Chapter 31

Market Regime Detection with Hidden Markov Models using QSTrader

In the previous chapter on Hidden Markov Models it was shown how their application to index returns data could be used as a mechanism for discovering latent "market regimes". The returns of the S&P500 were analysed using the R statistical programming environment. It was seen that periods of differing volatility were detected, using both two-state and three-state models.

In this chapter the Hidden Markov Model will be utilised within the QSTrader framework as a risk-managing market regime filter. It will disallow trades when higher volatility regimes are predicted. The hope is that by doing so it will eliminate unprofitable trades and possibly remove volatility from the strategy, thus increasing its Sharpe ratio.

In order to achieve this some small code modifications to QSTrader were necessary, which are part of the current version as of the release date of this book.

The market regime overlay will be paired with a simplistic short-term trend-following strategy, based on simple moving average crossover rules. The strategy itself is relatively unimportant for the purposes of this chapter, as the majority of the discussion will focus on implementing the risk management logic.

It should be noted that QSTrader is written in Python, while the previous implementation of the Hidden Markov Model was carried out in R. Hence for the purposes of this chapter it is necessary to utilise a Python library that already implements a Hidden Markov Model. hmmlearn is such a library and it will be used here.

31.1 Regime Detection with Hidden Markov Models

Hidden Markov Models will briefly be recapped. For a full discussion see the previous chapter in the Time Series Analysis section.

Hidden Markov Models are a type of stochastic state-space model. They assume the existence of "hidden" or "latent" states that are not directly observable. These hidden states have an influence on values which *are* observable, known as the *observations*. One of the goals of the model is to ascertain the current state from the set of known observations.

In quantitative trading this problem translates into having "hidden" or "latent" market regimes, such as changing regulatory environments, or periods of excess volatility. The observations in this case are the returns from a particular set of financial market data. The returns are indirectly influenced by the hidden market regimes. Fitting a Hidden Markov Model to the returns data allows prediction of new regime states, which can be used a risk management trading filter mechanism.

31.2 The Trading Strategy

The trading strategy for this chapter is exceedingly simple and is used because it can be well understood. The important issue is the risk management aspect, which will be given significantly more attention.

The short-term trend following strategy is of the classic moving average crossover type. The rules are simple:

- At every bar calculate the 10-day and 30-day simple moving averages (SMA)
- If the 10-day SMA exceeds the 30-day SMA and the strategy is not invested, then go long
- If the 30-day SMA exceeds the 10-day SMA and the strategy is invested, then close the position

This is not a particularly effective strategy with these parameters, especially on S&P500 index prices. It will not really achieve much in comparison to a buy-and-hold of the SPY ETF for the same period.

However, when combined with a risk management trading filter it becomes more effective due to the potential of eliminating trades occuring in highly volatile periods, where such trend-following strategies can lose money.

The risk management filter applied here works by training a Hidden Markov Model on S&P500 data from the 29th January 1993 (the earliest available data for SPY on Yahoo Finance) through to the 31st December 2004. This model is then *serialised* (via Python pickle) and utilised with a QSTrader `RiskManager` subclass.

The risk manager checks, for every trade sent, whether the current state is a low volatility or high volatility regime. If volatility is low any long trades are let through and carried out. If volatility is high any open trades are closed out upon receipt of the closing signal, while any new potential long trades are cancelled before being allowed to pass through.

This has the desired effect of eliminating trend-following trades in periods of high vol where they are likely to lose money due to incorrect identification of "trend".

The backtest of this strategy is carried out from 1st January 2005 to 31st December 2014, without retraining the Hidden Markov Model along the way. In particular this means the HMM is being used out-of-sample and not on in-sample training data.

31.3 Data

In order to carry out this strategy it is necessary to have daily OHLCV pricing data for the SPY ETF ticker for the period covered by both the HMM training and the backtest. This can be found in Table 31.3.

Ticker	Name	Period	Link
SPY	SPDR S&P 500 ETF	January 29th 1993 - 31st December 2014	Yahoo Finance

This data will need to placed in the directory specified by the QSTrader settings file if you wish to replicate the results.

31.4 Python Implementation

31.4.1 Returns Calculation with QSTrader

In order to carry out regime predictions using the Hidden Markov Model it is necessary to calculate and store the adjusted closing price returns of SPY. To date only the *prices* have been stored. The natural location to store the returns is in the `PriceHandler` subclass. However, QSTrader did not previously support this behaviour and so it has now been added as a feature.

It was a relatively simple modification involving two minor changes. The first was to add a `calc_adj_returns` boolean flag to the initialisation of the class. If this is set to `True` then the adjusted returns would be calculated and stored, otherwise they would not be. In order to minimise impact on other client code the default is set to `False`.

The second change overrides the "virtual" method `_store_event` found in the `AbstractBarPriceHandler` class with the following in the `YahooDailyCsvBarPriceHandler` subclass.

The code checks if `calc_adj_returns` is equal to `True`. It stores the previous and current adjusted closing prices, modifying them with the `PriceParser`, calculates the percentage returns and then adds them to the `adj_close_returns` list. This list is later called by the `RegimeHMMRiskManager` in order to predict the current regime state:

```
def _store_event(self, event):
    """
    Store price event for closing price and adjusted closing price
    """
    ticker = event.ticker
    # If the calc_adj_returns flag is True, then calculate
    # and store the full list of adjusted closing price
    # percentage returns in a list
    if self.calc_adj_returns:
        prev_adj_close = self.tickers[ticker][
            "adj_close"
        ] / PriceParser.PRICE_MULTIPLIER
        cur_adj_close = event.adj_close_price / PriceParser.PRICE_MULTIPLIER
        self.tickers[ticker][
            "adj_close_ret"
        ] = cur_adj_close / prev_adj_close - 1.0
        self.adj_close_returns.append(self.tickers[ticker]["adj_close_ret"])
    self.tickers[ticker]["close"] = event.close_price
    self.tickers[ticker]["adj_close"] = event.adj_close_price
```

```
    self.tickers[ticker]["timestamp"] = event.time
```

This modification is already in the latest version of QSTrader, which (as always) can be found at the Github page.

31.4.2 Regime Detection Implementation

Attention will now turn towards the implementation of the regime filter and short-term trend-following strategy that will be used to carry out the backtest.

There are four separate files required for this strategy to be carried out. The full listings of each are provided at the end of the chapter. This will allow straightforward replication of the results for those wishing to implement a similar method.

The first file encompasses the fitting of a Gaussian Hidden Markov Model to a large period of the S&P500 returns. The second file contains the logic for carrying out the short-term trend-following. The third file provides the regime filtering of trades through a risk manager object. The final file ties all of these modules together into a backtest.

Training the Hidden Markov Model

Prior to the creation of a regime detection filter it is necessary to fit the Hidden Markov Model to a set of returns data. For this the Python hmmlearn library will be used. The API is exceedingly simple, which makes it straightforward to fit and store the model for later use.

The first task is to import the necessary libraries. `warnings` is used to suppress the excessive deprecation warnings generated by Scikit-Learn, through API calls from hmmlearn. `GaussianHMM` is imported from hmmlearn forming the basis of the model. Matplotlib and Seaborn are imported to plot the in-sample hidden states, necessary for a "sanity check" on the models behaviour:

```
# regime_hmm_train.py

from __future__ import print_function

import datetime
import pickle
import warnings

from hmmlearn.hmm import GaussianHMM
from matplotlib import cm, pyplot as plt
from matplotlib.dates import YearLocator, MonthLocator
import numpy as np
import pandas as pd
import seaborn as sns
```

The `obtain_prices_df` function opens up the CSV file of the SPY data downloaded from Yahoo Finance into a Pandas DataFrame. It then calculates the percentage returns of the adjusted closing prices and truncates the ending date to the desired final training period. Calculating the percentage returns introduces `NaN` values into the DataFrame, which are then dropped in place:

```
def obtain_prices_df(csv_filepath, end_date):
    """
```

```
    Obtain the prices DataFrame from the CSV file,
    filter by the end date and calculate the
    percentage returns.
    """
    df = pd.read_csv(
        csv_filepath, header=0,
        names=[
            "Date", "Open", "High", "Low",
            "Close", "Volume", "Adj Close"
        ],
        index_col="Date", parse_dates=True
    )
    df["Returns"] = df["Adj Close"].pct_change()
    df = df[:end_date.strftime("%Y-%m-%d")]
    df.dropna(inplace=True)
    return df
```

The following function, `plot_in_sample_hidden_states`, is not strictly necessary for training purposes. It has been modified from the hmmlearn tutorial file found in the documentation.

The code takes the model along with the prices DataFrame and creates a subplot, one plot for each hidden state generated by the model. Each subplot displays the adjusted closing price masked by that particular hidden state/regime. This is useful to see if the HMM is producing "sane" states:

```
def plot_in_sample_hidden_states(hmm_model, df):
    """
    Plot the adjusted closing prices masked by
    the in-sample hidden states as a mechanism
    to understand the market regimes.
    """
    # Predict the hidden states array
    hidden_states = hmm_model.predict(rets)
    # Create the correctly formatted plot
    fig, axs = plt.subplots(
        hmm_model.n_components,
        sharex=True, sharey=True
    )
    colours = cm.rainbow(
        np.linspace(0, 1, hmm_model.n_components)
    )
    for i, (ax, colour) in enumerate(zip(axs, colours)):
        mask = hidden_states == i
        ax.plot_date(
            df.index[mask],
            df["Adj Close"][mask],
            ".", linestyle='none',
```

```
            c=colour
        )
        ax.set_title("Hidden State #%s" % i)
        ax.xaxis.set_major_locator(YearLocator())
        ax.xaxis.set_minor_locator(MonthLocator())
        ax.grid(True)
    plt.show()
```

The output of this particular function is given below:

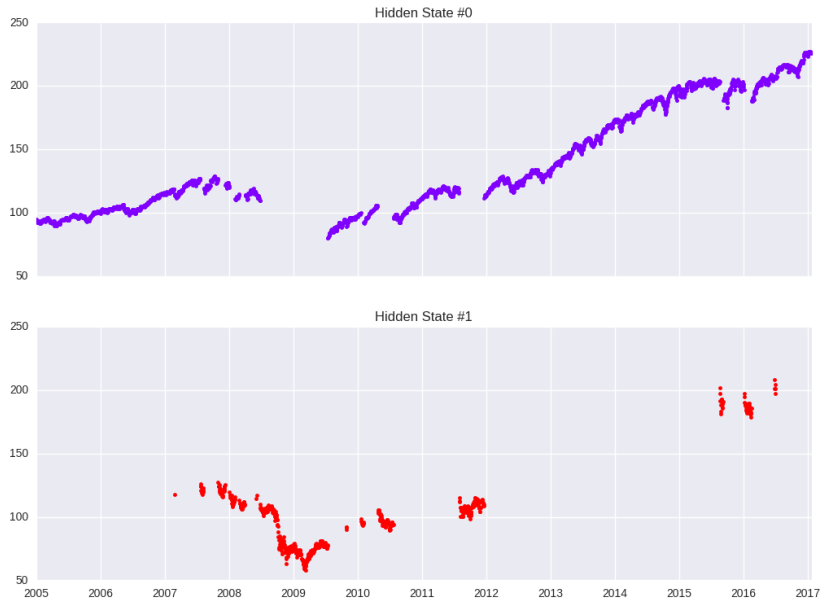

Figure 31.1:

It can be seen that the regime detection largely captures "trending" periods and highly volatile periods. In particular the majority of 2008 occurs in Hidden State #1.

This script is tied together in the `__main__` function. Firstly all warnings are ignored. Strictly speaking this is not best practice, but in this instance there are many deprecation warnings generated by Scikit-Learn that obscure the desired output of the script.

Subsequently the CSV file is opened and the `rets` variable is created using the `np.column_stack` command. This is because hmmlearn requires a matrix of series objects, despite the fact that this is a univariate model (it only acts upon the returns themselves). Using NumPy in this manner puts it into the correct format.

The `GaussianHMM` object requires specification of the number of states through the `n_components` parameter. Two states are used in this chapter, but three could also be tested easily. A full covariance matrix is used, rather than a diagonal version. The number of iterations used in the Expectation-Maximisation algorithm is given by the `n_iter` parameter.

The model is fitted and the score of the algorithm output. The hidden states masking the adjusted closing prices are plotted. Finally the model is pickled (serialised) to the `pickle_path`, ready to be used in the regime detection risk manager later in the chapter:

```
if __name__ == "__main__":
    # Hides deprecation warnings for sklearn
    warnings.filterwarnings("ignore")

    # Create the SPY dataframe from the Yahoo Finance CSV
    # and correctly format the returns for use in the HMM
    csv_filepath = "/path/to/your/data/SPY.csv"
    pickle_path = "/path/to/your/model/hmm_model_spy.pkl"
    end_date = datetime.datetime(2004, 12, 31)
    spy = obtain_prices_df(csv_filepath, end_date)
    rets = np.column_stack([spy["Returns"]])

    # Create the Gaussian Hidden markov Model and fit it
    # to the SPY returns data, outputting a score
    hmm_model = GaussianHMM(
        n_components=2, covariance_type="full", n_iter=1000
    ).fit(rets)
    print("Model Score:", hmm_model.score(rets))

    # Plot the in sample hidden states closing values
    plot_in_sample_hidden_states(hmm_model, spy)

    print("Pickling HMM model...")
    pickle.dump(hmm_model, open(pickle_path, "wb"))
    print("...HMM model pickled.")
```

Short-Term Trend Following Strategy

The next stage in the process is to create the `Strategy` class that encapsulates the short-term trend-following logic, which will ultimately be filtered by the `RiskManager` module.

As with all strategies developed within QSTrader it is necessary to import some specific classes, including the `PriceParser`, `SignalEvent` and `AbstractStrategy` base class. This is similar to many other strategies carried out in the book so the reason for these imports will not be stressed:

```
# regime_hmm_strategy.py

from __future__ import print_function

from collections import deque

import numpy as np

from qstrader.price_parser import PriceParser
from qstrader.event import (SignalEvent, EventType)
from qstrader.strategy.base import AbstractStrategy
```

The `MovingAverageCrossStrategy` subclass is actually one of the examples found within the QSTrader examples directory. However it has been replicated here for completeness. The strategy uses two double-ended queues, found in the `deque` module, to provide rolling windows on the pricing data. This is to calculate the long and short simple moving averages that form the short-term trend-following logic:

```
class MovingAverageCrossStrategy(AbstractStrategy):
    """
    Requires:
    tickers - The list of ticker symbols
    events_queue - A handle to the system events queue
    short_window - Lookback period for short moving average
    long_window - Lookback period for long moving average
    """
    def __init__(
        self, tickers,
        events_queue, base_quantity,
        short_window=10, long_window=30
    ):
        self.tickers = tickers
        self.events_queue = events_queue
        self.base_quantity = base_quantity
        self.short_window = short_window
        self.long_window = long_window
        self.bars = 0
        self.invested = False
        self.sw_bars = deque(maxlen=self.short_window)
        self.lw_bars = deque(maxlen=self.long_window)
```

All QSTrader `AbstractStrategy`-derived subclasses use a `calculate_signals` method to generate `SignalEvent` objects. The method here firstly checks whether the event is an OHLCV bar. For instance, it could be a `SentimentEvent` (as in other strategies) and thus a check is required. The prices are appended to the deques in the correct manner, thus providing rolling windows over which to perform the SMA.

If there are enough bars to perform the moving averages then they are both calculated. Once these values are present the trading rules described above are carried out. If the short window SMA exceeds the long window SMA, and the strategy is not already invested, then it generates a long position of `base_quantity` shares. If the long window SMA exceeds the short window SMA the position is closed if already invested:

```
def calculate_signals(self, event):
    # Applies SMA to first ticker
    ticker = self.tickers[0]
    if event.type == EventType.BAR and event.ticker == ticker:
        # Add latest adjusted closing price to the
        # short and long window bars
        price = event.adj_close_price / float(
```

```
            PriceParser.PRICE_MULTIPLIER
        )
        self.lw_bars.append(price)
        if self.bars > self.long_window - self.short_window:
            self.sw_bars.append(price)

        # Enough bars are present for trading
        if self.bars > self.long_window:
            # Calculate the simple moving averages
            short_sma = np.mean(self.sw_bars)
            long_sma = np.mean(self.lw_bars)
            # Trading signals based on moving average cross
            if short_sma > long_sma and not self.invested:
                print("LONG: %s" % event.time)
                signal = SignalEvent(ticker, "BOT", self.base_quantity)
                self.events_queue.put(signal)
                self.invested = True
            elif short_sma < long_sma and self.invested:
                print("SHORT: %s" % event.time)
                signal = SignalEvent(ticker, "SLD", self.base_quantity)
                self.events_queue.put(signal)
                self.invested = False
        self.bars += 1
```

Regime Detection Risk Manager

The subclassed `AbstractRiskManager` object is the first major usage of risk management applied separately to a strategy within the book. As outlined above the goal of this object is to filter the short-term trend-following trades when in an undesirable high volatility regime.

All `RiskManager` subclasses require access to an `OrderEvent` as they have the power to eliminate, modify or create orders depending upon the risk constraints of the portfolio:

```
# regime_hmm_risk_manager.py

from __future__ import print_function

import numpy as np

from qstrader.event import OrderEvent
from qstrader.price_parser import PriceParser
from qstrader.risk_manager.base import AbstractRiskManager
```

The `RegimeHMMRiskManager` simply requires access to the deserialised HMM model file. It also keeps track of whether the strategy is "invested" or not, since the `Strategy` object itself will have no knowledge of whether its signals have actually been executed:

```
class RegimeHMMRiskManager(AbstractRiskManager):
```

```
    """
    Utilises a previously fitted Hidden Markov Model
    as a regime detection mechanism. The risk manager
    ignores orders that occur during a non-desirable
    regime.

    It also accounts for the fact that a trade may
    straddle two separate regimes. If a close order
    is received in the undesirable regime, and the
    order is open, it will be closed, but no new
    orders are generated until the desirable regime
    is achieved.
    """
    def __init__(self, hmm_model):
        self.hmm_model = hmm_model
        self.invested = False
```

A helper method, `determine_regime`, uses the `price_handler` object and the `sized_order` event to obtain the full list of adjusted closing returns calculated by QSTrader (see the code in the previous section for details). It then uses the `predict` method of the `GaussianHMM` object to produce an array of predicted regime states. It takes the latest value and then uses this as the current "hidden state", or regime:

```
def determine_regime(self, price_handler, sized_order):
    """
    Determines the predicted regime by making a prediction
    on the adjusted closing returns from the price handler
    object and then taking the final entry integer as
    the "hidden regime state".
    """
    returns = np.column_stack(
        [np.array(price_handler.adj_close_returns)]
    )
    hidden_state = self.hmm_model.predict(returns)[-1]
    return hidden_state
```

The `refine_orders` method is necessary on all `AbstractRiskManager`-derived subclasses. In this instance it calls the `determine_regime` method to find the regime state. It then creates the correct `OrderEvent` object, but crucially at this stage does not return it yet:

```
def refine_orders(self, portfolio, sized_order):
    """
    Uses the Hidden Markov Model with the percentage returns
    to predict the current regime, either 0 for desirable or
    1 for undesirable. Long entry trades will only be carried
    out in regime 0, but closing trades are allowed in regime 1.
    """
    # Determine the HMM predicted regime as an integer
```

```
    # equal to 0 (desirable) or 1 (undesirable)
    price_handler = portfolio.price_handler
    regime = self.determine_regime(
        price_handler, sized_order
    )
    action = sized_order.action
    # Create the order event, irrespective of the regime.
    # It will only be returned if the correct conditions
    # are met below.
    order_event = OrderEvent(
        sized_order.ticker,
        sized_order.action,
        sized_order.quantity
    )
    ..
    ..
```

The latter half of the method is where the regime detection risk management logic is based. It consists of a conditional block that firstly checks which regime state has been identified.

If it is the low volatility state #0 it checks to see if the order is a "BOT" or "SLD" action. If it is a "BOT" (long) order then it simply returns the `OrderEvent` and keeps track of the fact that it now has a long position open. If it is "SLD" (close) action then it closes the position if one is open, otherwise it cancels the order.

If however the regime is predicted to be the high volatility state #1 then it also checks which action has occurred. It does not allow any long positions in this state. It only allows a close position to occur if a long position has previously been opened, otherwise it cancels it.

This has the effect of never generating a *new* long position when in regime #1. However, a previously open long position can be closed in regime #1.

An alternative approach might be to immediately close any open long position upon entering regime #1, although this is left as an exercise for the reader!

```
    ..
    ..
    # If in the desirable regime, let buy and sell orders
    # work as normal for a long-only trend following strategy
    if regime == 0:
        if action == "BOT":
            self.invested = True
            return [order_event]
        elif action == "SLD":
            if self.invested == True:
                self.invested = False
                return [order_event]
            else:
                return []
    # If in the undesirable regime, do not allow any buy orders
```

```
    # and only let sold/close orders through if the strategy
    # is already invested (from a previous desirable regime)
    elif regime == 1:
        if action == "BOT":
            self.invested = False
            return []
        elif action == "SLD":
            if self.invested == True:
                self.invested = False
                return [order_event]
            else:
                return []
```

This concludes the `RegimeHMMRiskManager` code. All that remains is to tie the above three scripts/modules together through a `Backtest` object. The full code for this script can be found, as with the rest of the modules, at the end of this chapter.

In `regime_hmm_backtest.py` both an `ExampleRiskManager` and the `RegimeHMMRiskManager` are imported. This allows straightforward "switching out" of risk managers across backtests to see how the results change:

```
# regime_hmm_backtest.py

..
..

from qstrader.risk_manager.example import ExampleRiskManager

..
..

from regime_hmm_strategy import MovingAverageCrossStrategy
from regime_hmm_risk_manager import RegimeHMMRiskManager
```

In the `run` function the first task is to specify the HMM model pickle path, necessary for deserialisation of the model. Subsequently the price handler is specified. Crucially the `calc_adj_returns` flag is set to true, which sets the price handler up to calculate and store the returns array.

At this stage the `MovingAverageCrossStrategy` is instantiated with a short window of 10 days, a long window of 30 days and a base quantity equal to 10,000 shares of SPY.

Finally the `hmm_model` is deserialised through `pickle` and the `risk_manager` is instantiated. The rest of the script is extremely similar to other backtests carried out in the book, so the full code will only be outlined at the end of the chapter.

It is straightforward to "switch out" risk managers by commenting the `RegimeHMMRiskManager` line, replacing it with the `ExampleRiskManager` line and then rerunning the backtest:

```
def run(config, testing, tickers, filename):
    ..
    ..
```

```
    pickle_path = "/path/to/your/model/hmm_model_spy.pkl"

    ..
    ..

    # Use the Moving Average Crossover trading strategy
    base_quantity = 10000
    strategy = MovingAverageCrossStrategy(
        tickers, events_queue, base_quantity,
        short_window=10, long_window=30
    )

    # Use Yahoo Daily Price Handler
    price_handler = YahooDailyCsvBarPriceHandler(
        csv_dir, events_queue, tickers,
        start_date=start_date,
        end_date=end_date,
        calc_adj_returns=True
    )

    ..
    ..

    # Use an example Risk Manager
    #risk_manager = ExampleRiskManager()
    # Use regime detection HMM risk manager
    hmm_model = pickle.load(open(pickle_path, "rb"))
    risk_manager = RegimeHMMRiskManager(hmm_model)

    ..
    ..
```

To run the backtest it is necessary to open up the Terminal and type the following:

```
$ python regime_hmm_backtest.py
```

The truncated output is as follows:

```
..
..
---------------------------------
Backtest complete.
Sharpe Ratio: 0.48
Max Drawdown: 23.98%
```

31.5 Strategy Results

31.5.1 Transaction Costs

The strategy results presented here are given *net* of transaction costs. The costs are simulated using Interactive Brokers US equities fixed pricing for shares in North America. They are reasonably representative of what could be achieved in a real trading strategy.

31.5.2 No Regime Detection Filter

Figure 31.2 displays the tearsheet for the "no filter" strategy.

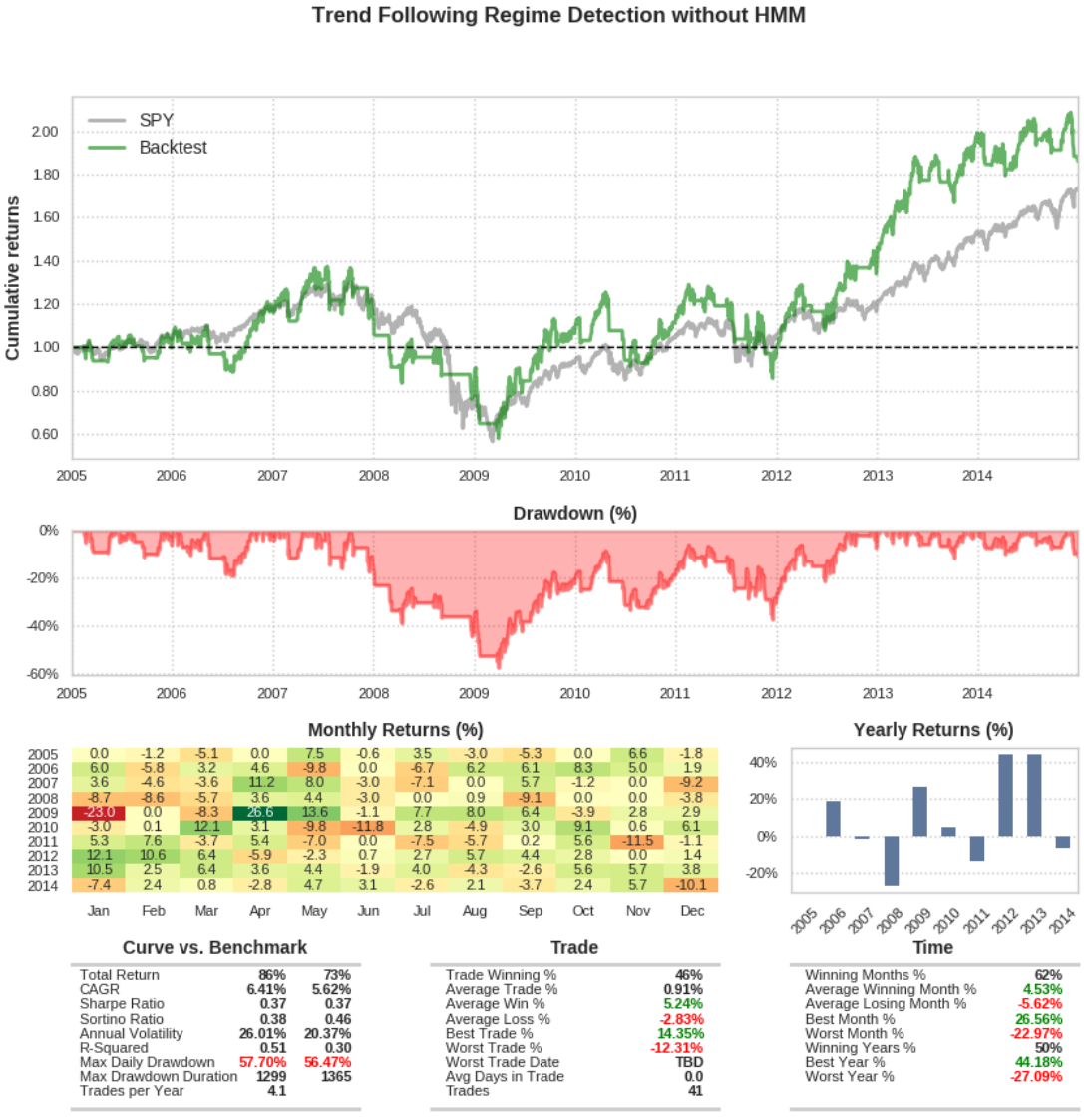

Figure 31.2: Trend Following Regime Detection without HMM

The underlying strategy is designed to capture short-term trends in the SPY ETF. It posts a Sharpe Ratio of 0.37, which means it is taking on a substantial amount of volatility in order to generate the returns. In fact the benchmark has an almost identical Sharpe ratio. The maximum

daily drawdown is slightly larger than the benchmark, but it produces a slight increase in CAGR at 6.41% compared to 5.62%.

In essence the strategy performs about as well as the buy-and-hold benchmark. This is to be expected given how it behaves. It is a lagged filter and, despite making 41 trades, does not necessarily avoid the large downward moves. The major question is whether a regime filter will improve the strategy or not.

31.5.3 HMM Regime Detection Filter

Figure 31.3 displays the tearsheet for the "with HMM filter" strategy.

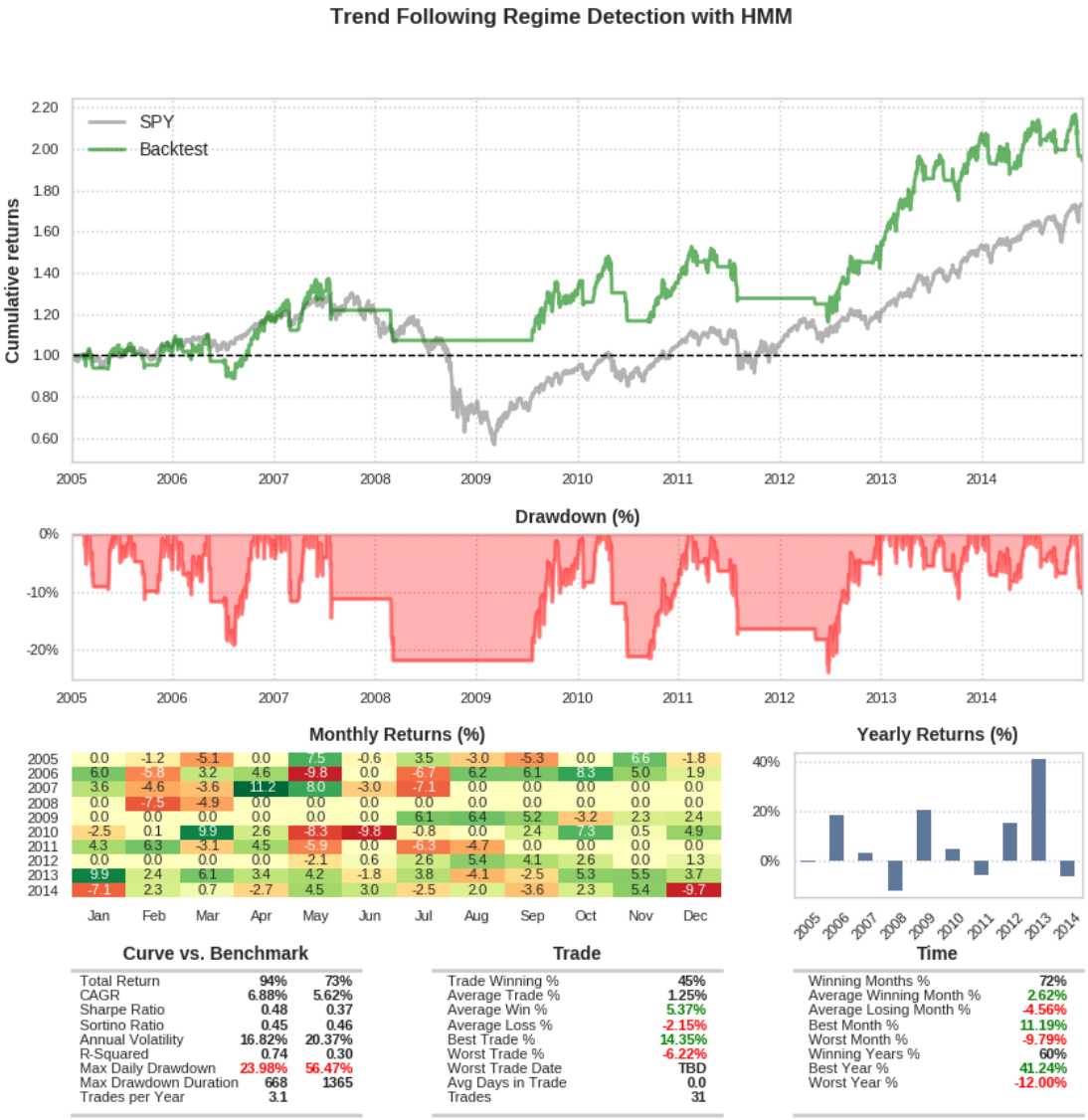

Figure 31.3: Trend Following Regime Detection with HMM

Note that this application of the regime filter is **out-of-sample**. *That is, no returns data used within the backtest were used in the training of the Hidden Markov Model.*

The regime detection filter strategy produces rather different results. Most notably it reduces the strategy maximum daily drawdown to approximately 24% compared to that produced by

the benchmark of approximately 56%. This is a big reduction in "risk". However the Sharpe ratio does not increase too heavily at 0.48 because the strategy still endures a lot of associated volatility to obtain those returns.

The CAGR does not see a vast improvement at 6.88% compared to 6.41% of the previous strategy but its risk has been reduced somewhat.

Perhaps a more subtle issue is that the number of trades has been reduced from 41 to 31. While the trades eliminated were large downward moves (and thus beneficial) it does mean that the strategy is making less "positively expected bets" and so has less statistical validity.

In addition the strategy did not trade at all from early 2008 to mid 2009. Thus the strategy effectively remained in drawdown from the previous high watermark through this period. The major benefit, of course, is that it did not lose money when many others would have!

A production implementation of such a strategy would likely periodically retrain the Hidden Markov Model as the estimated state transition probabilities are very unlikely to be stationary. In essence, the HMM can only predict state transitions based on previous returns distributions it has seen. If the distribution changes (i.e. due to some new regulatory environment) then the model will need to be retrained to capture its behaviour. The rate at which this needs to be carried out is, of course, the subject of potential future research!

31.6 Full Code

```
# regime_hmm_train.py

from __future__ import print_function

import datetime
import pickle
import warnings

from hmmlearn.hmm import GaussianHMM
from matplotlib import cm, pyplot as plt
from matplotlib.dates import YearLocator, MonthLocator
import numpy as np
import pandas as pd
import seaborn as sns


def obtain_prices_df(csv_filepath, end_date):
    """
    Obtain the prices DataFrame from the CSV file,
    filter by the end date and calculate the
    percentage returns.
    """
    df = pd.read_csv(
        csv_filepath, header=0,
```

```
        names=[
            "Date", "Open", "High", "Low",
            "Close", "Volume", "Adj Close"
        ],
        index_col="Date", parse_dates=True
    )
    df["Returns"] = df["Adj Close"].pct_change()
    df = df[:end_date.strftime("%Y-%m-%d")]
    df.dropna(inplace=True)
    return df


def plot_in_sample_hidden_states(hmm_model, df):
    """
    Plot the adjusted closing prices masked by
    the in-sample hidden states as a mechanism
    to understand the market regimes.
    """
    # Predict the hidden states array
    hidden_states = hmm_model.predict(rets)
    # Create the correctly formatted plot
    fig, axs = plt.subplots(
        hmm_model.n_components,
        sharex=True, sharey=True
    )
    colours = cm.rainbow(
        np.linspace(0, 1, hmm_model.n_components)
    )
    for i, (ax, colour) in enumerate(zip(axs, colours)):
        mask = hidden_states == i
        ax.plot_date(
            df.index[mask],
            df["Adj Close"][mask],
            ".", linestyle='none',
            c=colour
        )
        ax.set_title("Hidden State #%s" % i)
        ax.xaxis.set_major_locator(YearLocator())
        ax.xaxis.set_minor_locator(MonthLocator())
        ax.grid(True)
    plt.show()


if __name__ == "__main__":
    # Hides deprecation warnings for sklearn
```

```
    warnings.filterwarnings("ignore")

    # Create the SPY dataframe from the Yahoo Finance CSV
    # and correctly format the returns for use in the HMM
    csv_filepath = "/path/to/your/data/SPY.csv"
    pickle_path = "/path/to/your/model/hmm_model_spy.pkl"
    end_date = datetime.datetime(2004, 12, 31)
    spy = obtain_prices_df(csv_filepath, end_date)
    rets = np.column_stack([spy["Returns"]])

    # Create the Gaussian Hidden markov Model and fit it
    # to the SPY returns data, outputting a score
    hmm_model = GaussianHMM(
        n_components=2, covariance_type="full", n_iter=1000
    ).fit(rets)
    print("Model Score:", hmm_model.score(rets))

    # Plot the in sample hidden states closing values
    plot_in_sample_hidden_states(hmm_model, spy)

    print("Pickling HMM model...")
    pickle.dump(hmm_model, open(pickle_path, "wb"))
    print("...HMM model pickled.")
```

```
# regime_hmm_strategy.py

from __future__ import print_function

from collections import deque

import numpy as np

from qstrader.price_parser import PriceParser
from qstrader.event import (SignalEvent, EventType)
from qstrader.strategy.base import AbstractStrategy


class MovingAverageCrossStrategy(AbstractStrategy):
    """
    Requires:
    tickers - The list of ticker symbols
    events_queue - A handle to the system events queue
    short_window - Lookback period for short moving average
    long_window - Lookback period for long moving average
    """
```

```
    def __init__(
        self, tickers,
        events_queue, base_quantity,
        short_window=10, long_window=30
    ):
        self.tickers = tickers
        self.events_queue = events_queue
        self.base_quantity = base_quantity
        self.short_window = short_window
        self.long_window = long_window
        self.bars = 0
        self.invested = False
        self.sw_bars = deque(maxlen=self.short_window)
        self.lw_bars = deque(maxlen=self.long_window)

    def calculate_signals(self, event):
        # Applies SMA to first ticker
        ticker = self.tickers[0]
        if event.type == EventType.BAR and event.ticker == ticker:
            # Add latest adjusted closing price to the
            # short and long window bars
            price = event.adj_close_price / float(
                PriceParser.PRICE_MULTIPLIER
            )
            self.lw_bars.append(price)
            if self.bars > self.long_window - self.short_window:
                self.sw_bars.append(price)

            # Enough bars are present for trading
            if self.bars > self.long_window:
                # Calculate the simple moving averages
                short_sma = np.mean(self.sw_bars)
                long_sma = np.mean(self.lw_bars)
                # Trading signals based on moving average cross
                if short_sma > long_sma and not self.invested:
                    print("LONG: %s" % event.time)
                    signal = SignalEvent(ticker, "BOT", self.base_quantity)
                    self.events_queue.put(signal)
                    self.invested = True
                elif short_sma < long_sma and self.invested:
                    print("SHORT: %s" % event.time)
                    signal = SignalEvent(ticker, "SLD", self.base_quantity)
                    self.events_queue.put(signal)
                    self.invested = False
            self.bars += 1
```

```
# regime_hmm_risk_manager.py

from __future__ import print_function

import numpy as np

from qstrader.event import OrderEvent
from qstrader.price_parser import PriceParser
from qstrader.risk_manager.base import AbstractRiskManager


class RegimeHMMRiskManager(AbstractRiskManager):
    """
    Utilises a previously fitted Hidden Markov Model
    as a regime detection mechanism. The risk manager
    ignores orders that occur during a non-desirable
    regime.

    It also accounts for the fact that a trade may
    straddle two separate regimes. If a close order
    is received in the undesirable regime, and the
    order is open, it will be closed, but no new
    orders are generated until the desirable regime
    is achieved.
    """
    def __init__(self, hmm_model):
        self.hmm_model = hmm_model
        self.invested = False

    def determine_regime(self, price_handler, sized_order):
        """
        Determines the predicted regime by making a prediction
        on the adjusted closing returns from the price handler
        object and then taking the final entry integer as
        the "hidden regime state".
        """
        returns = np.column_stack(
            [np.array(price_handler.adj_close_returns)]
        )
        hidden_state = self.hmm_model.predict(returns)[-1]
        return hidden_state

    def refine_orders(self, portfolio, sized_order):
        """
        Uses the Hidden Markov Model with the percentage returns
```

```
        to predict the current regime, either 0 for desirable or
        1 for undesirable. Long entry trades will only be carried
        out in regime 0, but closing trades are allowed in regime 1.
        """
        # Determine the HMM predicted regime as an integer
        # equal to 0 (desirable) or 1 (undesirable)
        price_handler = portfolio.price_handler
        regime = self.determine_regime(
            price_handler, sized_order
        )
        action = sized_order.action
        # Create the order event, irrespective of the regime.
        # It will only be returned if the correct conditions
        # are met below.
        order_event = OrderEvent(
            sized_order.ticker,
            sized_order.action,
            sized_order.quantity
        )
        # If in the desirable regime, let buy and sell orders
        # work as normal for a long-only trend following strategy
        if regime == 0:
            if action == "BOT":
                self.invested = True
                return [order_event]
            elif action == "SLD":
                if self.invested == True:
                    self.invested = False
                    return [order_event]
                else:
                    return []
        # If in the undesirable regime, do not allow any buy orders
        # and only let sold/close orders through if the strategy
        # is already invested (from a previous desirable regime)
        elif regime == 1:
            if action == "BOT":
                self.invested = False
                return []
            elif action == "SLD":
                if self.invested == True:
                    self.invested = False
                    return [order_event]
                else:
                    return []
```

```
# regime_hmm_backtest.py

import datetime
import pickle

import numpy as np

from qstrader import settings
from qstrader.compat import queue
from qstrader.event import SignalEvent, EventType
from qstrader.portfolio_handler import PortfolioHandler
from qstrader.position_sizer.naive import NaivePositionSizer
from qstrader.price_handler.yahoo_daily_csv_bar import \
    YahooDailyCsvBarPriceHandler
from qstrader.price_parser import PriceParser
from qstrader.risk_manager.example import ExampleRiskManager
from qstrader.statistics.tearsheet import TearsheetStatistics
from qstrader.strategy.base import AbstractStrategy
from qstrader.trading_session import TradingSession

from regime_hmm_strategy import MovingAverageCrossStrategy
from regime_hmm_risk_manager import RegimeHMMRiskManager


def run(config, testing, tickers, filename):
    # Backtest information
    title = [
        #'Trend Following Regime Detection without HMM'
        'Trend Following Regime Detection with HMM'
    ]
    pickle_path = "/path/to/your/model/hmm_model_spy.pkl"
    events_queue = queue.Queue()
    csv_dir = config.CSV_DATA_DIR
    initial_equity = 500000.00
    start_date = datetime.datetime(2005, 1, 1)
    end_date = datetime.datetime(2014, 12, 31)

    # Use the Moving Average Crossover trading strategy
    base_quantity = 10000
    strategy = MovingAverageCrossStrategy(
        tickers, events_queue, base_quantity,
        short_window=10, long_window=30
    )

    # Use Yahoo Daily Price Handler
```

```
    price_handler = YahooDailyCsvBarPriceHandler(
        csv_dir, events_queue, tickers,
        start_date=start_date,
        end_date=end_date,
        calc_adj_returns=True
    )

    # Use the Naive Position Sizer
    # where suggested quantities are followed
    position_sizer = NaivePositionSizer()

    # Use an example Risk Manager
    #risk_manager = ExampleRiskManager()
    # Use regime detection HMM risk manager
    hmm_model = pickle.load(open(pickle_path, "rb"))
    risk_manager = RegimeHMMRiskManager(hmm_model)

    # Use the default Portfolio Handler
    portfolio_handler = PortfolioHandler(
        PriceParser.parse(initial_equity),
        events_queue, price_handler,
        position_sizer, risk_manager
    )

    # Use the Tearsheet Statistics class
    statistics = TearsheetStatistics(
        config, portfolio_handler,
        title, benchmark="SPY"
    )

    # Set up the backtest
    backtest = TradingSession(
        config, strategy, tickers,
        initial_equity, start_date, end_date,
        events_queue, title=title,
        price_handler=price_handler,
        position_sizer=position_sizer,
        risk_manager=risk_manager,
        statistics=statistics,
        portfolio_handler=portfolio_handler
    )
    results = backtest.start_trading(testing=testing)
    return results
```

```
if __name__ == "__main__":
    # Configuration data
    testing = False
    config = settings.from_file(
        settings.DEFAULT_CONFIG_FILENAME, testing
    )
    tickers = ["SPY"]
    filename = None
    run(config, testing, tickers, filename)
```