_type, cantidad, dirección):
        """
        Inicializa el tipo de orden, estableciendo si es de Mercado
        ('MKT') o Limitada ('LMT'), tiene una cantidad (integral) y su
        dirección ('COMPRA' o 'VENTA').


        Parámetros:
        símbolo - El instrumento para negociar. order_type - 'MKT' o
        'LMT' para Mercado o Límite. cantidad: entero no negativo
        para la cantidad. dirección - 'COMPRAR' o 'VENDER' para
        largo o corto. """


        self.tipo = 'PEDIDO'
        self.símbolo = símbolo
        self.order_type = order_type
        self.quantity = cantidad self.direction
        = dirección

    definitivamenteimprimir_pedido(auto):
        """
        Da salida a los valores dentro de la Orden. """

        impresión(
            "Orden: Símbolo=%s, Tipo=%s, Cantidad=%s, Dirección=%s" % (self.símbolo,
            self.order_type, self.quantity, self.direction)
        )
```

LlenarEvento

Cuando un ExecutionHandler recibe un OrderEvent, debe tramitar el pedido. Una vez que se ha tramitado un pedido, genera un FillEvent, que describe el costo de compra o venta, así como los costos de transacción, como tarifas o desfase.

El FillEvent es el Evento de mayor complejidad. Contiene una marca de tiempo para cuando se completó una orden, el símbolo de la orden y el intercambio en el que se ejecutó, la cantidad de acciones negociadas, el precio real de la compra y la comisión incurrida.

La comisión se calcula utilizando las comisiones de Interactive Brokers. Para pedidos API de EE. UU., esta comisión es de 1,30 USD como mínimo por pedido, con una tarifa fija de 0,013 USD o 0,08 USD por acción, dependiendo de si el tamaño de la operación es inferior o superior a 500 unidades de acciones.

```
# evento.py

claseLlenarEvento(Evento):
    """
    Encapsula la noción de una orden ejecutada, tal como se devuelve de
    una agencia de corretaje. Almacena la cantidad de un instrumento
    realmente llenado ya qué precio. Además, almacena la comisión de
    la operación de la correduría.
    """

    definitivamente__init__(self, timeindex, símbolo, intercambio, cantidad,
                            dirección, fill_cost, comisión=Ninguno):
```

```
        """
        Inicializa el objeto FillEvent. Establece el símbolo, el intercambio, la
        cantidad, la dirección, el costo de llenado y una comisión opcional.


        Si no se proporciona una comisión, el objeto Rellenar la calculará
        según el tamaño de la operación y las tarifas de Interactive
        Brokers.

        Parámetros:
        timeindex: la resolución de la barra cuando se completó la orden.
        símbolo - El instrumento que se llenó.
        intercambio - El intercambio donde se llenó el pedido. cantidad -
        La cantidad llena.
        dirección: la dirección de llenado ('COMPRAR' o 'VENDER')
        fill_cost: el valor de las existencias en dólares. comisión - Una
        comisión opcional enviada desde IB. """


        self.tipo = 'LLENAR'
        self.timeindex = timeindex
        self.symbol = símbolo
        self.intercambio = intercambio
        self.cantidad = cantidad
        self.direction = dirección
        self.fill_cost = fill_cost

        # Calcular comisión
        sicomisiónesNinguna:
            self.commission = self.calculate_ib_commission() más:

            self.commission = comisión

    definitivamentecalcular_ib_comision(auto):
        """
        Calcula las tarifas de negociación en función de una estructura de
        tarifas de Interactive Brokers para API, en USD.

        Esto no incluye tarifas de cambio o ECN.

        Basado en "Órdenes dirigidas por API de EE. UU.": https://
        www.interactivebrokers.com/en/index.php?
        f=comisión&p=acciones2
        """
        costo_total = 1.3
        siself.cantidad <= 500:
            full_cost = max(1.3, 0.013 * self.quantity) más:#Mayor
        que 500
            full_cost = max(1.3, 0.008 * self.quantity) devolvercosto
            total
```

14.2.2 Manejador de datos

Uno de los objetivos de un sistema de negociación basado en eventos es minimizar la duplicación de código entre el elemento de backtesting y el elemento de ejecución en vivo. Idealmente, sería óptimo utilizar la misma metodología de generación de señales y los mismos componentes de gestión de cartera tanto para el historial

Pruebas y comercio en vivo. Para que esto funcione, el objeto de Estrategia que genera las Señales y el objeto de Cartera que proporciona Órdenes basadas en ellas, deben utilizar una interfaz idéntica a una fuente de mercado tanto para la ejecución histórica como en vivo.

Esto motiva el concepto de una jerarquía de clases basada en un objeto DataHandler, que brinda a todas las subclases una interfaz para proporcionar datos de mercado a los componentes restantes dentro del sistema. De esta forma, cualquier manejador de datos de subclase puede ser "intercambiado", sin afectar la estrategia o el cálculo de la cartera.

Las subclases de ejemplo específicas podrían incluir HistoricCSVDataHandler, QuandlDataHandler, SecuritiesMasterDataHandler, InteractiveBrokersMarketFeedDataHandler, etc. Conjunto de barras Volume-OpenInterest. Esto se puede usar para "alimentar por goteo" barra por barra los datos en las clases de Estrategia y Cartera en cada latido del sistema, evitando así el sesgo de anticipación.

La primera tarea es importar las bibliotecas necesarias. Específicamente será necesario importar pandas y las herramientas de clase base abstracta. Dado que DataHandler genera MarketEvents, también se necesita event.py como se describe anteriormente.

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# datos.py

de__futuro__importarimprimir_funcion

dea B CimportarABCMeta, método abstracto
importarfecha y hora
importaros, os.ruta

importarnumpy como np
importarpandas como pd

deeventoimportarMercadoEvento
```

DataHandler es una clase base abstracta (ABC), lo que significa que es imposible instanciar una instancia directamente. Solo se pueden instanciar subclases. La razón de esto es que ABC proporciona una interfaz a la que deben adherirse todas las subclases DataHandler posteriores, lo que garantiza la compatibilidad con otras clases que se comunican con ellas.

Hacemos uso de la propiedad __metaclass__ para que Python sepa que se trata de un ABC. Además, usamos el decorador @abstractmethod para que Python sepa que el método se anulará en las subclases (esto es idéntico a un*metodo virtual puro*en C++).

Hay seis métodos enumerados para la clase. Los dos primeros métodos, get_latest_bar y get_latest_bars, se utilizan para recuperar un subconjunto reciente de las barras de negociación históricas de una lista almacenada de tales barras. Estos métodos son útiles dentro de las clases de Estrategia y Cartera, debido a la necesidad de estar constantemente al tanto de los precios y volúmenes actuales del mercado.

El siguiente método, get_latest_bar_datetime, simplemente devuelve un objeto de fecha y hora de Python que representa la marca de tiempo de la barra (por ejemplo, una fecha para las barras diarias o un objeto de resolución de minutos para las barras de minutos).

Los siguientes dos métodos, get_latest_bar_value y get_latest_bar_values, son métodos convenientes que se utilizan para recuperar valores individuales de una barra en particular o una lista de barras. Por ejemplo, a menudo ocurre que una estrategia solo está interesada en los precios de cierre. En este caso, podemos usar estos métodos para devolver una lista de valores de punto flotante que representan los precios de cierre de barras anteriores, en lugar de tener que obtenerlos de la lista de objetos de barra. Esto generalmente aumenta la eficiencia de las estrategias que utilizan una "ventana retrospectiva", como las que involucran regresiones.

El método final, update_bars, proporciona un mecanismo de "alimentación por goteo" para colocar información de barras en una nueva estructura de datos que prohíbe estrictamente el sesgo de búsqueda anticipada. Esta es una de las diferencias clave entre un sistema de backtesting basado en eventos y uno basado en vectorización. Tenga en cuenta que se generarán excepciones si se produce un intento de instanciación de la clase:

```
# datos.py

clase Manejador de datos (objeto):
    """
    DataHandler es una clase base abstracta que proporciona una interfaz para todos los
    controladores de datos posteriores (heredados) (tanto en vivo como históricos).

    El objetivo de un objeto DataHandler (derivado) es generar un conjunto de barras
    generado (OHLCVI) para cada símbolo solicitado.

    Esto replicará cómo funcionaría una estrategia en vivo, ya que los datos actuales del
    mercado se enviarían "por la tubería". Por lo tanto, un sistema histórico y en vivo será
    tratado de manera idéntica por el resto de la suite de backtesting. """


    __metaclase__ = ABCMeta

    @metodoabstracto
    definitivamente get_latest_bar(uno mismo, símbolo):
        """
        Devuelve la última barra actualizada.
        """
        elevar NotImplementedError("Debería implementar get_latest_bar()")

    @metodoabstracto
    definitivamente get_latest_bars(self, símbolo, N=1):
        """
        Devuelve las últimas N barras actualizadas.
        """
        elevar NotImplementedError("Debería implementar get_latest_bars()")

    @metodoabstracto
    definitivamente get_latest_bar_datetime(yo, símbolo):
        """
        Devuelve un objeto de fecha y hora de Python para la última barra. """

        elevar NotImplementedError("Debe implementar
            get_latest_bar_datetime()")

    @metodoabstracto
    definitivamente get_latest_bar_value(self, símbolo, val_type):
        """
        Devuelve uno de los valores Abierto, Alto, Bajo, Cerrado, Volumen u
        OI de la última barra.
        """
        elevar NotImplementedError("Debe implementar
            get_latest_bar_value()")

    @metodoabstracto
    definitivamente get_latest_bars_values(self, símbolo, val_type, N=1):
        """
        Devuelve los últimos N valores de barra de la lista
        latest_symbol, o Nk si hay menos disponibles. """

        elevar NotImplementedError("Debe implementar
            get_latest_bars_values()")
```

```
    @metodoabstracto
    definitivamenteactualizar_barras(auto):
        """
        Empuja las últimas barras a bars_queue para cada símbolo en un
        formato de tupla OHLCVI: (fecha y hora, apertura, máximo, mínimo,
        cierre, volumen, interés abierto).
        """
        elevarNotImplementedError("Debería implementar update_bars()")
```

Para crear un sistema de backtesting basado en datos históricos, debemos considerar un mecanismo para importar datos a través de fuentes comunes. Hemos discutido los beneficios de una base de datos maestra de valores en capítulos anteriores. Por lo tanto, un buen candidato para crear una clase DataHandler sería combinarlo con dicha base de datos.

Sin embargo, para mayor claridad en este capítulo, quiero discutir un mecanismo más simple, el de importar archivos de variables separadas por comas (CSV) (potencialmente grandes). Esto nos permitirá centrarnos en la mecánica de la creación del DataHandler, en lugar de preocuparnos por el código "repetitivo" de conectarse a una base de datos y usar consultas SQL para obtener datos.

Por lo tanto, vamos a definir la subclase HistoricCSVDataHandler, que está diseñada para procesar múltiples archivos CSV, uno para cada símbolo negociado, y convertirlos en un diccionario de pandas DataFrames al que se puede acceder mediante los métodos de barra mencionados anteriormente.

El controlador de datos requiere algunos parámetros, a saber, una cola de eventos en la que enviar la información de MarketEvent, la ruta absoluta de los archivos CSV y una lista de símbolos. Aquí está la inicialización de la clase:

```
# datos.py

clase HistoricCSVDataHandler(DataHandler):
    """
    HistoricCSVDataHandler está diseñado para leer archivos CSV para cada
    símbolo solicitado del disco y proporcionar una interfaz para obtener la
    barra "más reciente" de manera idéntica a una interfaz comercial en vivo.

    """

    definitivamente__init__(self, eventos, csv_dir, lista_símbolos):
        """
        Inicializa el controlador de datos históricos solicitando la
        ubicación de los archivos CSV y una lista de símbolos.

        Se supondrá que todos los archivos tienen el formato
        'símbolo.csv', donde símbolo es una cadena en la lista.

        Parámetros:
        eventos: la cola de eventos.
        csv_dir: ruta de directorio absoluta a los archivos CSV.
        symbol_list: una lista de cadenas de símbolos.
        """
        self.events = eventos
        self.csv_dir = csv_dir
        self.símbolo_lista = símbolo_lista

        self.simbolo_datos = {}
        self.latest_symbol_data = {}
        self.continue_backtest = Verdadero

        self._open_convert_csv_files()
```

El controlador buscará archivos en el directorio absoluto csv_dir e intentará abrirlos con el formato "SÍMBOLO.csv", donde SÍMBOLO es el símbolo de cotización (como GOOG o AAPL). El formato de los archivos coincide con el proporcionado por Yahoo Finance, pero se modifica fácilmente para manejar formatos de datos adicionales, como los proporcionados por Quandl o DTN IQFeed. La apertura de los archivos es manejada por el método _open_convert_csv_files a continuación.

Uno de los beneficios de usar pandas como almacén de datos internamente dentro de HistoricCSVData-Handler es que los índices de todos los símbolos que se rastrean se pueden fusionar. Esto permite que los puntos de datos faltantes se rellenen hacia adelante, hacia atrás o interpolados dentro de estos espacios, de modo que los tickers se puedan comparar barra a barra. Esto es necesario para las estrategias de reversión de la media, por ejemplo. Observe el uso de los métodos de unión y reindexación al combinar los índices de todos los símbolos:

```
# datos.py

    definitivamente_open_convert_csv_files(auto):
        """
        Abre los archivos CSV del directorio de datos, convirtiéndolos en
        pandas DataFrames dentro de un diccionario de símbolos.

        Para este controlador se supondrá que los datos se toman de
        Yahoo. Así se respetará su formato. """

        comb_index = Ninguno
        porsenself.symbol_list:
            # Cargue el archivo CSV sin información de encabezado, indexado en la fecha
            self.symbol_data[s] = pd.io.parsers.read_csv(
                os.path.join(self.csv_dir, '%s.csv' % s), header=0,
                index_col=0, parse_dates=True, names=[

                    'datetime', 'open', 'high', 'low', 'close', 'volume',
                    'adj_close'
                ]
            ) . clasificar()

            # Combina el índice para rellenar los valores de avance
            sipeine_índiceesNinguna:
                comb_index = self.symbol_data[s].index más:

                comb_index.union(self.symbol_data[s].index)

            # Establece el último symbol_data en Ninguno
            self.latest_symbol_data[s] = []

        # Reindexar los marcos de datos
        porsenself.symbol_list:
            self.symbol_data[s] = self.symbol_data[s].\
                reindex(index=comb_index, method='pad').iterrows()
```

El método _get_new_bar crea un generador para proporcionar una nueva barra. Esto significa que las llamadas subsiguientes al método *rendir* una nueva barra hasta que se alcance el final de los datos del símbolo:

```
# datos.py

    definitivamente_get_new_bar(self, símbolo):
        """
        Devuelve la última barra de la fuente de datos. """

        porbenself.symbol_data[símbolo]:
```

```
                rendimiento b
```

Los primeros métodos abstractos de DataHandler que se implementarán son get_latest_bar y get_latest_bars. Estos métodos simplemente proporcionan una barra o una lista de los últimos *norte* barras de la estructura last_symbol_data:

```
# datos.py

    definitivamente get_latest_bar(uno mismo, símbolo):
        """
        Devuelve la última barra de la lista Latest_symbol. """

        probar:
            bars_list = self.latest_symbol_data[símbolo] excepto Error
        de clave:
            impresión("Ese símbolo no está disponible en el conjunto de datos históricos"). elevar

        más:
            devolver bars_list[-1]

    definitivamente get_latest_bars(self, símbolo, N=1):
        """
        Devuelve las últimas N barras de la lista Latest_symbol, o Nk si
        hay menos disponibles.
        """
        probar:
            bars_list = self.latest_symbol_data[símbolo] excepto Error
        de clave:
            impresión("Ese símbolo no está disponible en el conjunto de datos históricos"). elevar

        más:
            devolver lista_barras[-N:]
```

El siguiente método, get_latest_bar_datetime, consulta la última barra en busca de un objeto de fecha y hora que represente el "último precio de mercado":

```
    definitivamente get_latest_bar_datetime(yo, símbolo):
        """
        Devuelve un objeto de fecha y hora de Python para la última barra. """

        probar:
            bars_list = self.latest_symbol_data[símbolo] excepto Error
        de clave:
            impresión("Ese símbolo no está disponible en el conjunto de datos históricos"). elevar

        más:
            devolver bars_list[-1][0]
```

Los siguientes dos métodos que se implementarán son get_latest_bar_value y get_latest_bar_values. Ambos métodos utilizan la función getattr de Python, que consulta un objeto para ver si existe un atributo particular en un objeto. Por lo tanto, podemos pasar una cadena como "abrir" o "cerrar" a getattr y obtener el valor directamente de la barra, lo que hace que el método sea más flexible. Esto evita que tengamos que escribir métodos del tipo get_latest_bar_close, por ejemplo:

```
    definitivamente get_latest_bar_value(self, símbolo, val_type):
        """
        Devuelve uno de los valores Open, High, Low, Close, Volume u OI del
        objeto de la serie Pandas Bar.
        """
```

```
            probar:
                bars_list = self.latest_symbol_data[símbolo] excepto Error
            de clave:
                impresión("Ese símbolo no está disponible en el conjunto de datos históricos"). elevar

            más:
                devolver getattr(bars_list[-1][1], val_type)

    definitivamente get_latest_bars_values(self, símbolo, val_type, N=1):
        """
        Devuelve los últimos N valores de barra de la lista
        latest_symbol, o Nk si hay menos disponibles. """

            probar:
                bars_list = self.get_latest_bars(símbolo, N) excepto Error
            de clave:
                impresión("Ese símbolo no está disponible en el conjunto de datos históricos"). elevar

            más:
                devolver np.array([getattr(b[1], val_type) por b en bars_list])
```

El método final, update_bars, es el segundo método abstracto de DataHandler. Simplemente genera un MarketEvent que se agrega a la cola a medida que agrega las últimas barras al diccionario latest_symbol_data:

```
# datos.py

    definitivamente actualizar_barras(auto):
        """
        Empuja la barra más reciente a la estructura last_symbol_data para
        todos los símbolos en la lista de símbolos.
        """
        por s en self.symbol_list:
            probar:
                barra = siguiente(self._get_new_bar(s))
            excepto Detener iteración:
                self.continue_backtest=Falso más:

                si bar no es Ninguna:
                    self.latest_symbol_data[s].append(barra)
        self.events.put(MarketEvent())
```

Por lo tanto, tenemos un objeto derivado de DataHandler, que utilizan los componentes restantes para realizar un seguimiento de los datos del mercado. Los objetos Strategy, Portfolio y ExecutionHandler requieren los datos de mercado actuales, por lo que tiene sentido centralizarlos para evitar la duplicación de almacenamiento entre estas clases.

14.2.3 Estrategia

AEstrategiaobjeto encapsula todos los cálculos sobre los datos de mercado que generan *consultivo* señales a un portafolioobjeto. Por lo tanto, toda la "lógica de la estrategia" reside dentro de esta clase. He optado por separar los objetos Estrategia y Cartera para este backtester, ya que creo que esto es más adecuado para la situación de múltiples estrategias que alimentan "ideas" a una Cartera más grande, que luego puede manejar su propio riesgo (como la asignación sectorial, aprovechar). En el comercio de mayor frecuencia, los conceptos de estrategia y cartera estarán estrechamente acoplados y serán extremadamente dependientes del hardware. Sin embargo, ¡esto está más allá del alcance de este capítulo!

En esta etapa del desarrollo del backtester basado en eventos, no existe el concepto de un *indicador* o *fi filtro*, como los que se encuentran en el comercio técnico. Estos también son buenos candidatos para crear

una jerarquía de clases, pero están más allá del alcance de este capítulo. Por lo tanto, dichos mecanismos se utilizarán directamente en los objetos de estrategia derivados.

La jerarquía de la estrategia es relativamente simple ya que consiste en una clase base abstracta con un solo método virtual puro para generarSeñalEventoobjetos. Para crear elEstrategia jerarquía es necesario importar NumPy, pandas, elColaobjeto (que se ha convertidocola en Python 3), herramientas de clase base abstracta y elEvento de señal:

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# estrategia.py

de__futuro__importarimprimir_funcion

dea B CimportarABCMeta, método abstracto
importarfecha y hora
probar:
    importarCola como cola
exceptoError de importación:
    importarcola

importarnumpy como np
importarpandas como pd

deeventoimportarSeñalEvento
```

losEstrategiala clase base abstracta simplemente define un virtual purocalcular_señalesmétodo. En clases derivadas esto se usa para manejar la generación deSeñalEventoobjetos basados en actualizaciones de datos de mercado:

```
# estrategia.py

claseEstrategia (objeto):
    """
    La estrategia es una clase base abstracta que proporciona una interfaz para todos
    los objetos de manejo de estrategias posteriores (heredados).

    El objetivo de un objeto de estrategia (derivado) es generar objetos de señal
    para símbolos particulares en función de las entradas de barras (OHLCV)
    generadas por un objeto DataHandler.

    Esto está diseñado para funcionar con datos históricos y en vivo, ya que el
    objeto de estrategia es independiente del origen de los datos, ya que
    obtiene las tuplas de barra de un objeto de cola.
    """

    __metaclase__ = ABCMeta

    @metodoabstracto
    definitivamentecalcular_señales(auto):
        """
        Proporciona los mecanismos para calcular la lista de señales. """

        elevarNotImplementedError("Debería implementar las señales de cálculo()")
```

14.2.4 Cartera

Esta sección describe unportafolioobjeto que realiza un seguimiento de las posiciones dentro de una cartera y genera órdenes de una cantidad fija de acciones en función de las señales. Los objetos de cartera más sofisticados podrían incluir herramientas de gestión de riesgos y dimensionamiento de posiciones (como el criterio de Kelly). De hecho, en los siguientes capítulos agregaremos tales herramientas a algunas de nuestras estrategias comerciales para ver cómo se comparan con un enfoque de cartera más "ingenuo".

El sistema de gestión de órdenes de cartera es posiblemente el componente más complejo de un backtester basado en eventos. Su función es realizar un seguimiento de todas las posiciones de mercado actuales, así como del valor de mercado de las posiciones (conocidas como "tenencias"). Esto es simplemente una estimación del valor de liquidación de la posición y se deriva en parte de la facilidad de manejo de datos del backtester.

Además de la gestión de posiciones y participaciones, la cartera también debe tener en cuenta los factores de riesgo y las técnicas de dimensionamiento de posiciones para optimizar las órdenes que se envían a una casa de bolsa u otra forma de acceso al mercado.

Desafortunadamente, los sistemas de gestión de carteras y pedidos (OMS) pueden volverse bastante complejos. Por lo tanto, he tomado la decisión de mantener el objeto Portafolio relativamente sencillo, para que pueda comprender las ideas clave y cómo se implementan. La naturaleza de un diseño orientado a objetos es que permite, de forma natural, la extensión a situaciones más complejas posteriormente.

Continuando en la línea delEventojerarquía de clases aportafolioel objeto debe ser capaz de manejar SeñalEventoobjetos, generarEvento de pedidoobjetos e interpretarLlenarEventoobjetos para actualizar posiciones. Por lo tanto, no es de extrañar que elportafoliolos objetos son a menudo el componente más grande de los sistemas controlados por eventos, en términos de líneas de código (LOC).

Creamos un nuevo archivoportafolio.pye importar las bibliotecas necesarias. Estos son los mismos que la mayoría de las otras implementaciones de clases, con la excepción de que Portfolio NO será una clase base abstracta. En su lugar, será una clase base normal. Esto significa que se puede crear una instancia y, por lo tanto, es útil como un objeto de cartera de "primer paso" cuando se prueban nuevas estrategias. Se pueden derivar otros Portafolios de él y anular secciones para agregar más complejidad.

Para completar, aquí está el archivo performance.py:

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# rendimiento.py

de__futuro__importarimprimir_funcion

importarnumpy como np
importarpandas como pd


definitivamentecreate_sharpe_ratio(devoluciones, periodos=252):
    """
    Cree el índice de Sharpe para la estrategia, basado en un punto de
    referencia de cero (es decir, sin información de tasa libre de riesgo).

    Parámetros:
    devoluciones: una serie de pandas que representa las devoluciones porcentuales del
    período. períodos: diario (252), por hora (252*6,5), por minuto (252*6,5*60), etc. """

    devolvernp.sqrt(períodos) * (np.mean(devoluciones)) / np.std(devoluciones)


definitivamentecrear_drawdowns(pnl):
    """
    Calcule la mayor reducción de pico a valle de la curva PnL, así como la duración
    de la reducción. Requiere que pnl_returns sea una serie pandas.
```

```
    Parámetros:
    pnl: una serie de pandas que representa los rendimientos porcentuales del período.

    Devoluciones:
    reducción, duración: reducción y duración más altas de pico a valle. """


    # Calcular la curva de rentabilidad acumulada
    # y configurar la marca de agua alta
    hwm = [0]

    # Crear la serie de reducción y duración idx =
    pnl.índice
    reducción = pd.Series(índice = idx) duración
    = pd.Series(índice = idx)

    # Bucle sobre el rango de índice por
    tenrango (1, largo (idx)):
        hwm.append(max(hwm[t-1], pnl[t]))
        reducción[t]= (hwm[t]-pnl[t])
        duración[t]= (0sireducción[t] ==  0másduración[t-1]+1) devolverreducción,
    reducción.max(), duración.max()
```

Aquí está la lista de importación para el archivo Portfolio.py. Necesitamos importar elpisofunción de laMatemáticas biblioteca para generar tamaños de pedido con valores enteros. También necesitamos elLlenarEventoy Evento de pedidoobjetos desde elportafoliomaneja ambos. Tenga en cuenta también que estamos agregando dos funciones adicionales, create_sharpe_ratio y create_drawdowns, ambas del archivo performance.py descrito anteriormente.

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# portafolio.py

de__futuro__importarimprimir_funcion

importarfecha y hora
deMatemáticasimportarpiso
probar:
    importarCola como cola
exceptoError de importación:
    importarcola

importarnumpy como np
importarpandas como pd

deeventoimportarLlenarEvento, OrdenEvento
deactuaciónimportarcreate_sharpe_ratio, create_drawdowns
```

La inicialización del objeto Portafolio requiere acceso a las barras DataHandler, la Cola de eventos de eventos, un sello de fecha y hora de inicio y un valor de capital inicial (predeterminado en 100,000 USD).

La Cartera está diseñada para manejar el tamaño de la posición y las tenencias actuales, pero ejecutará las órdenes comerciales de manera "tonta" simplemente enviándolas directamente a la correduría con un tamaño de cantidad fija predeterminado, independientemente del efectivo que se tenga. Todas estas son suposiciones poco realistas, pero ayudan a delinear cómo funciona un sistema de gestión de pedidos (OMS) de cartera de una manera impulsada por eventos.

El portafolios contiene latodas las posicionesyposiciones_actualesmiembros El primero almacena una lista de todos los anteriores*posiciones*registrado en la marca de tiempo de un evento de datos de mercado. Una posición es simplemente la cantidad del activo mantenido. Las posiciones negativas significan que el activo se ha puesto en corto. Las últimas tiendas del diccionario current_positions contienen las posiciones actuales para la última actualización de la barra de mercado, para cada símbolo.

Además de los datos de posiciones, la cartera almacena*valores en cartera*, que describen el mercado actual*valor* de los cargos ocupados. "Valor de mercado actual" en este caso significa el precio de cierre obtenido de la barra de mercado actual, que es claramente una aproximación, pero es lo suficientemente razonable por el momento. todas_las_posesionesalmacena la lista histórica de todas las tenencias de símbolos, mientras que tenencias_actualesalmacena el diccionario más actualizado de todos los valores de tenencia de símbolos:

```
# portafolio.py

clase Portafolio (objeto):
    """
    La clase Cartera maneja las posiciones y el valor de mercado de
    todos los instrumentos a una resolución de "barra", es decir, en
    segundo lugar, minuto a minuto, 5 min, 30 min, 60 min o EOD.

    El DataFrame de posiciones almacena un índice de tiempo de la
    cantidad de posiciones mantenidas.

    El marco de datos de tenencias almacena el valor en efectivo y el valor total de
    las tenencias de mercado de cada símbolo para un índice de tiempo en
    particular, así como el cambio porcentual en el total de la cartera a través de
    las barras.
    """

    definitivamente __init__(self, bars, events, start_date, initial_capital=100000.0):
        """
        Inicializa la cartera con barras y una cola de eventos. También incluye un
        índice de fecha y hora de inicio y capital inicial (USD a menos que se
        indique lo contrario).

        Parámetros:
        barras: el objeto DataHandler con datos de mercado actuales. events:
        el objeto de cola de eventos.
        start_date: la fecha de inicio (barra) de la cartera. initial_capital -
        El capital inicial en USD.
        """
        self.barras = barras
        self.events = eventos
        self.símbolo_lista = self.barras.símbolo_lista
        self.start_date = start_date
        self.capital_inicial = capital_inicial

        self.todas_posiciones = self.construir_todas_posiciones()
        self.posiciones_actuales = dict( (k,v)por k, ven\
            [(s, 0)porsenself.símbolo_lista] )

        self.todas las existencias = self.construir_todas las existencias()
        self.existencias_actuales = self.construir_existencias_actuales()
```

El siguiente método,construir_todas_las_posiciones,simplemente crea un diccionario para cada símbolo, establece el valor en cero para cada uno y luego agrega una clave de fecha y hora, y finalmente lo agrega a una lista. Utiliza una comprensión de diccionario, que es similar en espíritu a una comprensión de lista:

```
# portafolio.py
```

```
    definitivamenteconstruct_all_positions(self):
        """
        Construye la lista de posiciones utilizando start_date para
        determinar cuándo comenzará el índice de tiempo.
        """
        d = dict( (k,v)pork, venen[(s, 0)porsenself.symbol_list] ) d['datetime'] = self.start_date

        devolver[d]
```

losconstruct_all_holdingsEl método es similar al anterior, pero agrega claves adicionales para efectivo, comisión y total, que representan respectivamente el efectivo sobrante en la cuenta después de cualquier compra, la comisión acumulada acumulada y el patrimonio total de la cuenta, incluido el efectivo y cualquier posición abierta. Las posiciones cortas se tratan como negativas. El efectivo inicial y el capital total de la cuenta se establecen en el capital inicial.

De esta manera, hay "cuentas" separadas para cada símbolo, el "efectivo disponible", la "comisión" pagada (tarifas de Interactive Broker) y un valor de cartera "total". Claramente, esto no tiene en cuenta los requisitos de margen o las restricciones de venta en corto, pero es suficiente para darle una idea de cómo se crea dicho OMS:

```
# portafolio.py

    definitivamenteconstruct_all_holdings(self):
        """
        Construye la lista de existencias utilizando start_date para
        determinar cuándo comenzará el índice de tiempo.
        """
        d = dict( (k,v)pork, venen[(s, 0.0)porsenself.symbol_list] ) d['datetime'] = self.start_date

        d['efectivo'] = self.initial_capital
        d['comisión'] = 0.0
        d['total'] = self.capital_inicial devolver[d]
```

El siguiente método,construct_current_holdingses casi idéntico al método anterior, excepto que no envuelve el diccionario en una lista, porque solo crea una sola entrada:

```
# portafolio.py

    definitivamenteconstruct_current_holdings(self):
        """
        Esto construye el diccionario que contendrá el valor instantáneo de la cartera en
        todos los símbolos.
        """
        d = dict( (k,v)pork, venen[(s, 0.0)porsenself.símbolo_lista] ) d['efectivo'] = self.initial_capital

        d['comisión'] = 0.0
        d['total'] = self.capital_inicial devolverd
```

En cada*latido del corazón*, es decir, cada vez que se solicitan nuevos datos de mercado almanejador de datos objeto, la cartera deberá actualizar el valor de mercado actual de todas las posiciones mantenidas. En un escenario de negociación en vivo, esta información se puede descargar y analizar directamente desde el corredor, pero para una implementación de backtesting es necesario calcular estos valores manualmente desde las barras DataHandler.

Desafortunadamente, no existe el "valor de mercado actual" debido a los diferenciales de oferta y demanda y los problemas de liquidez. Por lo tanto, es necesario estimarlo multiplicando la cantidad del activo poseído por un "precio" aproximado particular. El enfoque que he tomado aquí es usar el cierre

precio de la última barra recibida. Para una estrategia intradía esto es relativamente realista. Para una estrategia diaria, esto es menos realista ya que el precio de apertura puede diferir sustancialmente del precio de cierre.

El métodoupdate_timeindexse encarga del seguimiento de las nuevas existencias. En primer lugar, obtiene los precios más recientes del controlador de datos de mercado y crea un nuevo diccionario de símbolos para representar las posiciones actuales, estableciendo las posiciones "nuevas" iguales a las posiciones "actuales".

Las posiciones actuales sólo se modifican cuando unLlenarEventose obtiene, el cual se maneja posteriormente en el código de cartera. Luego, el método agrega este conjunto de posiciones actuales al todas las posicioneslista.

Luego, las posiciones se actualizan de manera similar, con la excepción de que el valor de mercado se vuelve a calcular multiplicando las posiciones actuales con el precio de cierre de la última barra. Finalmente, las nuevas participaciones se añaden atodas las posesiones:

```
# portafolio.py

    definitivamenteupdate_timeindex(auto, evento):
        """
        Agrega un nuevo registro a la matriz de posiciones para la barra de datos de
        mercado actual. Esto refleja la barra ANTERIOR, es decir, se conocen todos
        los datos de mercado actuales en esta etapa (OHLCV).

        Hace uso de un MarketEvent de la cola de eventos. """

        last_datetime = self.bars.get_latest_bar_datetime(
            self.símbolo_lista[0]
        )

        # Actualizar posiciones
        # ================
        dp = dict( (k,v)pork, ven[(s, 0)porsenself.symbol_list] ) dp['datetime'] = last_datetime


        porsenself.symbol_list:
            dp[s] = self.posiciones_actuales[s]

        # Agregar las posiciones actuales
        self.todas_posiciones.append(dp)

        # Actualizar existencias
        # ===============
        dh = dictado( (k,v)pork, ven[(s, 0)porsenself.symbol_list] ) dh['fechahora'] =
        última_fechahora
        dh['efectivo'] = self.tenencias_actuales['efectivo'] dh['comisión'] =
        self.tenencias_actuales['comisión'] dh['total'] =
        self.tenencias_actuales['efectivo']

        porsenself.symbol_list:
            # Aproximación al valor real valor_mercado =
            self.posiciones_actuales[s] * \
                self.bars.get_latest_bar_value(s, "adj_close") dh[s] =
                valor_mercado
            dh['total'] += valor_de_mercado

        # Agregar las existencias actuales
        self.all_holdings.append(dh)
```

El métodoactualizar_posiciones_desde_llenardetermina si unLlenarEventoes una compra o a Sell y luego actualiza elposiciones_actualesdiccionario en consecuencia sumando / restando

la cantidad correcta de acciones:

```
# portafolio.py

    definitivamenteupdate_positions_from_fill(self, fill):
        """
        Toma un objeto de relleno y actualiza la matriz de posición para
        reflejar la nueva posición.

        Parámetros:
        fill: el objeto de relleno con el que actualizar las posiciones. """

        # Comprobar si el relleno es una compra o venta
        llenar_dir = 0
        sidirección.llenar == 'COMPRAR':
            llenar_dir = 1
        sidirección.llenar == 'VENDER':
            llenar_dir = -1

        # Actualizar la lista de posiciones con nuevas cantidades
        self.current_positions[fill.symbol] += fill_dir*fill.quantity
```

El correspondienteactualizar_existencias_desde_llenares similar al método anterior pero actualiza el *valores en cartera*valores en su lugar. Para simular el costo de un llenado, el siguiente método no utiliza el costo asociado de laLlenarEvento.¿Por qué es esto? En pocas palabras, en un entorno de backtesting, el costo de llenado es realmente desconocido (el*impacto en el mercado*y el*profundidad del libro*son desconocidos) y por lo tanto se debe estimar.

Por lo tanto, el costo de llenado se establece en el "precio de mercado actual", que es el precio de cierre de la última barra. Las existencias de un símbolo en particular se establecen entonces para que sean iguales al costo de llenado multiplicado por la cantidad negociada. Para la mayoría de las estrategias comerciales de frecuencia más baja en mercados líquidos, esta es una aproximación razonable, pero con una frecuencia alta, estos problemas deberán considerarse en una prueba retrospectiva de producción y un motor comercial en vivo.

Una vez que se conoce el costo de llenado, se pueden actualizar las existencias actuales, el efectivo y los valores totales. La comisión acumulada también se actualiza:

```
# portafolio.py

    definitivamenteupdate_holdings_from_fill(self, fill):
        """
        Toma un objeto de relleno y actualiza la matriz de existencias para
        reflejar el valor de las existencias.

        Parámetros:
        fill: el objeto de relleno con el que actualizar las existencias. """

        # Comprobar si el relleno es una compra o venta
        llenar_dir = 0
        sidirección.llenar == 'COMPRAR':
            llenar_dir = 1
        sidirección.llenar == 'VENDER':
            llenar_dir = -1

        # Actualizar la lista de existencias con nuevas cantidades
        costo_relleno = self.bars.get_latest_bar_value(relleno.símbolo, "adj_close") costo =
        dir_relleno * costo_relleno * relleno.cantidad
        self.tenencias_actuales[llenar.símbolo] += costo
        self.tenencias_actuales['comisión'] += llenar.comisión
        self.tenencias_actuales['efectivo'] -= (costo + llenar.comisión)
```

```
            self.current_holdings['total'] -= (costo + llenar.comisión)
```

La virtualidad puraactualizar_llenarmétodo de laportafolioLa clase se implementa aquí. Es simply ejecuta los dos métodos anteriores,actualizar_posiciones_desde_llenaryactualizar_existencias_desde_llenar, al recibir un evento de llenado:

```
# portafolio.py

    definitivamenteupdate_fill(auto, evento):
        """
        Actualiza las posiciones y participaciones actuales de la cartera
        desde un FillEvent.
        """
        sievent.type == 'LLENAR':
            self.update_positions_from_fill(evento)
            self.update_holdings_from_fill(evento)
```

Mientras que laportafolioel objeto debe manejarLlenarEventos, también debe cuidar de generar Evento de pedidos al recibir uno o másSeñalEventos.

losgenerar_pedido_ingenuoEl método simplemente toma una señal para ir en largo o en corto en un activo, enviando una orden para hacerlo por 100 acciones de dicho activo. Claramente, 100 es un valor arbitrario y claramente dependerá del capital total de la cartera en una simulación de producción.

En una implementación realista, este valor estará determinado por una superposición de gestión de riesgos o dimensionamiento de posiciones. Sin embargo, esto es un simplismoportafoliopor lo que "ingenuamente" envía todas las órdenes directamente desde las señales, sin un sistema de riesgo.

El método maneja el anhelo, el acortamiento y la salida de una posición, según la cantidad actual y el símbolo particular. CorrespondienteEvento de pedidoluego se generan los objetos:

```
# portafolio.py

    definitivamentegenerar_pedido_ingenuo(uno mismo, señal):
        """
        Simplemente archiva un objeto de pedido como un tamaño de
        cantidad constante del objeto de señal, sin consideraciones de
        gestión de riesgos ni de tamaño de posición.

        Parámetros:
        señal: la tupla que contiene información de la señal. """

        orden = Ninguno

        símbolo = señal.símbolo
        dirección = señal.tipo_señal fuerza =
        señal.fuerza

        mkt_quantity = 100
        cur_quantity = self.current_positions[símbolo] order_type =
        'MKT'

        sidirección == 'LARGO'ycantidad_actual == 0:
            order = OrderEvent(símbolo, order_type, mkt_quantity, 'COMPRAR') sidirección
        == 'CORTA'ycantidad_actual == 0:
            order = OrderEvent(símbolo, order_type, mkt_quantity, 'SELL')

        sidirección == 'SALIR'ycantidad_actual > 0:
            order = OrderEvent(símbolo, order_type, abs(cur_quantity), 'SELL') sidirección == 'SALIR'
        ycantidad_actual < 0:
            order = OrderEvent(símbolo, order_type, abs(cur_quantity), 'COMPRAR')
```

```
        devolverordenar
```

losactualizar_señalEl método simplemente llama al método anterior y agrega el orden generado a la cola de eventos:

```
# portafolio.py

    definitivamenteupdate_signal(auto, evento):
        """
        Actúa sobre un SignalEvent para generar nuevas órdenes
        basadas en la lógica de la cartera.
        """
        sievent.type == 'SEÑAL':
            order_event = self.generate_naive_order(evento)
            self.events.put(order_event)
```

El penúltimo método en elportafolioes la generación de una curva de equidad. Esto simplemente crea un flujo de retornos, útil para los cálculos de rendimiento, y luego normaliza la curva de capital para que se base en porcentajes. Por lo tanto, el tamaño inicial de la cuenta es igual a 1,0, a diferencia del monto absoluto en dólares:

```
# portafolio.py

    definitivamentecreate_equity_curve_dataframe(auto):
        """
        Crea un DataFrame de pandas a partir de la lista de
        diccionarios all_holdings.
        """
        curve = pd.DataFrame(self.all_holdings) curve.set_index('datetime',
        inplace=True) curve['returns'] = curve['total'].pct_change()
        curve['equity_curve'] = (1.0+curve ['devoluciones']).cumprod()
        self.equity_curve = curva
```

El último método en elportafolioes el resultado de la curva de equidad y varias estadísticas de rendimiento relacionadas con la estrategia. La línea final genera un archivo, equity.csv, en el mismo directorio que el código, que se puede cargar en un script Python de Matplotlib (o una hoja de cálculo como MS Excel o LibreOffice Calc) para su posterior análisis.

Tenga en cuenta que la Duración de la reducción se da en términos del número absoluto de "barras" durante las que se prolongó la reducción, en lugar de un período de tiempo particular.

```
    definitivamenteoutput_summary_stats(auto):
        """
        Crea una lista de estadísticas de resumen para la cartera. """

        retorno_total = self.equity_curve['equity_curve'][-1] devoluciones =
        self.equity_curve['returns']
        pnl = self.equity_curve['equity_curve']

        sharpe_ratio = create_sharpe_ratio(rendimientos, períodos=252*60*6,5) drawdown,
        max_dd, dd_duration = create_drawdowns(pnl) self.equity_curve['drawdown'] =
        drawdown

        estadísticas = [("Retorno total", "%0.2f%%" % \
                         ((total_return - 1.0) * 100.0)), ("Sharpe Ratio", "%0.2f" %
                    sharpe_ratio), ("Max Drawdown", "%0.2f%%" % (max_dd *
                    100.0)), ("Drawdown Duración", "%d" % dd_duración)]
```

```
        self.equity_curve.to_csv('equity.csv') devolver
        estadísticas
```

losportafolioEl objeto es el aspecto más complejo de todo el sistema de backtest basado en eventos. La implementación aquí, aunque compleja, es relativamente elemental en el manejo de posiciones.

14.2.5 Controlador de ejecución

En esta sección, estudiaremos la ejecución de órdenes comerciales mediante la creación de una jerarquía de clases que representará un mecanismo de manejo de órdenes simulado y, en última instancia, se vinculará con una correduría u otro medio de conectividad de mercado.

losManejador de ejecucióndescrito aquí es extremadamente simple, ya que ejecuta todas las órdenes al precio de mercado actual. Esto es muy poco realista, pero sirve como una buena línea de base para mejorar.

Al igual que con las jerarquías de clases base abstractas anteriores, debemos importar las propiedades y los decoradores necesarios desde ela B Cbiblioteca. Además necesitamos importar elLlenarEventoy Evento de pedido:

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# ejecución.py

de__futuro__importarimprimir_funcion

dea B CimportarABCMeta, método abstracto
importarfecha y hora
probar:
    importarCola como cola
exceptoError de importación:
    importarcola

deeventoimportarLlenarEvento, OrdenEvento
```

losManejador de ejecuciónes similar a las clases base abstractas anteriores y simplemente tiene un método virtual puro,ejecutar_orden:

```
# ejecución.py

claseExecutionHandler(objeto):
    """
    La clase abstracta ExecutionHandler maneja la interacción entre un
    conjunto de objetos de orden generados por un Portafolio y el conjunto
    final de objetos de Relleno que realmente ocurren en el mercado.


    Los controladores se pueden utilizar para crear subclases de corretaje simulado
    o corretaje en vivo, con interfaces idénticas. Esto permite que las estrategias se
    prueben retrospectivamente de una manera muy similar al motor de
    negociación en vivo.
    """

    __metaclase__ = ABCMeta

    @metodoabstracto
    definitivamenteejecutar_orden(uno mismo, evento):
        """
        Toma un evento de pedido y lo ejecuta, produciendo un evento de
        relleno que se coloca en la cola de eventos.
```

```
        Parámetros:
        evento: contiene un objeto de evento con información de pedido. """

        elevarNotImplementedError("Debería implementar execute_order()")
```

Para realizar una prueba retrospectiva de las estrategias, necesitamos simular cómo se realizará una operación. La implementación más simple posible es asumir que todas las órdenes se completan al precio de mercado actual para todas las cantidades. Esto es claramente extremadamente poco realista y una gran parte de la mejora del realismo del backtest vendrá del diseño de modelos más sofisticados de deslizamiento e impacto en el mercado.

Tenga en cuenta que elLlenarEventose le da un valor deNingunaPara elcoste_relleno (ver la penúltima línea enejecutar_orden)como ya nos hemos ocupado del costo de completar elportafolioobjeto descrito anteriormente. En una implementación más realista, haríamos uso del valor de datos de mercado "actual" para obtener un costo de llenado realista.

Simplemente he utilizado ARCA como intercambio, aunque para propósitos de backtesting esto es puramente un marcador de posición de cadena. En un entorno de ejecución en vivo, esta dependencia del lugar sería mucho más importante:

```
# ejecución.py

claseControlador de ejecución simulado (controlador de ejecución):
    """
    El controlador de ejecución simulado simplemente convierte todos los
    objetos de pedido en sus objetos de relleno equivalentes automáticamente
    sin problemas de latencia, deslizamiento o relación de relleno.

    Esto permite una prueba sencilla de "primer intento" de cualquier estrategia,
    antes de la implementación con un controlador de ejecución más sofisticado.

    """

    definitivamente__init__(uno mismo, eventos):
        """
        Inicializa el controlador, configurando las colas de eventos
        internamente.

        Parámetros:
        events: la cola de objetos de eventos. """

        self.events = eventos

    definitivamenteejecutar_orden(uno mismo, evento):
        """
        Simplemente convierte objetos de pedido en objetos de relleno de forma ingenua,
        es decir, sin problemas de latencia, deslizamiento o relación de relleno.

        Parámetros:
        evento: contiene un objeto de evento con información de pedido. """

        sievent.type == 'PEDIDO':
            evento_relleno = EventoRelleno(
                datetime.datetime.utcnow(), evento.símbolo, 'ARCA',
                evento.cantidad, evento.dirección, Ninguno
            )
            self.events.put(fill_event)
```

14.2.6 Prueba inversa

Ahora estamos en condiciones de crear la jerarquía de clases de Backtest. El objeto Backtest encapsula la lógica de manejo de eventos y esencialmente une todas las demás clases que hemos discutido anteriormente.

El objeto Backtest está diseñado para llevar a cabo un sistema anidado controlado por eventos de bucle while para manejar los eventos colocados en el objeto Cola de eventos. El ciclo while externo se conoce como "bucle de latido" y decide la resolución temporal del sistema de backtesting. En un entorno en vivo, este valor será un número positivo, como 600 segundos (cada diez minutos). Por lo tanto, los datos de mercado y las posiciones solo se actualizarán en este período de tiempo.

Para el backtester descrito aquí, el "latido del corazón" se puede establecer en cero, independientemente de la frecuencia de la estrategia, ya que los datos ya están disponibles en virtud del hecho de que son históricos.

Podemos ejecutar el backtest a la velocidad que queramos, ya que el sistema basado en eventos es independiente de*cuando*los datos estuvieron disponibles, siempre que tengan una marca de tiempo asociada. Por lo tanto, solo lo he incluido para demostrar cómo funcionaría un motor comercial en vivo. Por lo tanto, el ciclo externo finaliza una vez que DataHandler le informa al objeto Backtest, mediante el uso de un atributo booleano continue_backtest.

El ciclo while interno en realidad procesa las señales y las envía al componente correcto según el tipo de evento. Por lo tanto, la cola de eventos se llena y se vacía continuamente con eventos. Esto es lo que significa que un sistema sea*evento conducido*.

La primera tarea es importar las bibliotecas necesarias. Importamos pprint ("pretty-print"), porque queremos mostrar las estadísticas de una manera amigable con la salida:

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# backtest.py

de__futuro__importarimprimir_funcion

importarfecha y hora
importarpprint
probar:
    importarCola como cola
excepto Error de importación:
    importarcola
importartiempo
```

La inicialización del objeto Backtest requiere el directorio CSV, la lista completa de símbolos negociados, el capital inicial, el tiempo de latido en milisegundos, el sello de fecha y hora de inicio del backtest, así como los objetos DataHandler, ExecutionHandler, Portfolio y Strategy. Se utiliza una cola para celebrar los eventos. Las señales, órdenes y rellenos se cuentan:

```
# backtest.py

claseBacktest (objeto):
    """
    Encapsula la configuración y los componentes para llevar a cabo un
    backtest basado en eventos.
    """

    definitivamente__en eso__(
        yo, csv_dir, lista_símbolos, capital_inicial, latido,
        fecha_inicio, manejador_datos, manejador_ejecución,
        cartera, estrategia
    ) :
        """
        Inicializa el backtest.
```

```
        Parámetros:
        csv_dir: la raíz dura del directorio de datos CSV. symbol_list: la lista de
        cadenas de símbolos. initial_capital: el capital inicial de la cartera.
        latido del corazón - Backtest "latido del corazón" en segundos

        start_date: la fecha y hora de inicio de la estrategia. data_handler - (Clase) Maneja
        la fuente de datos de mercado. execution_handler - (Clase) Maneja las órdenes/
        rellenos para operaciones. cartera - (Clase) Realiza un seguimiento de la cartera
        actual
            y puestos anteriores.
        estrategia - (Clase) Genera señales basadas en datos de mercado. """

        self.csv_dir = csv_dir
        self.símbolo_lista = lista_símbolos self.initial_capital
        = initial_capital self.heartbeat = latido

        self.start_date = start_date

        self.data_handler_cls = data_handler
        self.execution_handler_cls = ejecucion_handler
        self.portfolio_cls = portafolio
        self.strategy_cls = estrategia

        self.eventos = cola.Cola()

        auto.señales = 0
        auto.pedidos = 0
        self.fills = 0
        self.num_strats = 1

        self._generar_instancias_comerciales()
```

El primer método, _generate_trading_instances, adjunta todos los objetos comerciales (DataHandler, Strategy, Portfolio y ExecutionHandler) a varios miembros internos:

```
# backtest.py

    definitivamente_generar_instancias_comerciales(uno mismo):
        """
        Genera los objetos de instancia comercial a partir de
        sus tipos de clase.
        """
        impresión(
            "Creación de DataHandler, Estrategia, Portafolio y ExecutionHandler"
        )
        self.controlador_de_datos = self.controlador_de_datos_cls(self.events, self.csv_dir,
            self.símbolo_lista)
        self.strategy = self.strategy_cls(self.data_handler, self.events) self.portfolio =
        self.portfolio_cls(self.data_handler, self.events,
                                                    self.start_date,
                                                    self.capital_inicial)
        self.execution_handler = self.execution_handler_cls(self.events)
```

El método _run_backtest es donde se lleva a cabo el manejo de señales del motor Backtest. Como se describió anteriormente, hay dos bucles while, uno anidado dentro de otro. El externo realiza un seguimiento del latido del corazón del sistema, mientras que el interno verifica si hay un evento en el objeto Queue y actúa en consecuencia llamando al método apropiado en el objeto necesario.

Para un evento de mercado, se le dice al objeto de estrategia que vuelva a calcular nuevas señales, mientras que al objeto de cartera se le dice que vuelva a indexar el tiempo. Si se recibe un objeto SignalEvent, se le dice a Portfolio que maneje la nueva señal y la convierta en un conjunto de OrderEvents, si corresponde. Si se recibe un OrderEvent, se envía a ExecutionHandler la orden para que se transmita al corredor (si se encuentra en un entorno comercial real). Finalmente, si se recibe un FillEvent, el Portafolio se actualizará para estar al tanto de las nuevas posiciones:

```
# backtest.py

    definitivamente_ejecutar_backtest(auto):
        """
        Ejecuta el backtest.
        """
        yo = 0
        tiempoVerdadero:
            yo += 1
            impresióni
            # Actualizar las barras del mercado
            siself.data_handler.continue_backtest == Verdadero:
                self.controlador_de_datos.update_bars()
            más:
                descanso

            # Manejar los eventos
            tiempoVerdadero:
                probar:
                    evento = self.events.get(Falso) excepto
                    cola.Vacío:
                    descanso
                más:
                    sieventono esNinguna:
                        sievent.type == 'MERCADO':
                            self.strategy.calculate_signals(evento)
                            self.portfolio.update_timeindex(evento)

                        elifevent.type == 'SEÑAL':
                            auto.señales += 1
                            self.portfolio.update_signal(evento)

                        elifevent.type == 'PEDIDO':
                            auto.pedidos += 1
                            self.execution_handler.execute_order(evento)

                        elifevent.type == 'LLENAR':
                            self.fills += 1
                            self.portfolio.update_fill(evento)

            time.sleep(auto.latido)
```

Una vez que se completa la simulación de backtest, el rendimiento de la estrategia se puede mostrar en la terminal/consola. Se crea el DataFrame de pandas de la curva de equidad y se muestran las estadísticas de resumen, así como el recuento de Señales, Órdenes y Rellenos:

```
# backtest.py

    definitivamente_rendimiento_salida(auto):
        """
        Muestra el rendimiento de la estrategia del backtest.
```

```
        """
        self.cartera.create_equity_curve_dataframe()

        impresión("Creando estadísticas de resumen...")
        estadísticas = self.portfolio.output_summary_stats()

        impresión("Creando la curva de equidad...") impresión
        (self.portfolio.equity_curve.tail(10))
        pprint.pprint(estadísticas)

        impresión("Señales: %s" % auto.señales)
        impresión("Pedidos: %s" % auto.pedidos)
        impresión("Rellenos: %s" % self.fills)
```

El último método a implementar essimular_comercio.Simplemente llama a los dos métodos descritos anteriormente, en orden:

```
# backtest.py

    definitivamentesimular_trading(auto):
        """
        Simula el backtest y el rendimiento de la cartera de salida. """

        self._run_backtest()
        self._rendimiento_de_salida()
```

Esto concluye los objetos operativos del backtester impulsado por eventos.

14.3 Ejecución dirigida por eventos

Arriba describimos un básicoManejador de ejecuciónclase que simplemente creó una correspondienteLlenarEvento instancia para cadaEvento de pedido.Esto es precisamente lo que necesitamos para una prueba retrospectiva de "primer paso", pero cuando deseamos conectar el sistema a una agencia de corretaje, necesitamos un manejo más sofisticado. En esta sección definimos elIBExecutionHandler,una clase que nos permite hablar con la popular API de Interactive Brokers y así automatizar nuestra ejecución.

La idea esencial de laIBExecutionHandlerla clase es para recibirEvento de pedidoinstancias de la cola de eventos y luego ejecutarlas directamente contra la API de pedidos de Interactive Brokers utilizando la biblioteca IbPy de código abierto. La clase también manejará los mensajes de "Respuesta del servidor" enviados a través de la API. En esta etapa, la única acción tomada será crear los correspondientes LlenarEventoinstancias que luego serán enviadas de vuelta a la cola de eventos.

La clase en sí podría volverse bastante compleja, con una lógica de optimización de ejecución y un manejo de errores sofisticado. Sin embargo, he optado por mantenerlo relativamente simple aquí para que pueda ver las ideas principales y ampliarlas en la dirección que se adapte a su estilo comercial particular.

Como siempre, la primera tarea es crear el archivo de Python e importar las bibliotecas necesarias. el archivo se llamaib_ejecucion.pyy vive en el mismo directorio que los otros archivos controlados por eventos.

Importamos las bibliotecas de manejo de fecha/hora necesarias, los objetos IbPy y los objetos Event específicos que son manejados porIBExecutionHandler:

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# ib_ejecucion.py

de__futuro__importarimprimir_funcion

importarfecha y hora
importartiempo
```

```
deib.ext.Contratoimportar Contrato de
ib.ext.Pedidoimportar Ordenar
deib.optarimportaribConnection, mensaje

deeventoimportar LlenarEvento, OrdenEvento de
ejecuciónimportar Manejador de ejecución
```

Ahora definimos elIBExecutionHandlerclase. Los __en eso__constructor requiere en primer lugar el conocimiento de laeventoscola. También requiere la especificación deorden_enrutamiento,que he predeterminado en "SMART". Si tiene requisitos de intercambio específicos, puede especificarlos aquí. El valor por defectodivisatambién se ha fijado en dólares estadounidenses.

Dentro del método creamos unllenar_dictdiccionario, necesario más tarde para su uso en la generación LlenarEventoinstancias. También creamos untws_connobjeto de conexión para almacenar nuestra información de conexión a la API de Interactive Brokers. También tenemos que crear un valor predeterminado inicialSolicitar ID, que realiza un seguimiento de todos los pedidos posteriores para evitar duplicados. Finalmente registramos los manejadores de mensajes (que definiremos con más detalle a continuación):

```
# ib_ejecucion.py

claseIBExecutionHandler(ExecutionHandler):
    """
    Maneja la ejecución de órdenes a través de la API de Interactive
    Brokers, para usar contra cuentas cuando se negocia en vivo
    directamente.
    """

    definitivamente__en eso__(
        self, eventos, order_routing="SMART", moneda="USD"
    ) :
        """
        Inicializa la instancia de IBExecutionHandler. """

        self.events = eventos
        self.order_routing = order_routing
        self.currency = moneda
        self.fill_dict = {}

        self.tws_conn = self.create_tws_connection() self.order_id =
        self.create_initial_order_id() self.register_handlers()
```

La API de IB utiliza un sistema de eventos basado en mensajes que permite que nuestra clase responda de formas particulares a ciertos mensajes, de manera similar al propio backtester basado en eventos. No he incluido ningún manejo de errores real (por razones de brevedad), más allá de la salida a la terminal, mediante el _manejador_de_erroresmétodo.

Los _controlador_de_respuestasmétodo, por otro lado, se utiliza para determinar si unLlenarEvento es necesario crear una instancia. El método pregunta si se ha recibido un mensaje de "pedido abierto" y comprueba si una entrada en nuestrollenar_dictpara este ID de pedido en particular ya se ha establecido. Si no, entonces se crea uno.

Si ve un mensaje de "estado del pedido" y ese mensaje en particular indica que se ha completado un pedido, entonces llamacrear_llenarpara crear unLlenarEvento.También envía el mensaje a la terminal para fines de registro/depuración:

```
# ib_ejecucion.py

    definitivamente_error_handler(self, mensaje):
        """
```

```
        Maneja la captura de mensajes de error """

        # Actualmente no hay manejo de errores.
        impresión("Error del servidor: %s" % mensaje)

    definitivamente_answer_handler(auto, mensaje):
        """
        Identificadores de las respuestas del servidor
        """
        # Manejar el procesamiento de orderId de orden
        abierta simsg.typeName == "pedido abierto"y\
            msg.orderId == self.order_idy\ no
            self.fill_dict.has_key(msg.orderId):
            self.create_fill_dict_entry(msg)
        # Manejar rellenos
        simsg.typeName == "estado del pedido"y\
            msg.status == "Llenado"y\ self.fill_dict[msg.orderId]["llenado"]
            == Falso: self.create_fill(msg)

        impresión("Respuesta del servidor: %s, %s\n" % (msg.typeName, msg))
```

El siguiente método,crear_conexión_tws,crea una conexión a la API de IB usando IbPyConexión ib objeto. Utiliza un puerto predeterminado de 7496 y un ID de cliente predeterminado de 10. Una vez que se crea el objeto, elconectarSe llama al método para realizar la conexión:

```
# ib_ejecucion.py

    definitivamentecreate_tws_connection(auto):
        """
        Conéctese a Trader Workstation (TWS) que se ejecuta en el puerto
        habitual de 7496, con un ID de cliente de 10.
        Nosotros elegimos el ID de cliente y necesitaremos ID separados tanto
        para la conexión de ejecución como para la conexión de datos de
        mercado, si esta última se usa en otro lugar. """

        tws_conn = ibConnection()
        tws_conn.conectar()
        devolvertws_conn
```

Para realizar un seguimiento de los pedidos separados (a los efectos de realizar un seguimiento de los rellenos), el siguiente método create_initial_order_idse usa Lo he predeterminado a "1", pero un enfoque más sofisticado sería consultar a IB para obtener la última ID disponible y usarla. Siempre puede restablecer el ID de orden de API actual a través de Trader Workstation > Configuración global > Panel de configuración de API:

```
# ib_ejecucion.py

    definitivamentecreate_initial_order_id(auto):
        """
        Crea el ID de pedido inicial utilizado por Interactive Brokers para
        realizar un seguimiento de los pedidos enviados.
        """
        # Aquí hay margen para más lógica, pero
        # usará "1" como predeterminado por ahora.
        devolver1
```

El siguiente método,registradores_manipuladores,simplemente registra los métodos de manejo de errores y respuestas definidos anteriormente con la conexión TWS:

```
# ib_ejecucion.py
```

```
    definitivamenteregister_handlers(auto):
        """
        Registre las funciones de manejo de mensajes de
        respuesta del servidor y de error.
        """
        # Asignar la función de manejo de errores definida anteriormente
        # a la conexión TWS self.tws_conn.register(self._error_handler, 'Error')


        # Asigne todos los mensajes de respuesta del servidor al
        # función answer_handler definida anteriormente
        self.tws_conn.registerAll(self._reply_handler)
```

Para realizar una transacción real, es necesario crear un IbPyContratoinstancia y luego emparejarlo con un IbPyOrdenarinstancia, que se enviará a la API de IB. El siguiente método, crear_contrato,genera el primer componente de este par. Espera un símbolo de cotización, un tipo de valor (por ejemplo, acciones o futuros), un intercambio/intercambio principal y una moneda. devuelve elContratoinstancia:

```
# ib_ejecucion.py

    definitivamentecreate_contract(self, símbolo, sec_type, exch, prim_exch, curr):
        """
        Cree un objeto de contrato que defina lo que
        comprarse, en qué cambio y en qué moneda.

        símbolo: el símbolo de cotización del contrato
        sec_type: el tipo de valor para el contrato ('STK' es 'stock') exch: el intercambio
        para llevar a cabo el contrato en
        prim_exch - El intercambio principal para llevar a cabo el contrato en curr - La
        moneda en la que comprar el contrato """

        contrato = contrato()
        contract.m_symbol = símbolo
        contract.m_secType = sec_type
        contract.m_exchange = exch
        contract.m_primaryExch = prim_exch
        contract.m_currency = curr
        devolvercontrato
```

El siguiente método,crear orden,genera el segundo componente del par, a saber, elOrdenar instancia. Espera un tipo de orden (por ejemplo, mercado o límite), una cantidad del activo a negociar y una "acción" (compra o venta). devuelve elOrdenarinstancia:

```
# ib_ejecucion.py

    definitivamentecreate_order(self, order_type, cantidad, acción):
        """
        Cree un objeto de Orden (Mercado/Límite) para ir largo/corto.

        order_type - 'MKT', 'LMT' para cantidad de órdenes de mercado o
        límite - Número integral de activos para ordenar acción -
        'COMPRAR' o 'VENDER'
        """
        orden = Orden()
        order.m_orderType = order_type
        order.m_totalQuantity = cantidad
```

```
        order.m_action = acción
        devolverordenar
```

Para evitar la duplicaciónLlenarEventoinstancias para un ID de pedido en particular, utilizamos un diccionario llamadollenar_dictpara almacenar claves que coincidan con ID de pedidos particulares. Cuando se ha generado un relleno, la clave "llenado" de una entrada para un ID de pedido en particular se establece enVerdadero.Si se recibe un mensaje posterior de "Respuesta del servidor" de IB que indica que se completó un pedido (y es un mensaje duplicado), no dará lugar a un nuevo cumplimiento. El siguiente métodocreate_fill_dict_entry lleva a cabo esto:

```
# ib_ejecucion.py

    definitivamentecreate_fill_dict_entry(self, mensaje):
        """
        Crea una entrada en el diccionario de relleno que enumera los ID de
        pedido y proporciona información de seguridad. Esto es necesario
        para el comportamiento basado en eventos del comportamiento del
        mensaje del servidor IB.
        """
        self.fill_dict[msg.orderId] = {
            "símbolo": msg.contract.m_symbol,
            "intercambio": msg.contract.m_exchange,
            "dirección": msg.order.m_action, "llenado": Falso

        }
```

El siguiente método,crear_llenar,en realidad crea elLlenarEventoinstancia y lo coloca en la cola de eventos:

```
# ib_ejecucion.py

    definitivamentecreate_fill(auto, mensaje):
        """
        Maneja la creación del FillEvent que se colocará en la cola de
        eventos después de que se complete un pedido.

        """
        fd = self.fill_dict[msg.orderId]

        # Preparar los datos de
        relleno símbolo = fd["símbolo"]
        intercambio = fd["intercambio"]
        llenado = mensaje.llenado
        dirección = fd["dirección"] fill_cost =
        msg.avgFillPrice

        # Crear un objeto de evento de
        relleno llenar = LlenarEvento(
            datetime.datetime.utcnow(), símbolo, intercambio,
            relleno, dirección, coste_relleno
        )

        # Asegúrese de que varios mensajes no creen
        # rellenos adicionales.
        self.fill_dict[msg.orderId]["llenado"] = Verdadero

        # Coloque el evento de relleno en la cola de eventos
        self.events.put(fill_event)
```

Ahora que se han implementado todos los métodos anteriores, queda anular el ejecutar_ordenmétodo de laManejador de ejecuciónclase base abstracta. Este método realmente lleva a cabo la colocación del pedido con la API de IB.

Primero verificamos que el evento que se recibe con este método es en realidad unEvento de pedidoy luego preparar elContratoyOrdenarobjetos con sus respectivos parámetros. Una vez creados ambos el método IbPyrealizar pedidodel objeto de conexión se llama con un asociado Solicitar ID.

Está*extremadamente importante*llamar a latiempo.dormir(1)para asegurarse de que el pedido realmente llegue a IB. La eliminación de esta línea conduce a un comportamiento inconsistente de la API, ¡al menos en mi sistema!

Finalmente, incrementamos el ID del pedido para asegurarnos de no duplicar pedidos:

```
# ib_ejecucion.py

    definitivamenteejecutar_orden(uno mismo, evento):
        """
        Crea el objeto de orden de InteractiveBrokers necesario y lo envía a
        IB a través de su API.

        A continuación, se consultan los resultados para generar un
        objeto de relleno correspondiente, que se vuelve a colocar en la
        cola de eventos.

        Parámetros:
        evento: contiene un objeto de evento con información de pedido. """

        sievent.type == 'PEDIDO':
            # Preparar los parámetros para la orden de activos
            activo = evento.símbolo
            activo_tipo = "STK"
            tipo_pedido = evento.tipo_pedido
            cantidad = evento.cantidad dirección =
            evento.dirección

            # Cree el contrato de Interactive Brokers a través del
            # evento de pedido superado ib_contrato =
            self.create_contrato(
                activo, tipo_de_activo, self.order_routing,
                self.order_routing, self.currency
            )

            # Cree la orden de Interactive Brokers a través del
            # evento de pedido superado
            ib_order = self.create_order(
                order_type, cantidad, dirección
            )

            # Use la conexión para enviar el pedido a IB
            self.tws_conn.placeOrder(
                self.id_pedido, ib_contrato, ib_pedido
            )

            # NOTA: La siguiente línea es crucial.
            # ¡Asegura que el pedido se realice!
            tiempo.dormir(1)

            # Incrementar el ID de pedido para esta sesión
```

```
                self.order_id += 1
```

Esta clase forma la base de un controlador de ejecución de Interactive Brokers y se puede utilizar en lugar del controlador de ejecución simulado, que solo es adecuado para pruebas retrospectivas. Sin embargo, antes de que se pueda utilizar el controlador IB, es necesario crear un controlador de alimentación de mercado en vivo para reemplazar el controlador de alimentación de datos históricos del sistema backtester.

De esta manera, estamos reutilizando tanto como sea posible de los sistemas de prueba y en vivo para garantizar que el código "intercambio" se minimice y, por lo tanto, el comportamiento en ambos sea similar, si no idéntico.

Capítulo 15

Implementación de la estrategia comercial

En este capítulo vamos a considerar la implementación completa de las estrategias comerciales utilizando el sistema de backtesting basado en eventos mencionado anteriormente. En particular, generaremos curvas de equidad para todas las estrategias comerciales utilizando montos de cartera teóricos, simulando así los conceptos de margen/apalancamiento, que es un enfoque mucho más realista en comparación con los enfoques basados en rendimientos/vectorizados.

El primer conjunto de estrategias se puede llevar a cabo con datos disponibles gratuitamente, ya sea de Yahoo Finance, Google Finance o Quandl. Estas estrategias son adecuadas para operadores algorítmicos a largo plazo que deseen estudiar solo el aspecto de generación de señales comerciales de la estrategia o incluso el sistema completo de extremo a extremo. Tales estrategias a menudo poseen índices de Sharpe más pequeños, pero son mucho más fáciles de implementar y ejecutar.

Esta última estrategia se lleva a cabo utilizando datos de renta variable intradía. A menudo, estos datos no están disponibles de forma gratuita y, por lo general, se necesita un proveedor comercial de datos para proporcionar suficiente calidad y cantidad de datos. Yo mismo uso DTN IQFeed para barras intradiarias. Tales estrategias a menudo poseen relaciones de Sharpe mucho mayores, pero requieren una implementación más sofisticada ya que la alta frecuencia requiere una automatización extensa.

Veremos que nuestros primeros dos intentos de crear una estrategia comercial en datos interdiarios no son del todo exitosos. Puede ser un desafío idear una estrategia comercial rentable con datos entre días una vez que se han tenido en cuenta los costos de transacción. Esto último es algo que muchos textos sobre comercio algorítmico tienden a dejar de lado. Sin embargo, creo que se deben agregar tantos factores como sea posible al backtest para minimizar las sorpresas en el futuro.

Además, este libro trata principalmente sobre cómo crear de manera efectiva un sistema realista de backtesting interdiario o intradiario (así como una plataforma de ejecución en vivo) y menos sobre estrategias individuales particulares. ¡Es mucho más difícil crear un backtester robusto y realista que encontrar estrategias comerciales en Internet! Si bien las dos primeras estrategias presentadas no son particularmente atractivas, la última estrategia (en datos intradía) funciona bien y nos da confianza para usar datos de mayor frecuencia.

15.1 Estrategia de cruce de medias móviles

Me gusta mucho el sistema técnico de cruce de media móvil porque es la primera estrategia no trivial que es extremadamente útil para probar una nueva implementación de backtesting. En un período de tiempo diario, durante varios años, con largos períodos retrospectivos, se generan pocas señales en una sola acción y, por lo tanto, es fácil verificar manualmente que el sistema se comporta como se esperaría.

Para generar realmente una simulación de este tipo basada en el código de backtesting anterior, necesitamos subclase laEstrategiaobjeto como se describe en el capítulo anterior para crear elPromedio móvilEstrategia cruzada objeto, que contendrá la lógica de los promedios móviles simples y la generación de señales comerciales.

Además necesitamos crear el __principal__función que cargará elprueba retrospectivaobjeto y realmente encapsular la ejecución del programa. El siguiente archivo,mac.py,contiene ambos objetos.

La primera tarea, como siempre, es importar correctamente los componentes necesarios. Estamos importando casi todos los objetos que se han descrito en el capítulo anterior:

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# mac.py

de__futuro__importarimprimir_funcion

importarfecha y hora

importarnumpy como np importar
pandas como pd importar
statsmodels.api como sm

deestrategiaimportarEstrategia deeventoimportar
SeñalEvento deprueba retrospectivaimportarprueba
retrospectiva dedatosimportar
HistoricCSVDataHandlerHistóricoCSVDataHandler deejecución
importarSimulatedExecutionHandler deportafolioimportar
portafolio
```

Ahora pasamos a la creación de laEstrategia cruzada de promedio móvil.La estrategia requiere tanto las barrasmanejador de datos,los eventos Event Queue y los períodos retrospectivos para los promedios móviles simples que se van a emplear dentro de la estrategia. Elegí 100 y 400 como los períodos retrospectivos "corto" y "largo" para esta estrategia.

El último atributo,comprado,se usa para decirle a laEstrategiacuando el backtest está realmente "en el mercado". Las señales de entrada solo se generan si esto es "SALIDA" y las señales de salida solo se generan si esto es "LARGO" o "CORTO":

```
# mac.py

claseMedia móvilEstrategia cruzada(Estrategia):
    """
    Lleva a cabo una estrategia básica de cruce de medias móviles con una media
    móvil ponderada simple corta/larga. Las ventanas cortas/largas
    predeterminadas son 100/400 períodos respectivamente.
    """


    definitivamente__en eso__(
        yo, barras, eventos, short_window=100, long_window=400
    ) :
        """
        Inicializa la estrategia cruzada de medias móviles.

        Parámetros:
        bars: el objeto DataHandler que proporciona información de la barra
        events: el objeto Event Queue.
        short_window - El lookback de la media móvil corta.
        long_window: la vista retrospectiva de la media móvil larga.
        """
        self.barras = barras
        self.symbol_list = self.bars.symbol_list self.events =
        eventos
        self.ventana_corta = ventana_corta
        self.ventana_larga = ventana_larga
```

```
        # Establecer en Verdadero si un símbolo está en el
        mercado self.comprado = self._calculate_initial_bought()
```

Dado que la estrategia comienza fuera del mercado, establecemos que el valor inicial "comprado" sea "FUERA", para cada símbolo:

```
# mac.py

    definitivamente_calcular_inicial_comprado(auto):
        """
        Agrega claves al diccionario comprado para todos los símbolos
        y los establece en 'FUERA'.
        """
        comprado = {}
        porsenself.symbol_list:
            comprado[s] = 'FUERA'
        devolvercomprado
```

El núcleo de la estrategia es elcalcular_señalesmétodo. Reacciona a unMercadoEvento objeto y para cada símbolo negociado obtiene la última*norte*precios de cierre de bares, donde*norte*es igual al período retrospectivo más grande.

Luego calcula tanto el período corto como el largo.*medias móviles simples*. La regla de la estrategia es ingresar al mercado (ir en largo una acción) cuando el valor promedio móvil corto excede el valor promedio móvil largo. Por el contrario, si el valor de la media móvil larga supera el valor de la media móvil corta, se le dice a la estrategia que salga del mercado.

Esta lógica se maneja colocando unSeñalEventoobjeto en los eventos Cola de eventos en cada una de las situaciones respectivas y luego actualizar el atributo "comprado" (por símbolo) para que sea "LARGO" o "FUERA", respectivamente. Dado que esta es una estrategia solo larga, no consideraremos posiciones "CORTAS":

```
# mac.py

    definitivamentecalcular_señales(auto, evento):
        """
        Genera un nuevo conjunto de señales basado en MAC SMA con la
        ventana corta cruzando la ventana larga, lo que significa una
        entrada larga y viceversa para una entrada corta.

        Parámetros
        evento: un objeto MarketEvent. """

        sievent.type == 'MERCADO':
            porsenself.symbol_list:
                barras = self.barras.get_latest_bars_values(
                    s, "adj_close", N=self.long_window
                )
                bar_date = self.bars.get_latest_bar_datetime(s) sibarrasno es
                Ningunaybarras != []:
                    short_sma = np.mean(barras[-self.short_window:]) long_sma =
                    np.mean(bars[-self.long_window:])

                    símbolo = s
                    dt = fechahora.fechahora.utcnow() sig_dir
                    = ""

                    sicorto_sma > largo_smayself.comprado[s] == "FUERA":
                        impresión("LARGO: %s" % bar_fecha)
                        sig_dir = 'LARGO'
```

```
                                    señal = SignalEvent(1, símbolo, dt, sig_dir, 1.0)
                                    self.events.put(señal)
                                    self.comprado[s] = 'LARGO'
                                elifcorto_sma <largo_smayself.comprado[s] == "LARGO":
                                    impresión("CORTO: %s" % bar_fecha)
                                    sig_dir = 'SALIR'
                                    señal = SignalEvent(1, símbolo, dt, sig_dir, 1.0)
                                    self.events.put(señal)
                                    self.comprado[s] = 'FUERA'
```

Eso concluye elPromedio móvilEstrategia cruzadaimplementación de objetos. La tarea final de todo el sistema de backtesting es llenar un __principal__método enmac.pypara ejecutar realmente el backtest.

En primer lugar, asegúrese de cambiar el valor decsv_dira la ruta absoluta de su directorio de archivos CSV para los datos financieros. También deberá descargar el archivo CSV de las acciones de AAPL (de Yahoo Finance), que se encuentra en el siguiente enlace (del 1 de enero de 1990 al 1 de enero de 2002), ya que esta es la acción en la que probaremos la estrategia:

http://ichart.finance.yahoo.com/table.csv?s=AAPL&a=00&b=1&c=1990&d=00&e=1&f=2002&g=d&ignore=.csv

Asegúrese de colocar este archivo en la ruta señalada desde la función principal encsv_dir. Los __principal__ simplemente crea una instancia de un nuevo objeto de backtest y luego llama al método de simulación_comercio para ejecutarlo:

```
# mac.py

si__nombre__ == "__principal__":
    csv_dir = '/ruta/a/su/csv/archivo' symbol_list =     # ¡CAMBIA ESTO!
    ['AAPL']
    capital_inicial = 100000.0 latido del
    corazón = 0.0
    fecha_inicio = fechahora.fechahora(1990, 1, 1, 0, 0, 0)

    prueba inversa = prueba inversa (
        csv_dir, lista_símbolos, capital_inicial, latido, fecha_inicio, CSVDataHandler histórico,
        SimulatedExecutionHandler, Portafolio, MovingAverageCrossStrategy

    )
    backtest.simular_trading()
```

Para ejecutar el código, asegúrese de haber configurado un entorno de Python (como se describe en los capítulos anteriores) y luego navegue por el directorio donde está almacenado su código. Simplemente debería poder ejecutar:

```
python mac.py
```

Verá la siguiente lista (¡truncada debido a la impresión del conteo de barras!):

```
. .
. .
3029
3030
Creando estadísticas resumidas...
Creando curva de equidad...
                AAPL    comisión en efectivo retornos totales equity_curve drawdown
fecha y hora
2001-12-18      0   99211       13  99211       0   0.99211     0.025383
2001-12-19      0   99211       13  99211       0   0.99211     0.025383
2001-12-20      0   99211       13  99211       0   0.99211     0.025383
2001-12-21      0   99211       13  99211       0   0.99211     0.025383
```

```
2001-12-24   0   99211   13   99211   0   0.99211   0.025383
2001-12-26   0   99211   13   99211   0   0.99211   0.025383
2001-12-27   0   99211   13   99211   0   0.99211   0.025383
2001-12-28   0   99211   13   99211   0   0.99211   0.025383
2001-12-31   0   99211   13   99211   0   0.99211   0.025383
2001-12-31   0   99211   13   99211   0   0.99211   0.025383
[('Retorno total', '-0.79%'), ('Ratio de
  Sharpe', '-0.09'), ('Reducción máxima',
  '2.56%'), ('Duración de reducción',
  '2312')] Señales : 10

Pedidos: 10
Rellenos: 10
```

El rendimiento de esta estrategia se puede ver en la Fig. 15.1:

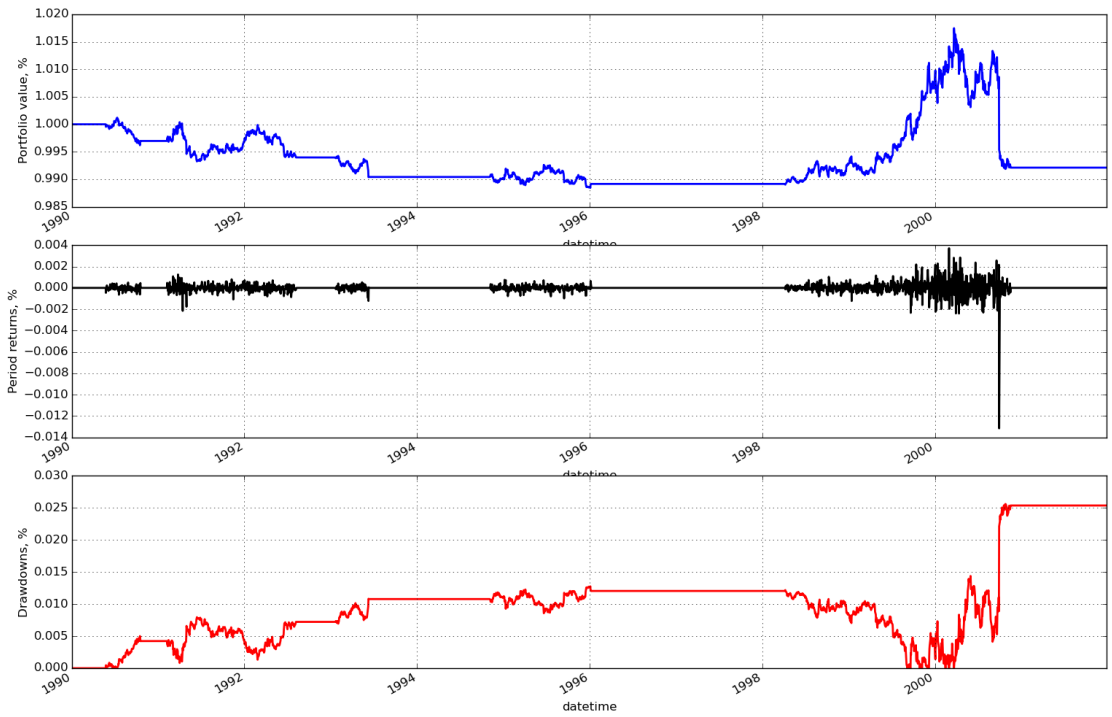

Figura 15.1: Curva de acciones, rendimientos diarios y reducciones para la estrategia de cruce de medias móviles

¡Evidentemente, los rendimientos y el índice de Sharpe no son estelares para las acciones de AAPL en este conjunto particular de indicadores técnicos! Claramente, tenemos trabajo por hacer en el próximo conjunto de estrategias para encontrar un sistema que pueda generar un rendimiento positivo.

15.2 S&P500 Pronóstico Comercial

En esta sección consideraremos una estrategia comercial construida alrededor del motor de pronóstico discutido en capítulos anteriores. Intentaremos compensar las predicciones hechas por un pronosticador del mercado de valores.

Vamos a intentar pronosticar SPY, que es el ETF que rastrea el valor del S&P500. En última instancia, queremos responder a la pregunta de si un algoritmo de pronóstico básico que utiliza datos de precios retrasados, con un ligero rendimiento predictivo, nos brinda algún beneficio sobre una estrategia de comprar y mantener.

Las reglas para esta estrategia son las siguientes:

1. Ajuste un modelo de pronóstico a un subconjunto de datos del S&P500. Esto podría ser una regresión logística, un analizador discriminante (lineal o cuadrático), una máquina de vectores de soporte o un bosque aleatorio. El procedimiento para hacer esto se describió en el capítulo Pronóstico.

2. Utilice dos retrasos anteriores de los datos de rendimientos de cierre ajustados como predictor de los rendimientos de mañana. Si los rendimientos se pronostican como positivos, vaya en largo. Si los rendimientos se pronostican como negativos, salga. No vamos a considerar la venta en corto para esta estrategia en particular.

Implementación

Para esta estrategia vamos a crear elsnp_forecast.py fie importar las siguientes bibliotecas necesarias:

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# snp_forecast.py

de__futuro__importarimprimir_funcion

importarfecha y hora

importarpandas como pd de
sklearn.qdaimportarQDA

deestrategiaimportarEstrategia deeventoimportar
SeñalEvento deprueba retrospectivaimportarprueba
retrospectiva dedatosimportar
HistoricCSVDataHandlerHistóricoCSVDataHandler deejecución
importarSimulatedExecutionHandler deportafolioimportar
portafolio
decrear_lag_serieimportarcrear_lag_serie
```

Hemos importado Pandas y Scikit-Learn para realizar el procedimiento de ajuste del modelo clasificador supervisado. También hemos importado elcrear_lag_seriefunción, que usamos en el capítulo Pronóstico.

El siguiente paso es crear elSPYDailyForecastEstrategiacomo una subclase de laEstrategia clase base abstracta. Dado que "codificaremos" los parámetros de la estrategia directamente en la clase, por simplicidad, los únicos parámetros necesarios para el __en eso__constructor son losbarras manejador de datos y eleventoscola.

Configuramos elself.model_***fechas de inicio/fin/prueba como objetos de fecha y hora y luego decirle a la clase que estamos fuera del mercado (self.long_market = Falso).Finalmente, establecemos auto.modeloser el modelo entrenado desde elcreate_symbol_forecast_modelabajo:

```
# snp_forecast.py

claseSPYDailyForecastStrategy(Estrategia):
    """
```

```
    Estrategia de previsión del S&P500. Utiliza un analizador discriminante
    cuadrático para predecir los retornos para un período de tiempo
    posterior y luego genera señales largas/de salida basadas en la
    predicción.
    """
    definitivamente__init__(auto, barras, eventos):
        self.barras = barras
        self.symbol_list = self.bars.symbol_list self.events =
        eventos
        self.datetime_now = datetime.datetime.utcnow()

        self.model_start_date = datetime.datetime(2001,1,10) self.model_end_date
        = datetime.datetime(2005,12,31) self.model_start_test_date =
        datetime.datetime(2005,1,1)

        self.long_market = Falso
        self.short_market = Falso
        self.bar_index = 0

        self.modelo = self.create_symbol_forecast_model()
```

Aquí definimos lacreate_symbol_forecast_model.Esencialmente llama a lacrear_lag_serie función, que produce un Pandas DataFrame con cinco retrasos de retorno diarios para cada predictor actual. Luego consideramos solo los dos más recientes de estos retrasos. Esto se debe a que estamos tomando la decisión de modelado de que es probable que el poder predictivo de los retrasos anteriores sea mínimo.

En esta etapa, creamos los datos de entrenamiento y prueba, el último de los cuales puede usarse para probar nuestro modelo si lo deseamos. He optado por no generar datos de prueba, ya que ya hemos entrenado el modelo antes en el capítulo Pronóstico. Finalmente, ajustamos los datos de entrenamiento al Analizador Discriminante Cuadrático y luego devolvemos el modelo.

Tenga en cuenta que podríamos reemplazar fácilmente el modelo con un bosque aleatorio, una máquina de vectores de soporte o una regresión logística, por ejemplo. Todo lo que tenemos que hacer es importar la biblioteca correcta de Scikit-Learn y simplemente reemplazar elmodelo = QDA()línea:

```
# snp_forecast.py

    definitivamentecreate_symbol_forecast_model(self):
        # Crear una serie rezagada del índice bursátil estadounidense S&P500 snpret
        = create_lagged_series(
            self.symbol_list[0], self.model_start_date,
            self.model_end_date, lags=5
        )

        # Use los dos días anteriores de devoluciones como predictor
        # valores, con dirección como respuesta X =
        snpret[["Lag1","Lag2"]] y = snpret["Dirección"]


        # Crear conjuntos de entrenamiento y prueba
        start_test = self.model_start_test_date X_train =
        X[X.index < start_test] X_test = X[X.index >=
        start_test] y_train = y[y.index < start_test] y_test =
        y[y.index >= start_test]


        modelo = QDA()
        modelo.fit(tren_X, tren_y) devolver
        modelo
```

En esta etapa estamos listos para anular elcalcular_señalesmetodo de laEstrategia clase básica. Primero calculamos algunos parámetros de conveniencia que entran en nuestroSeñalEventoobjeto y luego solo generar un conjunto de señales si hemos recibido unMercadoEventoobjeto (una comprobación de cordura básica).

Esperamos que hayan transcurrido cinco barras (¡es decir, cinco días en esta estrategia!) y luego obtenemos los valores de retorno retrasados. Luego envolvemos estos valores en una Serie Pandas para que elpredecirmétodo del modelo funcionará correctamente. Entonces calculamos una predicción, que se manifiesta como un + 1 o -1.

Si la predicción es un +1 y aún no estamos largos en el mercado, creamos unSeñalEvento ir largo y hacerle saber a la clase que ahora estamos en el mercado. Si la predicción es -1 y estamos largos en el mercado, simplemente salimos del mercado:

```
# snp_forecast.py

    definitivamentecalcular_señales(auto, evento):
        """
        Calcule los SignalEvents en función de los datos del mercado. """

        sym = self.símbolo_lista[0] dt =
        self.fechahora_ahora

        sievent.type == 'MERCADO':
            self.bar_index += 1
            siself.bar_index > 5:
                retrasos = self.bars.get_latest_bars_values(
                    self.symbol_list[0], "devoluciones", N=3
                )
                pred_series = pd.Series(
                    {
                        'Retraso1': retrasos[1]*100.0,
                        'Retraso2': retrasos[2]*100.0
                    }
                )
                pred = self.modelo.predecir(pred_series) sipresa >
                0y noself.long_market:
                    self.long_market = Verdadero
                    señal = SignalEvent(1, sym, dt, 'LONG', 1.0)
                    self.events.put(signal)

                sipresa < 0yself.long_market:
                    self.long_market = Falso
                    señal = SignalEvent(1, sym, dt, 'EXIT', 1.0)
                    self.events.put(señal)
```

Para ejecutar la estrategia, deberá descargar un archivo CSV de Yahoo Finance for SPY y colocarlo en un directorio adecuado (¡tenga en cuenta que deberá cambiar su ruta a continuación!). Luego terminamos el backtest a través de laprueba retrospectivaclase y realizar la prueba llamandosimular_comercio:

```
# snp_forecast.py

si__nombre__ == "__principal__":
    csv_dir = '/ruta/a/su/csv/archivo' symbol_list =     # ¡CAMBIA ESTO!
    ['SPY']
    capital_inicial = 100000.0 latido del
    corazón = 0.0
    fecha_inicio = fechahora.fechahora(2006,1,3)

    prueba inversa = prueba inversa (
```

```
        csv_dir, lista_símbolos, capital_inicial, latido, fecha_inicio, HistoricCSVDataHandler,
        SimulatedExecutionHandler, Portafolio, SPYDailyForecastStrategy

    )
    backtest.simular_trading()
```

El resultado de la estrategia es el siguiente y es neto de los costos de transacción:

```
. .
. .
2209
2210
Creando estadísticas resumidas...
Creando curva de equidad...
                ESPIAR      dinero  comisión      total  devoluciones  curva_equidad  \
fecha y hora
2014-09-29      19754     90563.3     349.7  110317.3     -0.000326       1.103173
2014-09-30      19702     90563.3     349.7  110265.3     -0.000471       1.102653
2014-10-01      19435     90563.3     349.7  109998.3     -0.002421       1.099983
2014-10-02      19438     90563.3     349.7  110001.3      0.000027       1.100013
2014-10-03      19652     90563.3     349.7  110215.3      0.001945       1.102153
2014-10-06      19629     90563.3     349.7  110192.3     -0.000209       1.101923
2014-10-07      19326     90563.3     349.7  109889.3     -0.002750       1.098893
2014-10-08      19664     90563.3     349.7  110227.3      0.003076       1.102273
2014-10-09      19274     90563.3     349.7  109837.3     -0.003538       1.098373
2014-10-09          0    109836.0     351.0  109836.0     -0.000012       1.098360

                reducción
fecha y hora
2014-09-29      0.003340
2014-09-30      0.003860
2014-10-01      0.006530
2014-10-02      0.006500
2014-10-03      0.004360
2014-10-06      0.004590
2014-10-07      0.007620
2014-10-08      0.004240
2014-10-09      0.008140
2014-10-09      0.008153
[('Retorno total', '9.84%'), ('Ratio de
  Sharpe', '0.54'), ('Reducción máxima',
  '5.99%'), ('Duración de la reducción',
  '811')] Señales: 270

Pedidos: 270
Rellenos: 270
```

La siguiente visualización en la Fig. 15.2 muestra la Curva de Equidad, los Retornos Diarios y el Drawdown de la estrategia en función del tiempo:

¡Observe de inmediato que el rendimiento no es excelente! Tenemos un índice de Sharpe < 1 pero una reducción razonable de poco menos del 6%. Resulta que si simplemente hubiéramos comprado y mantenido SPY en este período de tiempo, nos habríamos desempeñado de manera similar, aunque un poco peor.

Por lo tanto, en realidad no hemos ganado mucho con nuestra estrategia predictiva una vez que se incluyen los costos de transacción. Específicamente, quería incluir este ejemplo porque utiliza una implementación realista de "extremo a extremo" de una estrategia de este tipo que tiene en cuenta costos de transacción conservadores y realistas. Como puede verse, no es fácil hacer un pronosticador predictivo con datos diarios que produzca un buen desempeño.

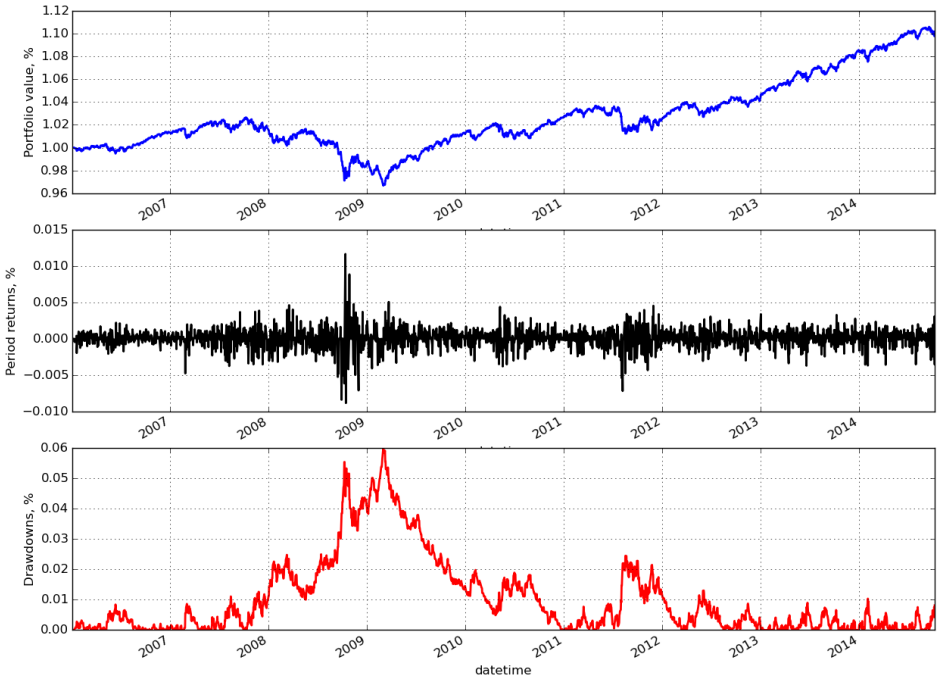

Figura 15.2: Curva de acciones, rendimientos diarios y reducciones para la estrategia de pronóstico SPY

Nuestra estrategia final utilizará otras series de tiempo y una frecuencia más alta. Veremos que el rendimiento se puede mejorar drásticamente tras modificar ciertos aspectos del sistema.

15.3 Comercio de pares de acciones con reversión a la media

Para buscar índices de Sharpe más altos para nuestras operaciones, debemos considerar estrategias intradía de mayor frecuencia.

El primer problema importante es que la obtención de datos es significativamente menos sencilla porque los datos intradía de alta calidad generalmente no son gratuitos. Como se indicó anteriormente, utilizo DTN IQFeed para las barras minuciosas intradía y, por lo tanto, necesitará su propia cuenta DTN para obtener los datos necesarios para esta estrategia.

El segundo problema es que las simulaciones de backtesting toman mucho más tiempo, especialmente con el modelo basado en eventos que hemos construido aquí. Una vez que comenzamos a considerar una prueba retrospectiva de una cartera diversificada de datos minuciosos que abarcan años, y luego realizamos cualquier optimización de parámetros, nos damos cuenta rápidamente de que las simulaciones pueden tardar horas o incluso días en calcularse en una PC de escritorio moderna. Esto deberá tenerse en cuenta en su proceso de investigación.

El tercer problema es que la ejecución en vivo ahora deberá estar completamente automatizada, ya que estamos acercándonos a operaciones de mayor frecuencia. Esto significa que dichos entornos de ejecución y código deben ser altamente confiables y libres de errores, de lo contrario, pueden ocurrir pérdidas significativas.

Esta estrategia amplía la estrategia interdiaria anterior para hacer uso de los datos intradiarios. En particular, vamos a utilizar barras de OHLCV minuciosas, en lugar de OHLCV diarias.

Las reglas de la estrategia son sencillas:

1. Identifique un par de acciones que posean una serie temporal de residuos que se haya identificado estadísticamente como de reversión a la media. En este caso, he encontrado dos acciones estadounidenses del sector energético con tickers AREX y WLL.

2. Cree las series de tiempo de residuos del par realizando una regresión lineal móvil, para una ventana retrospectiva en particular, a través del algoritmo de mínimos cuadrados ordinarios (OLS). Este período retrospectivo es un parámetro a optimizar.

3. Cree una puntuación z continua de la serie temporal de residuos del mismo período retrospectivo y utilícela para determinar los umbrales de entrada/salida para las señales comerciales.

4. Si se supera el umbral superior cuando no está en el mercado, ingrese al mercado (largo o corto dependiendo de la dirección del exceso del umbral). Si se supera el umbral inferior cuando está en el mercado, salga del mercado. Una vez más, los umbrales superior e inferior son parámetros a optimizar.

De hecho, podríamos haber utilizado la prueba Dickey-Fuller aumentada cointegrada (CADF) para identificar un parámetro de cobertura aún más preciso. Esto sería una extensión interesante de la estrategia.

Implementación

El primer paso, como siempre, es importar las bibliotecas necesarias. Requerimos pandas para el rodar_aplicarmétodo, que se utiliza para aplicar el cálculo de la puntuación z con una ventana retrospectiva de forma continua. Importamos statsmodels porque proporciona un medio para calcular el algoritmo de mínimos cuadrados ordinarios (OLS) para la regresión lineal, necesario para obtener el índice de cobertura para la construcción de los residuos.

También requerimos una ligera modificación.manejador de datosyportafolioCon el fin de llevar a cabo el comercio de barras minuciosamente en los datos de DTN IQFeed. Para crear estos archivos, simplemente puede copiar todo el código enportafolio.pyydatos.pyen los nuevos archivoshft_portafolio.pyy hft_data.pyrespectivamente y luego modificar los apartados necesarios, que a continuación detallaré.

Aquí está la lista de importación paraintraday_mr.py:

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# intraday_mr.py

de__futuro__importarimprimir_funcion

importarfecha y hora

importarnumpy como np importar
pandas como pd importar
statsmodels.api como sm

deestrategiaimportarEstrategia deevento
importarSeñalEvento deprueba
retrospectivaimportarprueba retrospectiva
dedatos_hftimportarHistoricCSVDataHandlerHFT dehft_cartera
importarPortafolioHFT deejecuciónimportar
SimulatedExecutionHandler
```

En el siguiente fragmento creamos elIntradíaOLSMRestrategiaclase derivada de la Estrategiaclase base abstracta. El constructor __en eso__El método requiere acceso a labarras proveedor de datos históricos, eleventoscola, unzscore_bajoumbral y unzscore_altoumbral, utilizado para determinar cuándo la serie residual entre los dos pares es de reversión a la media.

Además, especificamos la ventana retrospectiva OLS (establecida en 100 aquí), que es un parámetro que está sujeto a una posible optimización. Al comienzo de la simulación no somos ni largos ni corto el mercado, por lo que establecemos ambosself.long_marketyself.short_marketigual aFalso:

```
# intraday_mr.py
```

```
clase IntradíaOLSMRStrategy(Estrategia):
    """
    Utiliza mínimos cuadrados ordinarios (OLS) para realizar una regresión lineal móvil para
    determinar el índice de cobertura entre un par de acciones. La puntuación z de la serie
    temporal de residuos se calcula de forma continua y si supera un intervalo de umbrales
    (predeterminado en [0,5, 3,0]), se genera un par de señales largas/cortas (para el umbral
    alto) o una se generan pares de señales de salida (para el umbral bajo).


    """

    definitivamente __en eso__(
        yo, barras, eventos, ols_window=100,
        zscore_low=0.5, zscore_high=3.0
    ) :
        """
        Inicializa la estrategia de arbitraje estadístico.

        Parámetros:
        bars: el objeto DataHandler que proporciona información de la barra
        events: el objeto Event Queue.
        """
        self.barras = barras
        self.symbol_list = self.bars.symbol_list self.events =
        eventos
        self.ols_window = ols_window
        self.zscore_low = zscore_low
        self.zscore_high = zscore_high

        self.par = ('AREX', 'WLL') self.fechahora =
        fechahora.fechahora.utcnow()

        self.long_market = Falso
        self.short_market = Falso
```

El siguiente método,calcular_xy_señales,toma el zscore actual (del cálculo continuo realizado a continuación) y determina si es necesario generar nuevas señales comerciales. Estas señales luego se devuelven.

Hay cuatro estados potenciales que nos pueden interesar. Son:

1. Mercado largo y por debajo del umbral superior de puntuación z negativa
2. Mercado largo y entre el valor absoluto del umbral inferior de zscore
3. Acortar el mercado y por encima del umbral superior de puntuación z positiva
4. Acortar el mercado y entre el valor absoluto del umbral inferior de zscore

En cualquier caso, es necesario generar dos señales, una para el primer componente del par (AREX) y otra para el segundo componente del par (WLL). Si no se alcanza ninguna de estas condiciones, entonces un par deNingunase devuelven valores:

```
# intraday_mr.py

    definitivamente calcular_xy_señales(uno mismo, zscore_last):
        """
        Calcula los pares de señales x, y reales que se
        enviarán al generador de señales.
```

```
        Parámetros
        zscore_last: el zscore actual para probar contra """

        y_señal = Ninguno
        señal_x = Ninguno
        p0 = self.par[0]
        p1 = self.par[1]
        dt = self.fechahora
        hr = abs(self.hedge_ratio)

        # Si estamos largos en el mercado y por debajo del
        # negativo del umbral alto de zscore
        sizscore_last <= -self.zscore_highy noself.long_market:
            self.long_market = Verdadero
            y_signal = SignalEvent(1, p0, dt, 'LARGO', 1.0) x_signal =
            SignalEvent(1, p1, dt, 'CORTO', hr)

        # Si estamos largos en el mercado y entre el
        # valor absoluto del umbral bajo de zscore
        siabs(zscore_last) <= self.zscore_lowyself.long_market:
            self.long_market = Falso
            y_signal = SignalEvent(1, p0, dt, 'SALIR', 1.0) x_signal =
            SignalEvent(1, p1, dt, 'SALIR', 1.0)

        # Si estamos cortos en el mercado y por encima
        # el umbral alto de zscore
        sizscore_last >= self.zscore_highy noself.short_market:
            self.short_market = Verdadero
            y_signal = SignalEvent(1, p0, dt, 'CORTO', 1.0) x_signal =
            SignalEvent(1, p1, dt, 'LARGO', hr)

        # Si estamos cortos en el mercado y entre el
        # valor absoluto del umbral bajo de zscore
        siabs(zscore_last) <= self.zscore_lowyself.short_market:
            self.short_market = Falso
            y_signal = SignalEvent(1, p0, dt, 'SALIR', 1.0) x_signal =
            SignalEvent(1, p1, dt, 'SALIR', 1.0)

        devolverseñal_y, señal_x
```

El siguiente método,calcular_señales_para_paresobtiene el último conjunto de barras para cada componente del par (en este caso, 100 barras) y las utiliza para construir una regresión lineal basada en mínimos cuadrados ordinarios. Esto permite identificar el ratio de cobertura, necesario para la construcción de la serie temporal de residuos.

Una vez que se construye el índice de cobertura, ununtadose construye una serie de residuos. El siguiente paso es calcular el último zscore de la serie residual restando su media y dividiendo por su desviación estándar durante el período retrospectivo.

Finalmente, ely_señalyseñal_xse calculan sobre la base de este zscore. Si las señales no son ambasNingunaentonces elSeñalEventolas instancias se devuelven aleventoscola:

```
# intraday_mr.py

    definitivamentecalcular_señales_para_pares(auto):
        """
        Genera un nuevo conjunto de señales basado en la estrategia de reversión a la
        media.
```

```
        Calcula la relación de cobertura entre el par de tickers. Usamos OLS para
        esto, aunque lo ideal sería usar CADF. """

        # Obtener la última ventana de valores para cada
        # componente del par de tickers y =
        self.barras.get_latest_bars_values(
            self.par[0], "cerrar", N=self.ols_window
        )
        x = self.bars.get_latest_bars_values(
            self.par[1], "cerrar", N=self.ols_window
        )

        siyno esNingunayXno esNinguna:
            # Verifique que todos los períodos de ventana estén disponibles
            silen(y) >= self.ols_windowylen(x) >= self.ols_window:
                # Calcule el índice de cobertura actual usando OLS
                self.hedge_ratio = sm.OLS(y, x).fit().params[0]

                # Calcule el puntaje z actual de los residuos spread = y -
                self.hedge_ratio * x
                zscore_last = ((spread - spread.mean())/spread.std())[-1]

                # Calcular señales y agregar a la cola de eventos
                señal_y, señal_x = self.calculate_xy_signals(zscore_last) siy_señalno esNingunay
                señal_xno esNinguna:
                    self.events.put(y_signal)
                    self.events.put(x_signal)
```

El último método,calcular_señalesse anula de la clase base y se usa para verificar si un evento recibido de la cola es realmente unevento de mercado,en cuyo caso se realiza el cálculo de las nuevas señales:

```
# intraday_mr.py

    definitivamentecalcular_señales(auto, evento):
        """
        Calcule los SignalEvents en función de los datos del mercado. """

        sievent.type == 'MERCADO':
            self.calculate_signals_for_pairs()
```

Los __principal__La sección une los componentes para producir un backtest para la estrategia. Le decimos a la simulación dónde se almacenan los datos de ticker minuciosamente. Estoy usando el formato DTN IQFeed. Trunqué ambos archivos para que comenzaran y terminaran en el mismo minuto respectivo. Para este par particular de AREX y WLL, la fecha de inicio común es el 8 de noviembre de 2007 a las 10:41:00 a. m.

Finalmente, construimos el objeto de backtest y comenzamos a simular el comercio:

```
# intraday_mr.py

si__nombre__ == "__principal__":
    csv_dir = '/ruta/a/su/csv/archivo' symbol_list =     # ¡CAMBIA ESTO!
    ['AREX', 'WLL'] initial_capital = 100000.0

    latido del corazón = 0.0
    fecha_inicio = fechahora.fechahora(2007, 11, 8, 10, 41, 0)

    prueba inversa = prueba inversa (
        csv_dir, lista_de_símbolos, capital_inicial, latido del corazón,
```

```
        start_date, HistoricCSVDataHandlerHFT, SimulatedExecutionHandler, PortfolioHFT,
        IntradayOLSMRStrategy
    )
    backtest.simular_trading()
```

Sin embargo, antes de que podamos ejecutar este archivo, debemos realizar algunas modificaciones en el controlador de datos y en los objetos de la cartera.

En particular, es necesario crear nuevos archivoshft_data.pyyhft_portafolio.pyque son copias dedatos.pyyportafolio.pyrespectivamente.

Enhft_data.pytenemos que cambiar el nombreHistoricCSVDataHandlerHistóricoCSVDataHandleraHistoricCSVDataHandlerHFT y reemplace elnombreslista en el _abrir_convertir_archivos_csvmétodo.

La línea antigua es:

```
                    nombres=[
                        'datetime', 'open', 'high', 'low', 'close', 'volume',
                        'adj_close'
                    ]
```

Esto debe ser reemplazado por:

```
                    nombres=[
                        'fechahora', 'abrir', 'bajo', 'alto', 'cerrar',
                        'volumen', 'oi'
                    ]
```

Esto es para garantizar que el nuevo formato de DTN IQFeed funcione con el backtester.

El otro cambio es cambiar el nombreportafolioaPortafolioHFTenhft_portafolio.py.Nosotros luego debe modificar algunas líneas para tener en cuenta la frecuencia mínima de los datos DTN.

En particular, dentro delupdate_timeindexmétodo, debemos cambiar el siguiente código:

```
        porsenself.symbol_list:
            # Aproximación al valor real valor_mercado =
            self.posiciones_actuales[s] * \
                self.bars.get_latest_bar_value(s, "adj_close") dh[s] =
                valor_mercado
            dh['total'] += valor_de_mercado
```

A:

```
        porsenself.symbol_list:
            # Aproximación al valor real valor_mercado =
            self.posiciones_actuales[s] * \
                self.bars.get_latest_bar_value(s, "cerrar") dh[s] =
                valor_mercado
            dh['total'] += valor_de_mercado
```

Esto asegura que obtengamos lacercaprecio, en lugar deladj_cerrarprecio. El último es para Yahoo Finance, mientras que el primero es para DTN IQFeed.

También debemos hacer un ajuste similar enactualizar_existencias_desde_llenar.Necesitamos cambiar el siguiente código:

```
        # Actualizar la lista de existencias con nuevas
        cantidades fill_cost = self.bars.get_latest_bar_value(
            llenar.símbolo, "adj_cerrar"
        )
```

A:

```
        # Actualizar la lista de existencias con nuevas
        cantidades fill_cost = self.bars.get_latest_bar_value(
            llenar.símbolo, "cerrar"
        )
```

El cambio final se produce en elresumen_de_estadísticas_de_salidamétodo en la parte inferior del archivo. Necesitamos modificar la forma en que se calcula el índice de Sharpe para tener en cuenta el comercio minucioso. La siguiente línea:

```
        sharpe_ratio = create_sharpe_ratio(devoluciones)
```

Debe ser cambiado a:

```
        sharpe_ratio = create_sharpe_ratio(devoluciones, períodos=252*6,5*60)
```

Esto completa los cambios necesarios. Tras la ejecución deintraday_mr.pyobtenemos el siguiente resultado (truncado) de la simulación de backtest:

```
. .
. .
375072
375073
Creando estadísticas resumidas...
Creando curva de equidad...
                          AREX   WLL       dinero  comisión        total  devoluciones  \
fecha y hora
2014-03-11 15:53:00       2098 -6802   120604.3      9721.4  115900.3 -0.000052
2014-03-11 15:54:00       2101 -6799   120604.3      9721.4  115906.3  0.000052
2014-03-11 15:55:00       2100 -6802   120604.3      9721.4  115902.3 -0.000035
2014-03-11 15:56:00       2097 -6810   120604.3      9721.4  115891.3 -0.000095
2014-03-11 15:57:00       2098 -6801   120604.3      9721.4  115901.3  0.000086
2014-03-11 15:58:00       2098 -6800   120604.3      9721.4  115902.3  0.000009
2014-03-11 15:59:00       2099 -6800   120604.3      9721.4  115903.3  0.000009
2014-03-11 16:00:00       2100 -6801   120604.3      9721.4  115903.3  0.000000
2014-03-11 16:01:00       2100 -6801   120604.3      9721.4  115903.3  0.000000
2014-03-11 16:01:00       2100 -6801   120604.3      9721.4  115903.3  0.000000

                          curva_equidad  reducción
fecha y hora
2014-03-11 15:53:00            1.159003   0.003933
2014-03-11 15:54:00            1.159063   0.003873
2014-03-11 15:55:00            1.159023   0.003913
2014-03-11 15:56:00            1.158913   0.004023
2014-03-11 15:57:00            1.159013   0.003923
2014-03-11 15:58:00            1.159023   0.003913
2014-03-11 15:59:00            1.159033   0.003903
2014-03-11 16:00:00            1.159033   0.003903
2014-03-11 16:01:00            1.159033   0.003903
2014-03-11 16:01:00            1.159033   0.003903
[('Retorno total', '15,90 %'), ('Ratio de
  Sharpe', '1,89'), ('Reducción máxima', '3,03
  %'), ('Duración de la reducción', '120718')]
  Señales: 7594

Pedidos: 7478
Rellenos: 7478
```

Puede ver que la estrategia funciona adecuadamente durante este período. Tiene un rendimiento total de poco menos del 16%. El índice de Sharpe es razonable (en comparación con una estrategia diaria típica), pero dada la naturaleza de alta frecuencia de la estrategia, deberíamos esperar más. El mayor atractivo de esta estrategia es que la reducción máxima es baja (aproximadamente 3%). Esto sugiere que podríamos aplicar más apalancamiento para obtener más rendimiento.

El rendimiento de esta estrategia se puede ver en la Fig. 15.3:

Tenga en cuenta que estas cifras se basan en la negociación de un total de 100 acciones. Puede ajustar el apalancamiento simplemente ajustando elgenerar_pedido_ingenuometodo de laportafolioclase. Busca el

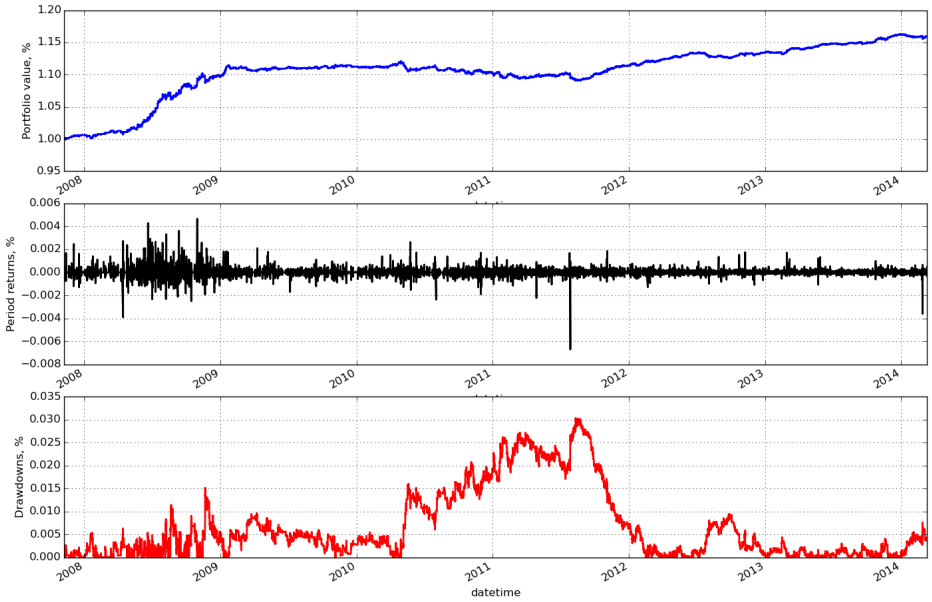

Figura 15.3: Curva de acciones, rendimientos diarios y retiros para la estrategia de reversión a la media intradía

atributo conocido comomkt_quantity.Se establecerá en 100. Cambiar esto a 2000, por ejemplo, proporciona estos resultados:

```
. .
. .
[('Retorno total', '392,85 %'), ('Ratio de
  Sharpe', '2,29'), ('Reducción máxima', '45,69
  %'), ('Duración de la reducción', '102150')] . .



. .
```

Claramente, el índice de Sharpe y el rendimiento total son mucho más atractivos, ¡pero también tenemos que soportar una reducción máxima del 45% durante este período!

15.4 Rendimiento de trazado

Las tres figuras que se muestran arriba se crean usando elplot_performance.pyguion. Para completar, he incluido el código para que pueda usarlo como base para crear sus propios gráficos de rendimiento.

Es necesario ejecutar esto en el mismo directorio que el archivo de salida del backtest, es decir, dondeequidad.csvreside El listado es el siguiente:

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# plot_rendimiento.py

importaros.ruta
```

```
importarnumpy como np
importarmatplotlib.pyplot como plt
importarpandas como pd


si__nombre__ == "__principal__":
    datos = pd.io.parsers.read_csv(
        "equidad.csv", encabezado=0,
        parse_dates=Verdadero, index_col=0
    ) . clasificar()

    # Trace tres gráficos: Curva de equidad,
    # rendimientos del período,
    retiros fig = plt.figura()
    # Establecer el color exterior en blanco
    fig.patch.set_facecolor('blanco')

    # Trazar la curva de equidad
    ax1 = fig.add_subplot(311, ylabel='Valor de la cartera, %')
    data['equity_curve'].plot(ax=ax1, color="blue", lw=2.) plt.grid(True)


    # Graficar los retornos
    ax2 = fig.add_subplot(312, ylabel='Devoluciones del período, %')
    data['devoluciones'].plot(ax=ax2, color="black", lw=2.) plt.grid(True)


    # Graficar los retornos
    ax3 = fig.add_subplot(313, ylabel='Drawdowns, %')
    data['drawdown'].plot(ax=ax3, color="red", lw=2.) plt.grid(True)


    # Graficar la figura
    plt.mostrar()
```

capitulo 16

Optimización de la estrategia

En capítulos anteriores, hemos considerado cómo crear un modelo predictivo subyacente (como con la máquina de vectores de soporte y el clasificador de bosques aleatorios), así como una estrategia comercial basada en él. Por el camino hemos visto que hay muchos *parámetros* a tales modelos. En el caso de un SVM tenemos los parámetros de "tuning" *y* y *C*. En una estrategia comercial de cruce de media móvil, tenemos los parámetros para las dos ventanas retrospectivas de los filtros de media móvil.

En este capítulo vamos a describir métodos de optimización para mejorar el rendimiento de nuestras estrategias comerciales ajustando los parámetros de forma sistemática. Para ello utilizaremos mecanismos del campo estadístico de *Selección de modelo*, como la validación cruzada y la búsqueda en cuadrícula. La literatura sobre la selección de modelos y la optimización de parámetros es amplia y la mayoría de los métodos están más allá del alcance de este libro. Quiero introducir el tema aquí para que puedas explorar técnicas más sofisticadas a tu propio ritmo.

16.1 Optimización de parámetros

En esta etapa, casi todas las estrategias comerciales y los modelos estadísticos subyacentes han requerido uno o más parámetros para ser utilizados. En las estrategias de impulso que utilizan indicadores técnicos, como las medias móviles (simples o exponenciales), es necesario especificar una ventana retrospectiva. Lo mismo ocurre con muchas estrategias de reversión a la media, que requieren una ventana retrospectiva (móvil) para calcular una regresión entre dos series temporales. Los modelos de aprendizaje automático estadístico particulares, como una regresión logística, SVM o Random Forest, también requieren parámetros para poder calcularse.

El mayor peligro al considerar la optimización de parámetros es el de *sobreajuste* un modelo o estrategia comercial. Este problema ocurre cuando se entrena un modelo en un *en la muestra* segmento retenido de datos de entrenamiento y está optimizado para funcionar bien (según la medida de rendimiento adecuada), pero el rendimiento se degrada sustancialmente cuando se aplica a *fuera de muestra* datos. Por ejemplo, una estrategia comercial podría funcionar extremadamente bien en la prueba retrospectiva (en los datos de muestra), pero cuando se implementa para el comercio en vivo puede no ser rentable.

Una preocupación adicional de la optimización de parámetros es que puede volverse muy costosa desde el punto de vista computacional. Con los sistemas informáticos modernos, esto es un problema menor que antes, debido a la paralelización y las CPU rápidas. Sin embargo, la optimización de múltiples parámetros puede aumentar la complejidad computacional en órdenes de magnitud. Uno debe ser consciente de esto como parte del proceso de investigación y desarrollo.

16.1.1 ¿Qué parámetros optimizar?

Un modelo de comercio algorítmico basado en estadísticas a menudo tendrá muchos parámetros y diferentes medidas de rendimiento. Un algoritmo de aprendizaje estadístico subyacente tendrá su propio conjunto de parámetros. En el caso de una regresión lineal múltiple o logística, estos serían los β coeficientes En el caso de un bosque aleatorio, uno de esos parámetros sería el número de árboles de decisión subyacentes que se utilizarán en el conjunto. Una vez aplicado a un modelo comercial, otros parámetros pueden ser de entrada.

y umbrales de salida, como una puntuación z de una serie de tiempo particular. El puntaje z en sí mismo podría tener una ventana retrospectiva móvil implícita. Como puede verse, el número de parámetros puede ser bastante extenso.

Además de los parámetros, existen numerosos medios para evaluar el rendimiento de un modelo estadístico y la estrategia comercial basada en él. Hemos definido conceptos como la tasa de acierto y la matriz de confusión. Además, hay más medidas estadísticas como el *Error medio cuadrado*(MSE). Estas son medidas de rendimiento que se optimizarían a nivel de modelo estadístico, a través de parámetros relevantes para su dominio.

La estrategia comercial real se evalúa según diferentes criterios, como la tasa de crecimiento anual compuesta (CAGR) y la reducción máxima. Necesitaríamos variar los criterios de entrada y salida, así como otros umbrales que no están directamente relacionados con el modelo estadístico. Por lo tanto, esto motiva la pregunta sobre qué conjunto de parámetros optimizar y cuándo.

En las siguientes secciones vamos a optimizar tanto los parámetros del modelo estadístico, en la etapa inicial de investigación y desarrollo, así como los parámetros asociados con una estrategia comercial utilizando un modelo estadístico optimizado subyacente, en cada una de sus respectivas medidas de rendimiento.

16.1.2 La optimización es costosa

Con múltiples parámetros de valor real, la optimización puede volverse extremadamente costosa rápidamente, ya que cada nuevo parámetro agrega una dimensión espacial adicional. Si consideramos el ejemplo de un*búsqueda de cuadrícula*(a ser discutido en su totalidad a continuación), y tienen un solo parámetroα, entonces podríamos desear variarαdentro del conjunto$\{0.1,0.2,0.3,0.4,0.5\}$. Esto requiere 5 simulaciones.

Si ahora consideramos un parámetro adicionalβ, que puede variar en el rango$\{0.2,0.4,0.6,0.8,1.0\}$, entonces tendremos que considerar$5_2=25$simulaciones Otro parámetro,γ, con 5 variaciones trae esto a$5_3=125$simulaciones Si cada parámetro tuviera 10 valores separados para probar, esto sería igual a$10_3=1000$simulaciones Como se puede ver, el espacio de búsqueda de parámetros puede hacer rápidamente que tales simulaciones sean extremadamente costosas.

Está claro que existe una compensación entre realizar una búsqueda exhaustiva de parámetros y mantener un tiempo de simulación total razonable. Si bien el paralelismo, incluidas las CPU de muchos núcleos y las unidades de procesamiento de gráficos (GPU), ha mitigado un poco el problema, aún debemos tener cuidado al introducir parámetros. Esta noción de reducción de parámetros también es un problema de efectividad del modelo, como veremos más adelante.

16.1.3 Reequipamiento

El sobreajuste es el proceso de optimizar un parámetro, o conjunto de parámetros, frente a un conjunto de datos en particular, de modo que se encuentra que una medida de rendimiento (o medida de error) adecuada se maximiza (o minimiza), pero cuando se aplica a un conjunto de datos no visto, tal una medida de desempeño se degrada sustancialmente. El concepto está íntimamente relacionado con la idea de*dilema de sesgo-varianza*.

El dilema de sesgo-varianza se refiere a la situación en la que un modelo estadístico tiene un equilibrio entre ser un modelo de bajo sesgo o un modelo de baja varianza, o un compromiso entre los dos.*Parcialidad* se refiere a la diferencia entre la estimación del modelo de un parámetro y el verdadero valor de "población" del parámetro, o supuestos erróneos en el modelo estadístico.*Diferencia*se refiere al error de la sensibilidad del modelo a pequeñas fluctuaciones en el conjunto de entrenamiento (en datos de muestra).

En todos los modelos estadísticos, uno intenta simultáneamente minimizar tanto el error de sesgo como el error de varianza para mejorar la precisión del modelo. Tal situación puede conducir a un sobreajuste en los modelos, ya que el error de entrenamiento puede reducirse sustancialmente al introducir modelos con más flexibilidad (variación). Sin embargo, estos modelos pueden funcionar extremadamente mal con datos nuevos (fuera de la muestra), ya que esencialmente se "ajustaron" a los datos de la muestra.

Un ejemplo común de un modelo de alto sesgo y baja varianza es el de la regresión lineal aplicada a un conjunto de datos no lineales. Las adiciones de nuevos puntos no afectan drásticamente la pendiente de regresión (suponiendo que no estén demasiado lejos de los datos restantes), pero dado que el problema es inherentemente no lineal, existe un sesgo sistemático en los resultados al usar un modelo lineal.

Un ejemplo común de un modelo de baja desviación y alta varianza es el de un ajuste spline polinomial aplicado a un conjunto de datos no lineales. El parámetro del modelo (el grado del polinomio)

podría ajustarse para ajustarse a un modelo de este tipo con mucha precisión (es decir, un sesgo bajo en los datos de entrenamiento), pero la adición de nuevos puntos conduciría casi con seguridad a que el modelo tuviera que modificar su grado de polinomio para ajustarse a los nuevos datos. Esto lo convertiría en un modelo de varianza muy alta en los datos de la muestra. Es probable que un modelo de este tipo tenga una previsibilidad o una capacidad de inferencia muy deficientes en los datos fuera de la muestra.

El sobreajuste también puede manifestarse en la estrategia comercial y no solo en el modelo estadístico. Por ejemplo, podríamos optimizar el índice de Sharpe variando los parámetros de umbral de entrada y salida. Si bien esto puede mejorar la rentabilidad en el backtest (o minimizar el riesgo sustancialmente), probablemente no sería un comportamiento que se replique cuando la estrategia se implementó en vivo, ya que podríamos haber ajustado tales optimizaciones al ruido en los datos históricos.

Discutiremos las técnicas a continuación para minimizar el sobreajuste, tanto como sea posible. Sin embargo, hay que ser consciente de que es un peligro siempre presente tanto en el comercio algorítmico como en el análisis estadístico en general.

16.2 Selección de modelo

En esta sección vamos a considerar cómo optimizar el modelo estadístico que será la base de una estrategia comercial. En el campo de la estadística y el aprendizaje automático esto se conoce como *Selección de modelo*. Si bien no presentaré una discusión exhaustiva sobre las diversas técnicas de selección de modelos, describiré algunos de los mecanismos básicos, como *Validación cruzada* y *Búsqueda de cuadrícula* que funcionan bien para las estrategias comerciales.

16.2.1 Validación cruzada

La validación cruzada es una técnica utilizada para evaluar cómo se generalizará un modelo estadístico a nuevos datos a los que no se ha expuesto antes. Esta técnica suele utilizarse en modelos predictivos, como los clasificadores supervisados antes mencionados, que se utilizan para predecir el signo de los siguientes rendimientos diarios de una serie de precios de activos. Fundamentalmente, el objetivo de la validación cruzada es minimizar el error en los datos de la muestra sin generar un modelo sobreajustado.

En esta sección describiremos la *división de entrenamiento/prueba* y *validación cruzada k-fold*, así como utilizar técnicas dentro de Scikit-Learn para realizar automáticamente estos procedimientos sobre modelos estadísticos que ya hemos desarrollado.

Tren/División de prueba

El ejemplo más simple de validación cruzada se conoce como *división de entrenamiento/prueba*, o un *Validación cruzada doble*. Una vez que se ensambla un conjunto de datos históricos anterior (como una serie de tiempo diaria de precios de activos), se divide en dos componentes. La relación de la división suele variar entre 0,5 y 0,8. En el último caso, esto significa que el 80 % de los datos se utilizan para entrenamiento y el 20 % para pruebas. Todas las estadísticas de interés, como la tasa de aciertos, la matriz de confusión o el error cuadrático medio, se calculan en el conjunto de prueba, que no se ha utilizado en el proceso de entrenamiento.

Para realizar este proceso en Python con Scikit-Learn podemos utilizar el sklearn cross_validation train_test_split método. Continuaremos con nuestro modelo como se discutió en el capítulo sobre Pronóstico. En particular, vamos a modificar previsión.py y crea un nuevo archivo llamado tren_prueba_split.py. Tendremos que agregar la nueva importación a la lista de importaciones:

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# tren_prueba_split.py

de __futuro__ importar imprimir_funcion

importar fecha y hora

importar aprender
```

```
desklearn.cross_validationimportartren_prueba_dividir de
sklearn.ensembleimportarRandomForestClassifier de
sklearn.modelo_linealimportarRegresión logística desklearn.lda
importarLDA
desklearn.metricsimportarmatriz de confusión de
sklearn.qdaimportarQDA
desklearn.svmimportarSVC lineal, SVC

decrear_lag_serieimportarcrear_lag_serie
```

Enprevisión.pyoriginalmente dividimos los datos en función de una fecha particular dentro de la serie temporal:

```
# pronóstico.py

    . .

    # Los datos de prueba se dividen en dos partes: antes y después del 1 de enero de 2005.
    prueba_inicio = fechahora.fechahora(2005,1,1)

    # Crear conjuntos de entrenamiento y
    prueba X_train = X[X.index < start_test]
    X_test = X[X.index >= start_test] y_train =
    y[y.index < start_test] y_test = y[y.index >=
    start_test] . .
```

Esto puede ser reemplazado con el métodotren_prueba_dividirde Scikit-Learn en eltren_prueba_split.py expediente. Para completar, el __ completoprincipal__método se proporciona a continuación:

```
# tren_prueba_split.py

si__nombre__ == "__principal__":
    # Crear una serie rezagada del índice bursátil estadounidense S&P500 snpret
    = create_lagged_series(
        " ĜSPC", datetime.datetime(2001,1,10),
        datetime.datetime(2005,12,31), lags=5
    )

    # Use los dos días anteriores de devoluciones como predictor
    # valores, con dirección como respuesta X =
    snpret[["Lag1","Lag2"]] y = snpret["Dirección"]


    # División de entrenamiento/prueba
    X_tren, X_prueba, y_tren, y_prueba = tren_prueba_dividir(
        X, y, tamaño_prueba=0.8, estado_aleatorio=42
    )

    # Crear los modelos (parametrizados) impresión
    ("Tasas de aciertos/Matrices de confusión:\n")
    modelos= [("LR", Regresión Logística()),
                ("LDA", LDA()),
                ("QDA", QDA()),
                ("LSVC", LinearSVC()),
                ("RSVM", SVC(
                    C=1000000.0, cache_size=200, class_weight=Ninguno, coef0=0.0,
                    grado=3, gamma=0.0001, kernel='rbf', max_iter=-1,
                    probabilidad=Falso, random_state=Ninguno,
```

```
                    reducción = Verdadero, tol = 0.001, detallado = Falso)
                ) ,
                ("RF", ClasificadorBosqueAleatorio(
                    n_estimators=1000, criterio='gini', max_ depth=Ninguno,
                    min_samples_split=2, min_samples_leaf=1,
                    max_features='auto', bootstrap=True, oob_score=False,
                    n_jobs=1, random_state=Ninguno, detallado=0)

                ) ]

    # Iterar a través de los modelos por
    metroen modelos:

        # Entrena a cada uno de los modelos en el set de
        entrenamiento m[1].fit(tren_X, tren_y)

        # Hacer una serie de predicciones en el conjunto de
        prueba pred = m[1].predecir(X_prueba)

        # Muestra la tasa de aciertos y la matriz de confusión para cada modelo
        impresión("%s:\n%0.3f" % (m[0], m[1].score(X_test, y_test))) impresión("%s\n" %
        confusion_matrix(pred, y_test))
```

Tenga en cuenta que hemos elegido la proporción del conjunto de entrenamiento para que sea el 80 % de los datos, dejando los datos de prueba con solo el 20 %. Además hemos especificado unestado_aleatorioaleatorizar el muestreo dentro de la selección de datos. Esto significa que los datos no se dividen secuencialmente cronológicamente, sino que se muestrean aleatoriamente.

Los resultados de la validación cruzada en el modelo son los siguientes (probablemente aparecerá ligeramente diferente debido a la naturaleza del procedimiento de ajuste):

```
Tasas de acierto/matrices de confusión:

LR:
0.511
[[ 70  70]
 [419 441]]

LDA:
0.513
[[ 69  67]
 [420 444]]

QDA:
0.503
[[ 83  91]
 [406 420]]

LSVC:
0.513
[[ 69  67]
 [420 444]]

RSVM:
0.506
[[ 8  13]
 [481 498]]
```

```
FR:
0.490
[[200 221]
 [289 290]]
```

Se puede observar que las tasas de acierto son sustancialmente inferiores a las encontradas en el capítulo de previsión antes mencionado. En consecuencia, es probable que podamos concluir que la elección particular de la división de entrenamiento/prueba conduce a una visión demasiado optimista de la capacidad predictiva del clasificador.

El siguiente paso es aumentar la cantidad de veces que se realiza una validación cruzada para minimizar cualquier posible sobreajuste. Para ello utilizaremos la validación cruzada k-fold.

Validación cruzada de K-Fold

En lugar de dividir el conjunto en un solo conjunto de prueba y entrenamiento, podemos usar la validación cruzada k-fold para*al azar*dividir el conjunto en*k*submuestras de igual tamaño. Para cada iteración (de las cuales hay*k*), uno de los*k*submuestras se conserva como un conjunto de prueba, mientras que el resto *k*−1las submuestras juntas forman un conjunto de entrenamiento. Luego se entrena un modelo estadístico en cada uno de los *k*pliegues y su desempeño evaluado en su específico*k*-ésimo conjunto de prueba.

El propósito de esto es combinar los resultados de cada modelo en un*ensamblar*mediante el promedio de los resultados de la predicción (o de otro modo) para producir una única predicción. El principal beneficio de usar la validación cruzada k-fold es que cada predictor dentro del conjunto de datos original se usa tanto para entrenamiento como para prueba solo una vez.

Esto motiva una pregunta sobre cómo elegir*k*, que ahora es otro parámetro! En general, *k* =10pero también se puede realizar otro análisis para elegir un valor óptimo de*k*.

Ahora haremos uso de lavalidación cruzadamódulo de Scikit-Learn para obtener elKDoblar objeto de validación cruzada k-fold. Creamos un nuevo archivo llamadok_fold_cross_val.py,que es una copia detren_prueba_split.pyy modifique las importaciones agregando la siguiente línea:

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# k_fold_cross_val.py

de__futuro__importarimprimir_funcion

importarfecha y hora

importarpandas como pd
importaraprender
deaprenderimportarvalidación cruzada desklearn.metrics
importarmatriz de confusión desklearn.svmimportarCVS


decrear_lag_serieimportarcrear_lag_serie
```

Entonces necesitamos hacer cambios en __principal__función quitando eltren_prueba_dividir método y reemplazándolo con una instancia deKDoblar.Toma cinco parámetros.

El primero es la duración del conjunto de datos, que en este caso es de 1250 días. El segundo valor es*k* que representa el número de pliegues, que en este caso es 10. El tercer valor esíndices,que he puesto aFalso.Esto significa que los valores de índice reales se utilizan para las matrices devueltas por el iterador. El cuarto y el quinto se utilizan para aleatorizar el orden de las muestras.

como antes enprevisión.pyytren_prueba_split.pyobtenemos la serie rezagada del S&P500. Luego creamos un conjunto de vectores de predictores (*X*) y respuestas (*y*). Entonces utilizamos elKDoblarobjeto e iterar sobre él. Durante cada iteración, creamos los conjuntos de entrenamiento y prueba para cada uno de los*X*y*y*vectores Luego, estos se alimentan a una máquina de vectores de soporte radial con parámetros idénticos a los archivos antes mencionados y el modelo se ajusta.

Finalmente, se genera la tasa de aciertos y la matriz de confusión para cada instancia de SVM.

```
# k_fold_cross_val.py

si__nombre__ == "__principal__":
    # Crear una serie rezagada del índice bursátil estadounidense S&P500 snpret
    = create_lagged_series(
        " ĜSPC", datetime.datetime(2001,1,10),
        datetime.datetime(2005,12,31), lags=5
    )

    # Use los dos días anteriores de devoluciones como predictor
    # valores, con dirección como respuesta X =
    snpret[["Lag1","Lag2"]] y = snpret["Dirección"]


    # Crear un objeto de validación cruzada k-fold kf =
    validación_cruzada.KFold(
        len(snpret), n_folds=10, indices=Falso, shuffle=True,
        random_state=42
    )

    # Use el objeto kf para crear arreglos de índices que
    # indicar qué elementos se han retenido para el entrenamiento
    # y qué elementos se han retenido para la prueba
    # para cada iteración del elemento k por
    índice_tren, índice_pruebaenkf:
        X_tren = X.ix[X.índice[índice_tren]] X_prueba =
        X.ix[X.índice[índice_prueba]] y_tren =
        y.ix[y.índice[índice_tren]] y_prueba =
        y.ix[y.índice[índice_prueba] ]]

        # En este caso solo use el
        # Máquina de vectores de soporte radial (SVM)
        impresión("Tasa de aciertos/matriz de confusión:")
        modelo = SVC(
            C=1000000.0, cache_size=200, class_weight=Ninguno, coef0=0.0,
            grado=3, gamma=0.0001, kernel='rbf', max_iter=-1,
            probabilidad=Falso, random_state=Ninguno, reducción=Verdadero,
            tol=0.001 , detallado=Falso
        )

        # Entrenar el modelo en los datos de entrenamiento
        retenidos modelo.fit(tren_X, tren_y)

        # Hacer una serie de predicciones en el conjunto de
        prueba pred = modelo.predecir(X_test)

        # Muestra la tasa de aciertos y la matriz de confusión para cada modelo
        impresión("%0.3f" % modelo.puntuación(X_test, y_test)) impresión("%s\n" %
        confusion_matrix(pred, y_test))
```

La salida del código es la siguiente:

```
Tasa de aciertos/matriz de confusión:
0,528
[[11 10]
 [49 55]]
```

```
Tasa de aciertos/matriz de confusión:
0,400
[[ 2 5]
 [70 48]]

Tasa de aciertos/matriz de confusión:
0,528
[[ 8 8]
 [51 58]]

Tasa de aciertos/matriz de confusión:
0,536
[[ 6 3]
 [55 61]]

Tasa de aciertos/matriz de confusión:
0,512
[[ 7 5]
 [56 57]]

Tasa de aciertos/matriz de confusión:
0,480
[[11 11]
 [54 49]]

Tasa de aciertos/matriz de confusión:
0,608
[[12 13]
 [36 64]]

Tasa de aciertos/matriz de confusión:
0,440
[[ 8 17]
 [53 47]]

Tasa de aciertos/matriz de confusión:
0,560
[[10 9]
 [46 60]]

Tasa de aciertos/matriz de confusión:
0,528
[[ 9 11]
 [48 57]]
```

Está claro que la tasa de aciertos y las matrices de confusión varían drásticamente entre los distintos pliegues. Esto es indicativo de que el modelo es propenso a sobreajustarse en este conjunto de datos en particular. Un remedio para esto es usar una cantidad significativamente mayor de datos, ya sea con una frecuencia más alta o durante más tiempo.

Para utilizar este modelo en una estrategia comercial, sería necesario combinar cada uno de estos clasificadores entrenados individualmente (es decir, cada uno de los*k*objetos) en un promedio de conjunto y luego use ese modelo combinado para la clasificación dentro de la estrategia.

Tenga en cuenta que técnicamente no es apropiado utilizar técnicas simples de validación cruzada en datos ordenados temporalmente (es decir, series temporales). Existen mecanismos más sofisticados para hacer frente a la autocorrelación de esta manera, pero quería resaltar el enfoque, por lo que hemos utilizado datos de series temporales para simplificar.

16.2.2 Búsqueda en cuadrícula

Hasta ahora hemos visto que la validación cruzada de k-fold nos ayuda a evitar el sobreajuste en los datos al realizar la validación en cada elemento de la muestra. Ahora dirigimos nuestra atención a la optimización de la*hiperparámetros* de un modelo estadístico particular. Dichos parámetros son aquellos que no se aprenden directamente mediante el procedimiento de estimación del modelo. Por ejemplo,*Cy*ypara una máquina de vectores de soporte. En esencia, son los parámetros que necesitamos especificar al llamar a la inicialización de cada modelo estadístico. Para este procedimiento utilizaremos un proceso conocido como*búsqueda de cuadrícula*.

La idea básica es tomar un rango de parámetros y evaluar el desempeño del modelo estadístico en cada elemento de parámetro dentro del rango. Para lograr esto en Scikit-Learn podemos crear unCuadrícula de parámetros.Dicho objeto producirá una lista de diccionarios de Python, cada uno de los cuales contiene una combinación de parámetros que se incluirán en un modelo estadístico.

A continuación se muestra un fragmento de código de ejemplo que produce una cuadrícula de parámetros, para parámetros relacionados con una máquina de vectores de soporte:

```
> > > desklearn.grid_searchimportarCuadrícula de parámetros
> > > param_grid = {'C': [1, 10, 100, 1000], 'gamma': [0.001, 0.0001]}
> > > lista(ParameterGrid(param_grid))

[{'C': 1, 'gamma': 0,001}, {'C': 1, 'gamma':
  0,0001}, {'C': 10, 'gamma': 0,001}, {'C':
  10, 'gamma': 0.0001}, {'C': 100,
  'gamma': 0.001}, {'C': 100, 'gamma':
  0.0001}, {'C': 1000, 'gamma': 0.001},
  { 'C': 1000, 'gamma': 0.0001}]
```

Ahora que tenemos un medio adecuado para generar unCuadrícula de parámetrosnecesitamos introducir esto en un modelo estadístico de forma iterativa para buscar una puntuación de rendimiento óptima. En este caso vamos a buscar maximizar la tasa de acierto del clasificador.

losGridSearchCVEl mecanismo de Scikit-Learn nos permite realizar la búsqueda de cuadrícula real. De hecho, nos permite realizar no solo una búsqueda de cuadrícula estándar, sino también un esquema de validación cruzada al mismo tiempo.

Ahora vamos a crear un nuevo archivo,grid_search.py,que una vez más usacreate_lagged_series.py y una máquina de vectores de soporte para realizar una búsqueda de cuadrícula de hiperparámetros con validación cruzada. Para ello debemos importar las bibliotecas correctas:

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# grid_search.py

de__futuro__importarimprimir_funcion

importarfecha y hora

importaraprender
deaprenderimportarvalidación cruzada
desklearn.cross_validationimportartren_prueba_dividir de
sklearn.grid_searchimportarGridSearchCV desklearn.metricsimportar
informe_clasificación desklearn.svmimportarCVS


decrear_lag_serieimportarcrear_lag_serie
```

como antes conk_fold_cross_val.pycreamos una serie retrasada y luego usamos los dos días anteriores de retornos como predictores. Inicialmente creamos una división de entrenamiento/prueba tal que el 50% de los

los datos se pueden usar para capacitación y validación cruzada, mientras que los datos restantes se pueden "retener" para la evaluación.

Posteriormente creamos elparámetros_sintonizadoslist, que contiene un solo diccionario que denota los parámetros que deseamos probar. Esto creará un*producto cartesiano*de todas las listas de parámetros, es decir, una lista de pares de todas las combinaciones de parámetros posibles.

Una vez creada la lista de parámetros la pasamos alGridSearchCVclass, junto con el tipo de clasificador que nos interesa (a saber, una máquina de vectores de soporte radial), con una validación cruzada de k-fold*k*-valor de 10.

Finalmente, entrenamos el modelo y generamos el mejor estimador y sus puntajes de tasa de acierto asociados. De esta manera, no solo hemos optimizado los parámetros del modelo a través de la validación cruzada, sino que también hemos optimizado los hiperparámetros del modelo a través de una búsqueda de cuadrícula parametrizada, ¡todo en una clase! Tal concisión del código permite una experimentación significativa sin empantanarse por una "disputa de datos" excesiva.

```
si__nombre__ == "__principal__":
    # Crear una serie rezagada del índice bursátil estadounidense S&P500 snpret
    = create_lagged_series(
        " ĜSPC", datetime.datetime(2001,1,10),
        datetime.datetime(2005,12,31), lags=5
    )

    # Use los dos días anteriores de devoluciones como predictor
    # valores, con dirección como respuesta X =
    snpret[["Lag1","Lag2"]] y = snpret["Dirección"]


    # División de entrenamiento/prueba
    X_tren, X_prueba, y_tren, y_prueba = tren_prueba_dividir(
        X, y, tamaño_prueba=0.5, estado_aleatorio=42
    )

    # Establecer los parámetros por validación cruzada
    parámetros_sintonizados = [
        {'núcleo': ['rbf'], 'gamma': [1e-3, 1e-4], 'C': [1, 10, 100, 1000]}
    ]

    # Realice la búsqueda de cuadrícula en los parámetros sintonizados
    modelo = GridSearchCV(SVC(C=1), tuned_parameters, cv=10)
    modelo.fit(X_tren, y_tren)

    impresión("Parámetros optimizados encontrados en el conjunto de entrenamiento:")
    impresión(modelo.mejor_estimador_, "\n")

    impresión("Puntuaciones de cuadrícula calculadas en el conjunto de entrenamiento:") por
    parámetros, puntuación_media, puntuacionesenmodelo.grid_puntuaciones_:
        impresión("%0.3f para %r" % (puntuación_media, parámetros))
```

El resultado del procedimiento de validación cruzada de búsqueda de cuadrícula es el siguiente:

```
Parámetros optimizados encontrados en el conjunto de entrenamiento:
SVC(C=1, cache_size=200, class_weight=Ninguno, coef0=0.0, grado=3, gamma=0.001,
  kernel='rbf', max_iter=-1, probabilidad=Falso, estado_aleatorio=Ninguno,
  reducción=Verdadero, tol=0.001, detallado=Falso)

Puntuaciones de cuadrícula calculadas en el conjunto de entrenamiento:
0.541por{'núcleo': 'rbf', 'C': 1, 'gamma': 0.001} 0.541por{'núcleo': 'rbf',
'C': 1, 'gamma': 0.0001} 0.541por{'núcleo': 'rbf', 'C': 10, 'gamma': 0.001}
```

```
0.541por{'núcleo': 'rbf', 'C': 10, 'gamma': 0.0001} 0.541por{'núcleo': 'rbf',
'C': 100, 'gamma': 0.001} 0.541por{'núcleo': 'rbf', 'C': 100, 'gamma': 0.0001}
0.538por{'núcleo': 'rbf', 'C': 1000, 'gamma': 0.001} 0.541por{'núcleo': 'rbf',
'C': 1000, 'gamma': 0.0001}
```

Como podemos verγ=0.001yC=1proporciona la mejor tasa de aciertos, en el conjunto de validación, para esta máquina de vector de soporte de kernel radial en particular. Este modelo ahora podría formar la base de una estrategia comercial basada en pronósticos, como hemos demostrado anteriormente en el capítulo anterior.

16.3 Estrategias de optimización

Hasta este punto nos hemos concentrado en*selección de modelo*y optimizar el modelo estadístico subyacente que (podría) formar la base de una estrategia comercial. Sin embargo, un modelo predictivo y una estrategia algorítmica rentable y funcional son dos entidades diferentes. Ahora centramos nuestra atención en optimizar los parámetros que tienen un efecto directo en las métricas de rentabilidad y riesgo.

Para lograr esto, vamos a hacer uso del software de backtesting basado en eventos que se describió en un capítulo anterior. Consideraremos una estrategia particular que tiene tres parámetros asociados y buscaremos a través del espacio formado por el producto cartesiano de parámetros, utilizando un mecanismo de búsqueda en cuadrícula. Luego intentaremos maximizar métricas particulares, como la relación de Sharpe, o minimizar otras, como la reducción máxima.

16.3.1 Pares revertidos a la media intradía

La estrategia que nos interesa en este capítulo es la "Operación de pares de acciones con reversión a la media intradía" utilizando las acciones de energía AREX y WLL. Contiene tres parámetros que somos capaces de optimizar: el período retrospectivo de la regresión lineal, el umbral de entrada de la puntuación z de los residuos y el umbral de salida de la puntuación z de los residuos.

Consideraremos un rango de valores para cada parámetro y luego calcularemos una prueba retrospectiva para la estrategia en cada uno de estos rangos, generando el rendimiento total, la relación de Sharpe y las características de reducción de cada simulación, en un archivo CSV para cada conjunto de parámetros. Esto nos permitirá determinar un Sharpe optimizado o una reducción máxima minimizada para nuestra estrategia comercial.

16.3.2 Ajuste de parámetros

Dado que el software de backtesting basado en eventos consume bastante CPU, restringiremos el rango de parámetros a tres valores por parámetro. Esto nos dará un total de3_3=27simulaciones separadas para llevar a cabo. Los rangos de parámetros se enumeran a continuación:

- Ventana retrospectiva OLS -$w_{yo} \in \{50,100,200\}$
- Umbral de entrada de puntuación Z -$z_h \in \{2.0,3.0,4.0\}$
- Umbral de salida de puntuación Z -$z_{yo} \in \{0.5,1.0,1.5\}$

Para realizar el conjunto de simulaciones se calculará un producto cartesiano de los tres rangos y luego se realizará la simulación para cada combinación de parámetros.

La primera tarea es modificar elintraday_mr.py fipara incluir elproductométodo de la itertoolsbiblioteca:

```
# intraday_mr.py

. .
deitertoolsimportarproducto . .
```

Entonces podemos modificar el __principal__método para incluir la generación de una lista de parámetros para los tres parámetros discutidos anteriormente.

La primera tarea es crear los rangos de parámetros reales para la ventana retrospectiva de OLS, el umbral de entrada de zscore y el umbral de salida de zscore. Cada uno de estos tiene tres variaciones separadas que conducen a un total de 27 simulaciones.

Una vez creados los rangos,itertools.productoEl método se usa para crear un producto cartesiano de todas las variaciones, que luego se alimenta a una lista de diccionarios para garantizar que los argumentos de palabras clave correctos se pasen alEstrategiaobjeto.

Finalmente, se instancia el backtest con elstrat_params_listformando el argumento de palabra clave final:

```
si__nombre__ == "__principal__":
    csv_dir = '/ruta/a/su/csv/archivo' symbol_list =    # ¡CAMBIA ESTO!
    ['AREX', 'WLL'] initial_capital = 100000.0

    latido del corazón = 0.0
    fecha_inicio = fechahora.fechahora(2007, 11, 8, 10, 41, 0)

    # Crear la grilla de parámetros de la estrategia
    # utilizando el generador de productos cartesianos de
    itertools strat_lookback = [50, 100, 200] strat_z_entry = [2.0, 3.0,
    4.0] strat_z_exit = [0.5, 1.0, 1.5] strat_params_list = list(product(


        strat_lookback, strat_z_entry, strat_z_exit
    ) )

    # Crear una lista de diccionarios con la correcta
    # pares de palabra clave/valor para los parámetros de la
    estrategia strat_params_dict_list = [
        dict(ols_window=sp[0], zscore_high=sp[1], zscore_low=sp[2]) porspen
        strat_params_list
    ]

    # Llevar a cabo el conjunto de backtests para todas las combinaciones de parámetros prueba
    inversa = prueba inversa (
        csv_dir, lista_símbolos, capital_inicial, latido, fecha_inicio, HistoricCSVDataHandlerHFT,
        SimulatedExecutionHandler, PortfolioHFT, IntradayOLSMRStrategy,

        strat_params_list=strat_params_dict_list
    )
    backtest.simular_trading()
```

El siguiente paso es modificar elprueba retrospectivaobjeto enbacktest.pypara poder manejar múltiples conjuntos de parámetros. Necesitamos modificar el _generar_instancias_comercialesmétodo para tener un argumento que represente el parámetro particular establecido en la creación de un nuevoEstrategiaobjeto:

```
# backtest.py

. .
    definitivamente_generar_trading_instances(yo, estrategia_params_dict):
        """
        Genera los objetos de instancia comercial a partir de
        sus tipos de clase.
        """
        impresión("Creación de DataHandler, Estrategia, Portafolio y ExecutionHandler impresión     por")
        ("lista de parámetros de estrategia: %s..." %strategy_params_dict) self.data_handler =
        self.data_handler_cls(
            self.events, self.csv_dir, self.symbol_list, self.header_strings
        )
```

```
            self.estrategia = self.estrategia_cls(
                self.data_handler, self.events, **strategy_params_dict
            )
            self.cartera = self.cartera_cls(
                self.data_handler, self.events, self.start_date, self.num_strats,
                self.periods, self.initial_capital
            )
            self.execution_handler = self.execution_handler_cls(self.events)
. .
```

Este método ahora se llama dentro de un ciclo de lista de parámetros de estrategia, en lugar de en la construcción del prueba retrospectivaobjeto. Si bien puede parecer un desperdicio recrear todos los controladores de datos, las colas de eventos y los objetos de cartera para cada conjunto de parámetros, garantiza que todos los iteradores se hayan restablecido y que realmente estemos comenzando con una "borrón y cuenta nueva" para cada simulación.

La siguiente tarea es modificar elsimular_tradingmétodo para recorrer todas las variantes de los parámetros de la estrategia. El método crea un archivo CSV de salida que se usa para almacenar combinaciones de parámetros y sus métricas de rendimiento particulares. Esto nos permitirá más tarde trazar las características de rendimiento a través de los parámetros.

El método recorre todos los parámetros de la estrategia y genera una nueva instancia comercial en cada simulación. Luego se ejecuta el backtest y se calculan las estadísticas. Estos se almacenan y se envían al archivo CSV. Una vez que finaliza la simulación, el archivo de salida se cierra:

```
# backtest.py

. .
    definitivamentesimular_trading(auto):
        """
        Simula el backtest y el rendimiento de la cartera de salida. """

        fuera = abrir("salida/opt.csv", "w")

        spl = len(self.strat_params_list)
        poryo, spenenumerar (self.strat_params_list):
            impresión("Estrategia %s de %s..." % (i+1, spl))
            self._generate_trading_instances(sp)
            self._run_backtest()
            estadísticas = self._output_rendimiento()
            pprint.pprint(estadísticas)

            tot_ret = float(stats[0][1].replace("%","")) cagr = float(stats[1]
            [1].replace("%","")) sharpe = float(stats[ 2][1])

            max_dd = float(estadísticas[3][1].replace("%","")) dd_dur =
            int(estadísticas[4][1])

            fuera.escribir(
                "%s,%s,%s,%s,%s,%s,%s,%s\n" % (
                    sp["ols_window"], sp["zscore_high"], sp["zscore_low"], tot_ret, cagr,
                    sharpe, max_dd, dd_dur
                )
            )

        fuera.cerrar()
```

¡En mi sistema de escritorio, este proceso toma algún tiempo! 27 simulaciones de parámetros en más de 600 000 puntos de datos por simulación tardan alrededor de 3 horas. El backtester no se ha paralelizado en esta etapa, por lo que la ejecución simultánea de trabajos de simulación haría que el proceso fuera mucho más rápido. El resultado del estudio de parámetros actual se muestra a continuación. Las columnas están dadas por

OLS Lookback, ZScore High, ZScore Low, Total Return (%), CAGR (%), Sharpe, Max DD (%), DD Duration (minutos):

```
50,2.0,0.5,213.96,20.19,1.63,42.55,255568
50,2.0,1.0,264.9,23.13,2.18,27.83,160319
50,2.0,1.5,167.71,17.15,1.63,60.52,293207
50,3.0,0.5,298.64,24.9,2.82,14.06,35127
50,3.0,1.0,324.0,26.14,3.61,9.81,33533
50,3.0,1.5,294.91,24.71,3.71,8.04,31231
50,4.0,0.5,212.46,20.1,2.93,8.49,23920
50,4.0,1.0,222.94,20.74,3.5,8.21,28167
50,4.0,1.5,215.08,20.26,3.66,8.25,22462
100,2.0,0.5,378.56,28.62,2.54,22.72,74027
100,2.0,1.0,374.23,28.43,3.0,15.71,89118
100,2.0,1.5,317.53,25.83,2.93,14.56,80624
100,3.0,0.5,320.1,25.95,3.06,13.35,66012
100,3.0,1.0,307.18,25.32,3.2,11.57,32185
100,3.0,1.5,306.13,25.27,3.52,7.63,33930
100,4.0,0.5,231.48,21.25,2.82,7.44,29160
100,4.0,1.0,227.54,21.01,3.11,7.7,15400
100,4.0,1.5,224.43,20.83,3.33,7.73,18584
200,2.0,0.5,461.5,31.97,2.98,19.25,31024
200,2.0,1.0,461.99,31.99,3.64,10.53,64793
200,2.0,1.5,399.75,29.52,3.57,10.74,33463
200,3.0,0.5,333.36,26.58,3.07,19.24,56569
200,3.0,1.0,325.96,26.23,3.29,10.78,35045
200,3.0,1.5,284.12,24.15,3.21,9.87,34294
200,4.0,0.5,245.61,22.06,2.9,12.52,51143
200,4.0,1.0,223.63,20.78,2.93,9.61,40075
200,4.0,1.5,203.6,19.55,2.96,7.16,40078
```

Podemos ver que para este estudio en particular los valores de los parámetros de w_{yo}=50, z_h=3.0y z_{yo}=1.5 proporcionar la mejor proporción de Sharpe en S=3.71.Para esta relación de Sharpe tenemos un rendimiento total de294.91%y una reducción máxima de8.04%.La mejor rentabilidad total de461.99%,aunque con una reducción máxima de10.53%viene dada por el conjunto de parámetros de w_{yo}=200, z_h=2.0y z_{yo}=1.0.

16.3.3 Visualización

Como tarea final en la optimización de la estrategia, ahora vamos a visualizar las características de rendimiento del backtester utilizando Matplotlib, que es un paso extremadamente útil cuando se realiza una investigación de estrategia inicial. Desafortunadamente, estamos en una situación en la que tenemos un problema tridimensional y, por lo tanto, ¡la visualización del rendimiento no es sencilla! Sin embargo, hay algunos remedios parciales a la situación.

En primer lugar, podríamos fijar el valor de un parámetro y tomar una "porción de parámetro" a través del resto del "cubo de datos". Por ejemplo, podríamos fijar la ventana retrospectiva en 100 y luego ver a quién afecta la variación en los umbrales de entrada y salida del puntaje z al índice de Sharpe o la reducción máxima.

Para lograr esto utilizaremos Matplotlib. Leeremos el CSV de salida y remodelaremos los datos para que podamos visualizar los resultados.

Mapa de calor de Sharpe/Drawdown

Arreglaremos el período retrospectivo de w_{yo}=100y luego generar un 3×3cuadrícula y "mapa de calor" de la relación de Sharpe y reducción máxima para la variación en los umbrales de puntuación z.

En el siguiente código importamos el archivo CSV de salida. La primera tarea es filtrar los períodos retrospectivos que no son de interés ($w_{yo} \in \{50,200\}$). Luego remodelamos los datos de rendimiento restantes en dos 3×3matrices. El primero representa la relación de Sharpe para cada combinación de umbral de puntuación z, mientras que el segundo representa la reducción máxima.

Aquí está el código para crear el mapa de calor de Sharpe Ratio. Primero importamos Matplotlib y NumPy. Entonces definimos una función llamadacrear_matriz_de_datosque remodela los datos de Sharpe Ratio en un3×3cuadrícula. Dentro de __principal__función, abrimos el archivo CSV (¡asegúrese de cambiar la ruta en su sistema!) y excluimos cualquier línea que no haga referencia a un período retrospectivo de 100.

Luego creamos un mapa de calor sombreado en azul y aplicamos las etiquetas de fila/columna correctas utilizando los umbrales de puntuación z. Posteriormente, colocamos el valor real de la relación de Sharpe en el mapa de calor. Finalmente, establecemos las marcas, las etiquetas, el título y luego trazamos el mapa de calor:

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# plot_sharp.py

importarmatplotlib.pyplot como plt
importarnumpy como np


definitivamentecrear_matriz_datos(csv_ref, col_index):
    datos = np.ceros((3, 3)) porien
    rango (0, 3):
        porjenrango (0, 3):
            datos[i][j] = float(csv_ref[i*3+j][col_index]) devolverdatos



si__nombre__ == "__principal__":
    # Abra el archivo CSV y obtenga solo las líneas
    # con un valor retrospectivo de 100
    csv_file = open("/ruta/a/opt.csv", "r").readlines() csv_ref = [

        c.strip().split(",")
        porCenarchivo_csvsic[:3] == "100"
    ]
    datos = crear_matriz_datos(csv_ref, 5)

    higo, hacha = plt.subplots()
    mapa de calor = ax.pcolor(datos, cmap=plt.cm.Blues)
    row_labels = [0.5, 1.0, 1.5]
    etiquetas_columna = [2.0, 3.0, 4.0]

    poryenrango (datos.forma[0]):
        porXenrango (datos.forma[1]):
            plt.text(x + 0.5, y + 0.5, '%.2f' % datos[y, x],
                     alineación horizontal = 'centro',
                     alineación vertical = 'centro',)


    plt.colorbar(mapa de calor)

    ax.set_xticks(np.arange(data.shape[0])+0.5, minor=False)
    ax.set_yticks(np.arange(data.shape[1])+0.5, minor=False)
    ax.set_xticklabels(row_labels, menor = falso)
    ax.set_yticklabels(column_labels, menor=Falso)

    plt.suptitle('Mapa de calor de relación de Sharpe', tamaño de fuente=18)
    plt.xlabel('Umbral de salida de puntuación Z', tamaño de fuente=14)
```

```
    plt.ylabel('Umbral de entrada de puntuación Z', tamaño de fuente=14)
    plt.show()
```

La gráfica para la reducción máxima es casi idéntica con la excepción de que usamos un mapa de calor sombreado en rojo y modificamos el índice de la columna en elcrear_matriz_de_datosfunción para utilizar los datos de porcentaje máximo de reducción.

```
#!/usr/bin/python
# - * - codificación: utf-8 -*-

# plot_drawdown.py

importarmatplotlib.pyplot como plt
importarnumpy como np


definitivamentecrear_matriz_datos(csv_ref, col_index):
    datos = np.ceros((3, 3)) porien
    rango (0, 3):
        porjenrango (0, 3):
            datos[i][j] = float(csv_ref[i*3+j][col_index]) devolverdatos



si__nombre__ == "__principal__":
    # Abra el archivo CSV y obtenga solo las líneas
    # con un valor retrospectivo de 100
    csv_file = open("/ruta/a/opt.csv", "r").readlines() csv_ref = [

        c.strip().split(",")
        porCenarchivo_csvsic[:3] == "100"
    ]
    datos = crear_matriz_datos(csv_ref, 6)

    higo, hacha = plt.subplots()
    mapa de calor = ax.pcolor(data, cmap=plt.cm.Reds)
    row_labels = [0.5, 1.0, 1.5]
    etiquetas_columna = [2.0, 3.0, 4.0]

    poryenrango (datos.forma[0]):
        porXenrango (datos.forma[1]):
            plt.text(x + 0.5, y + 0.5, '%.2f%%' % datos[y, x],
                     alineación horizontal = 'centro',
                     alineación vertical = 'centro',)


    plt.colorbar(mapa de calor)

    ax.set_xticks(np.arange(data.shape[0])+0.5, minor=False)
    ax.set_yticks(np.arange(data.shape[1])+0.5, minor=False)
    ax.set_xticklabels(row_labels, menor = falso)
    ax.set_yticklabels(column_labels, menor=Falso)

    plt.suptitle('Mapa de calor de reducción máxima', tamaño de fuente=18)
    plt.xlabel('Umbral de salida de puntuación Z', tamaño de fuente=14)
    plt.ylabel('Umbral de entrada de puntuación Z', tamaño de fuente=14)
    plt.show( )
```

Los mapas de calor producidos a partir de los fragmentos anteriores se muestran en la Fig. 16.3.3 y la Fig. 16.3.3:

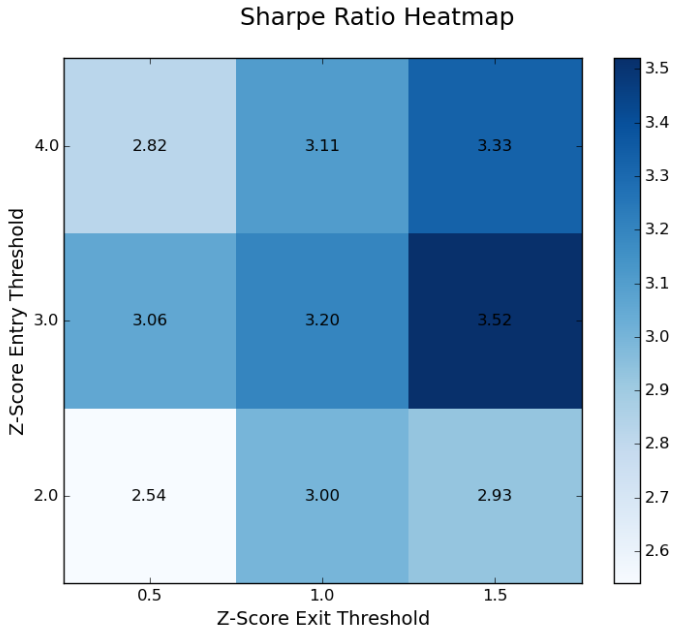

Figura 16.1: Mapa de calor de la relación de Sharpe para umbrales de entrada/salida de puntuación z

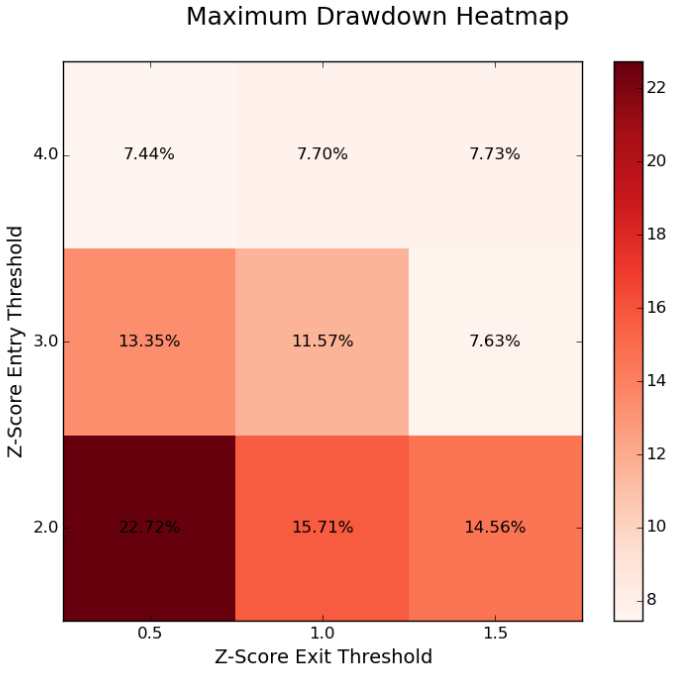

Figura 16.2: Mapa de calor de reducción máxima para umbrales de entrada/salida de puntuación z

Aw_{yo}=100las diferencias entre los índices de Sharpe más pequeños y más grandes, así como las reducciones máximas más pequeñas y más grandes, son evidentes. La relación de Sharpe está optimizada para umbrales de entrada y salida más grandes, mientras que la reducción se minimiza en la misma región. La relación de Sharpe y la reducción máxima están en su peor momento cuando los umbrales de entrada y salida son bajos.