Final Project

1. Data Retrieval

1.1 Historical Daily Data

To perform the analysis we need to fetch the historical price data for the DOW 30 stocks over the previous 25 years. We utilize Google Finance to perform this operation by first changing the pandas Data Reader google finance URL to the older version of the google finance API URL.

```
# because pandas datareader url is set to http://www.google.com/finance/historical
# in the package, it is making use of the most up to date google finance api
# url. The problem is that the new google finance api only returns
# 1 year worth of data. To get around that we will replace the new
# api url with the old one which will work until google
# finally permenently disconnects it.
GoogleDailyReader.url = 'http://finance.google.com/finance/historical'
```

After making this URL edit we pull as much historical data as we can from Google Finance. I was able to get data going back to the early 2000's but not anything earlier.

```
############################
# ####### SETTINGS ####### #
############################

# create all the static input variables
# http://money.cnn.com/data/dow30/
# list of tickers to trade
ticker_list = ['MMM', 'AXP', 'AAPL', 'BA',
               'CAT', 'CVX', 'CSCO', 'KO', 'DIS',
               'XOM', 'GE', 'GS', 'HD', 'IBM', 'INTC',
               'JNJ', 'JPM', 'MCD', 'MRK', 'MSFT', 'NKE', 'PFE',
               'PG', 'TRV', 'UTX', 'UNH', 'VZ', 'V', 'WMT']
```

```
# create the date related inputs
in_sample_s_dt = dt.datetime(2000, 1, 1)  # In Sample Start
in_sample_e_dt = dt.datetime(2010, 12, 31)  # In Sample End
out_sample_s_dt = dt.datetime(2011, 1, 1)  # Out of Sample Start
out_sample_e_dt = dt.datetime.today()  # Out of Sample End
```

All Data is retrieved through a uniform interface, known as the DataHandler. The DataHandler is specific to the source of data being retrieved and the Abstract Base Class is constructed like so;

```python
class DataHandler(object):
    """
    Abstract Base Class which contains and fetches all the data that the backtesting engine
    would utilize

    Attributes:
        continue_running (bool): This is set to False when we are out of data to force the Backtester to end
    """

    __metaclass__ = ABCMeta

    def __new__(cls, *args, **kwargs):
        """
        Factory method for base/subtype creation. Simply creates an
        (new-style class) object instance and sets a base property.
        """
        instance = object.__new__(cls, *args, **kwargs)
        # set DataHandler to continue running to begin with
        instance.continue_running = True

        return instance

    @abstractmethod
    def _load_timeseries_data(self):
        """
        Abstract method for loading data from a source
        """
        raise NotImplementedError("Should Implement _load_timeseries_data()")

    def update_data(self, push_market_event = True):
        """
        The main function which asks the DataHandler to get the latest
        set of bars and push them onto the 'latest_ticker_data' dictionary.

        Args:
            push_market_event (bool, optional): If set to False then doesn't push a MarketEvent onto the queue
        """
        # give a temporary first date incase we stop iterating
        curr_dt = dt.datetime(1970, 1, 1)
        for ticker, ticker_gen in self._ticker_data.items():
            try:
                # get next bar from generator
                next_bar = next(ticker_gen)
                # append latest bar to latest_ticker_data
                self.latest_ticker_data[ticker].append(next_bar)
                # set the new curr_dt value
                curr_dt = max(next_bar[0], curr_dt)
            except StopIteration:
                # If I stopped then I am out of data and the backtest does not need to continue
                self.continue_running = False
                push_market_event = False

        if push_market_event:
            self._event_queue.put(MarketEvent(curr_dt))
```

The DataHandler updates a dictionary which is responsible for storing the latest bar data for each ticker.

The subclasses (GoogleFinanceDataHandler) all implement their own version of the _load_timeseries_data function because each data source has a unique and specific API usually.

```
class GoogleDataHandler(DataHandler):
    def __init__(self, event_queue, ticker_list, trade_st_dt, trade_ed_dt, load_data_days_offset = 365, **kwargs):
        """
        DataHandler Class which retrieves its data from Google

        Args:
            event_queue (Queue.queue): queue object which stores each event
            ticker_list (list): list of tickers to get data for
            trade_st_dt (dt.datetime): the start date of the backtest
            trade_ed_dt (dt.datetime): the end date of the backtest
            load_data_days_offset (int, optional): how many days prior to trade_st_dt do I need to get data from?
            **kwargs: any additional keyword arguments
        """
        self._event_queue = event_queue
        self._trade_st_dt = trade_st_dt
        self._trade_ed_dt = trade_ed_dt
        self._data_st_dt = trade_st_dt - dt.timedelta(days = load_data_days_offset)
        self._ticker_data = dict()  # stores all the raw data
        self.ticker_list = ticker_list  # stores a list of the tickers
        self.latest_ticker_data = defaultdict(list)  # stores the latest bars

        self._load_timeseries_data()

    def _load_timeseries_data(self):
        """
        Loads timeseries data from Google
        """
        ticker_df_list = list()
        index_union = None
        for ticker in self.ticker_list:
            # get ticker data from Google
            ticker_df = data.DataReader(ticker,
                                        data_source='google',
                                        start = self._data_st_dt,
                                        end = self._trade_ed_dt)
            ticker_df.columns = [str(x).lower() for x in ticker_df.columns]
            if index_union is None:
                index_union = ticker_df.index
            else:
                index_union = index_union.union(ticker_df.index).sort_values()

            ticker_df_list.append((ticker, ticker_df))

        # determine how much of the data needs to be loaded and pushed
        # to latest_ticker_data at the start (because self._data_st_dt < self._trade_st_dt)
        autoload_idx = index_union[index_union < self._trade_st_dt]
        futureload_idx = index_union.difference(autoload_idx)

        for ticker, ticker_df in ticker_df_list:
            ticker_df = ticker_df.reindex(index_union)
            ticker_df['volume'] = ticker_df['volume'].fillna(0)
            ticker_df = ticker_df.ffill().bfill()
            ticker_df['return'] = ticker_df['close'].pct_change()

            autoload_df = ticker_df.loc[autoload_idx, :]
            futureload_df = ticker_df.loc[futureload_idx, :]

            # add the generator for the future data
            self._ticker_data[ticker] = futureload_df.iterrows()
            # load all the data in the autoload_df to the latest_ticker_data
            autoload_gen = autoload_df.iterrows()
            while True:
                try:
                    next_bar = next(autoload_gen)
                    self.latest_ticker_data[ticker].append(next_bar)
                except StopIteration:
                    break
```

2. Strategy Construction

This project utilizes two different strategies, with 3 different Pyramid Models, and 3 different Position Sizing models. Each strategy is an implementation of a "Strategy" abstract base class to enforce a uniform API with which the strategy would communicate with the data and to make use of functions which would be repeated across strategy class implementations.

The Strategy Class creates a SignalEvent which is passed to the Portfolio through a Queue object before it is transformed into an Order.

```python
class Strategy(object):
    """
    Abstract Base Class of the Strategy
    """

    __metaclass__ = ABCMeta

    @abstractmethod
    def create_signals(self):
        """
        executes strategy logic to create signals for the tickers
        """
        raise NotImplementedError("Should Implement create_signals()")

    def create_signal_event(self, ticker, signal):
        """
        creates a signal event based on the ticker and the signal value
        and pushes the SignalEvent on to the event queue

        Args:
            ticker (string): Ticker we want to be trading
            signal (float): a float value for how much the strategy wants to invest.
                            ie:
                                1.0 means invest 1x whatvever my portfolio will let me invest in this ticker
                                -1.0 means invest -1x whatvever my portfolio will let me invest in this ticker
        """
        self._event_queue.put(SignalEvent(ticker, signal))
```

2.1 EMA Trend Following

The two strategies are both EMA trend following strategies (lookbacks of 21 days or 45 days) whereby if the price moves above (below) 0.5 Average True Range over (under) the EMA, we enter into a long (short) position in the stock. This signal is adjusted using one of the Pyramid Models that we have also created.

```python
class EMA_Strategy(Strategy.Strategy):
    def __init__(self, ticker_list, ema_com, PyramidModel):
        """
        This Strategy executes a trend following system whereby
        we enter into a long position when the current
        price of a ticker is greater than 0.5ATR (lookback equal to ema_com)
        above the EMA price (with lookback equal to ema_com)
        and a short position if the exact opposite is true

        Args:
            ticker_list (list): list of tickers interested in trading
            ema_com (int): center of mass for the EMA calculation
            PyramidModel (PyramidModel.PyramidModel): PyramidModel class to scale signals
        """
        self.ticker_list = ticker_list
        self.ema_com = ema_com
        # set current signals for each ticker to 0
        self.current_signals = dict(zip(ticker_list, np.zeros(len(ticker_list))))
        # create a dictionary to store any entry prices when I enter a position
        self.enter_prices = dict()
        self.PyramidModel = PyramidModel

    def determine_open_position(self, current_price, current_ema, current_atr):
        """
        Determine whether or not I should consider opening a position
        in a ticker

        Args:
            current_price (float): Current price of the ticker
            current_ema (float): Current EMA price of the ticker
            current_atr (float): Current ATR value of the ticker

        Returns:
            float: returns signal for entering into a position
        """
        signal = 0.0
        # calculate half the current ATR value
        current_atr_half = current_atr * 0.5
        # create the cutoff values for entering a long / short position
        enter_long_pr = current_ema * (1.0 + current_atr_half)
        enter_short_pr = current_ema * (1.0 - current_atr_half)
        # if the current price is greater than the enter long cutoff then use signal of 1.0
        if (current_price > enter_long_pr):
            signal = 1.0
        # if the current price is less than the enter short cutoff then use the signal of -1.0
        elif (current_price < enter_short_pr):
            signal = -1.0
        return signal
```

```python
class EMA_Strategy(Strategy.Strategy):
    def __init__(self, ticker_list, ema_com, PyramidModel):
        """
        This Strategy executes a trend following system whereby
        we enter into a long position when the current
        price of a ticker is greater than 0.5ATR (lookback equal to ema_com)
        above the EMA price (with lookback equal to ema_com)
        and a short position if the exact opposite is true

        Args:
            ticker_list (list): list of tickers interested in trading
            ema_com (int): center of mass for the EMA calculation
            PyramidModel (PyramidModel.PyramidModel): PyramidModel class to scale signals
        """
        self.ticker_list = ticker_list
        self.ema_com = ema_com
        # set current signals for each ticker to 0
        self.current_signals = dict(zip(ticker_list, np.zeros(len(ticker_list))))
        # create a dictionary to store any entry prices when I enter a position
        self.enter_prices = dict()
        self.PyramidModel = PyramidModel

    def determine_open_position(self, current_price, current_ema, current_atr): ...

    def calculate_rebalance_signal(self, current_signal, prev_cum_profit, curr_cum_profit):
        """
        Determine what the new signal for the ticker should be because
        we are rebalancing based on the latest data. This is what
        utilizes the Pyramid Model

        Args:
            current_signal (float): The current signal for the ticker
            prev_cum_profit (float): The amount of cumulative profit I have made on this ticker prior to today
            curr_cum_profit (float): The amount of cumulative profit I have made on this ticker including today

        Returns:
            float: the scaled signal value
        """
        # create tuple of input and bool for scaling
        signal_scale_inputs = tuple([current_signal])
        scale_signal = False

        # if we are doing the reflective pyramid we need to do some calculations
        if self.PyramidModel.name == 'RPM':
            scale_signal = True
            cum_prof_diff = curr_cum_profit - prev_cum_profit
            cum_profit_up = cum_prof_diff > 0
            signal_scale_inputs = tuple([current_signal, cum_profit_up])
        # if we are not doing the reflecting pyramid then we just
        # scale up the signal if our current cumulative profit is positive
        elif curr_cum_profit > 0:
            scale_signal = True

        # if we are going to scale the signal then call the Pyramid Model
        if scale_signal:
            new_signal = self.PyramidModel.scale_signal(*signal_scale_inputs)
        else:
            new_signal = current_signal
        return new_signal
```

```python
    def create_signals(self):
        """
        This method creates the signals for the tickers in self.ticker_list
        utilizing the latest data from the DataHandler. After calculating
        the latest signals the strategy will create a SignalEvent
        to push that information on to the event queue
        """
        # the number of latest bars to get from the DataHandler
        num_bars_to_fetch = self.ema_com + 10

        for ticker in self.ticker_list:
            try:
                # set bool to False. We don't trade unless we need to
                submit_signal_to_trade = False
                # get the latest ticker data bars and convert to DataFrame
                ticker_df = self.DataHandler.get_latest_dataframe(ticker, num_bars_to_fetch)
                # get the latest price
                current_pr = ticker_df.iloc[-1]['close']
                # calculate the EMA of the ticker prices
                ticker_ema = AlphaLab.calc_ewma(ticker_df['close'], self.ema_com)
                # calculate the True Range of the ticker prices
                ticker_tr = AlphaLab.calc_true_range(ticker_df)
                # calculate the Average True Range of the ticker prices
                ticker_atr = ticker_tr.rolling(self.ema_com).mean()
                # get the current EMA, ATR, and Signal
                current_ema = ticker_ema.iloc[-1]
                current_atr = ticker_atr.iloc[-1]
                current_signal = self.current_signals.get(ticker)
                # if I currently am not invested with this ticker then lets look
                # to see if we should enter a trade
                if current_signal == 0.0:
                    # check to enter the trade
                    new_signal = self.determine_open_position(current_pr,
                                                              current_ema,
                                                              current_atr)
                    # If I am not going to open up a trade then continue to the next ticker
                    if new_signal == 0.0:
                        continue
                    else:
                        # I am going to enter a position then lets record the enter price
                        # and lets set the submit_signal_to_trade bool to True so I will
                        # create a signal event for this ticker with its new signal
                        self.enter_prices[ticker] = current_pr
                        submit_signal_to_trade = True

                else:
                    # if my current signal is != to 0 then I must currently have
                    # a trade on for this ticker.
                    close_trade = False
                    # check if the current price is higher or lower than the current
                    # EMA value
                    curr_pr_higher_lower = current_pr > current_ema
                    # If I am long and the current price is not greater than the EMA
                    # then lets close the position
                    if (current_signal > 0.0) & (not curr_pr_higher_lower):
                        # check to exit the long position
                        close_trade = True
                    # If I am short and the current price is greater than the EMA
                    # then lets close the position
                    elif (current_signal < 0.0) & (curr_pr_higher_lower):
                        # check to exit the short position
                        close_trade = True

                    if close_trade:
                        # If I am closing the position then my new signal
                        # must be set to 0
                        new_signal = 0.0
                        self.enter_prices.pop(ticker)
                    else:
                        # If I am not closing my trade then I need to
                        # rebalance the trade and use the Pyramid Model
                        curr_signal_sign = np.sign(current_signal)
                        enter_price = self.enter_prices.get(ticker)
                        previous_pr = ticker_df.iloc[-2]['close']

                        prev_cum_profit = ((previous_pr / enter_price) - 1.0) * curr_signal_sign
                        curr_cum_profit = ((current_pr / enter_price) - 1.0) * curr_signal_sign
                        # call Pyramid Model for the scaled signal
                        new_signal = self.calculate_rebalance_signal(current_signal,
                                                                     prev_cum_profit,
                                                                     curr_cum_profit)
```

2.1.1 Buy and Hold

Our Analysis also includes working with a Buy and Hold Strategy so this is also implemented in the same way as the EMA strategy.

```
class BuyHold_Strategy(Strategy.Strategy):
    def __init__(self, ticker_list):
        """
        This Strategy executes a simple Buy and Hold
        with Daily Rebalancing

        Args:
            ticker_list (list): list of tickers interested in trading
        """
        self.ticker_list = ticker_list

    def create_signals(self):
        """
        This method creates the signals for the tickers in self.ticker_list
        utilizing the latest data from the DataHandler. After calculating
        the latest signals the strategy will create a SignalEvent
        to push that information on to the event queue
        """

        for ticker in self.ticker_list:
            self.create_signal_event(ticker, 1.0)
```

2.2 Pyramid Models

We adjust the signal created by the strategy based on one of the Pyramid Models for adding to our positions. This only occurs when we are rebalancing our strategy. The PyramidModel is also an Abstract Base Class to enforce a uniform API for communicating with the rest of the code base. This backtest utilizes three different Pyramid Models; Upright Pyramid Model, Inverted Pyramid Model, Reflective Pyramid Model. For the purpose of this backtest we use a maximum signal of 2.0, which is a 100% increase in the original position of 1.0 (Long or Short).

```
class PyramidModel(object):
    __metaclass__ = ABCMeta

    @abstractmethod
    def scale_signal(self):
        """
        Abstract Method for scaling the signal of the strategy
        """
        raise NotImplementedError("Should Implement scale_signal()")
```

2.2.1 Upright Pyramid Model (UPM)

The Upright Pyramid Model allows us to add to our current position when rebalancing, if our trade is profitable so far, by adding to the current position an amount that is equal to half the amount that was added previously. As an example, if we added 1 unit last time, this time we will add 0.5. We also set a maximum level for the signal so as not to become overly concentrated in just one stock.

```python
class UprightPyramidModel(PyramidModel):
    def __init__(self, max_signal):
        """
        http://www.investopedia.com/articles/trading/09/pyramid-trading.asp

        Upright Pyramid Model to scale the signal.
        As our investment produces positive returns
        we scale up our signal by half of the amount
        that it was previously scaled up by

        Args:
            max_signal (float): A maximum signal so that the strategy cannot request the
                                portfolio to invest more than 'max_signal' times the amount of
                                dollars into this trade as the portfolio would let it
        """
        self.name = 'UPM'
        self.max_signal = max_signal
        # determine all possible signals
        self._possible_signals = self.determine_possible_signals()

    def determine_possible_signals(self):
        """
        This determines all possible signal values so that, given
        the current signal, we can determine easily what the next signal
        will be if we are looking to increase the signal in our trade.

        Returns:
            list: list of all possible signal values
        """
        # set a base signal of 1.0
        curr_signal = 1.0
        all_sigs = [curr_signal]
        # execute while loop to create a list of possible signals
        while True:
            # if the current signal is 1.0 then the difference
            # between the current signal of 1.0 and the previous
            # signal of 0 is a total of 1.0
            if curr_signal == 1.0:
                signal_diff = curr_signal
            else:
                # find out how much we increased the signal last time
                signal_diff = all_sigs[-1] - all_sigs[-2]

            # calculate how much we should add on to our signal
            signal_addon = round(signal_diff / 2.0, 2)
            # calculate the new signal based on how much signal I added
            # on previously and the current signal as compared to what
            # the maximum allowed signal is
            new_signal = min(curr_signal + signal_addon, self.max_signal)
            # append this possible new signal to the all_sigs list
            all_sigs.append(new_signal)
            curr_signal = new_signal
            # if we have reached what the maximum possible signal is
            # then break the while loop
            if new_signal == self.max_signal:
                break
        return all_sigs

    def scale_signal(self, current_signal):
        """
        Scales the signal of the strategy by some factor

        Args:
            current_signal (float): The current signal of the ticker

        Returns:
            float: A scaled signal value to pass to the portfolio
        """
        # get the sign of the current signal
        curr_signal_sign = np.sign(current_signal)
        # if the absolute value of the current signal
        # is less than the maximum signal allowed then
        # lets look to increase the signal along the
        # scale, otherwise dont
        if (abs(current_signal) < self.max_signal):
            # find the index of the current signal among the possible signals
            idx_curr_signal = self._possible_signals.index(abs(round(current_signal, 2)))
            # the new signal is the next signal in line from the list of possible signals
            new_signal = self._possible_signals[idx_curr_signal + 1]
        else:
            new_signal = abs(current_signal)
        # return the new signal adjusted for the appropriate sign on
        return round(new_signal * curr_signal_sign, 2)
```

2.2.2 Inverted Pyramid Model (IPM)

The Inverted Pyramid Model allows us to add to our current position when rebalancing, if our trade is profitable so far, by adding to the current position an equal amount at each point in time that is predetermined. As an example, if we added 1 unit last time, and at each interval we intend to add 0.5, then this time we will add 0.5 and have 1.5. We also set a maximum level for the signal so as not to become overly concentrated in just one stock.

```python
class InvertedPyramidModel(PyramidModel):
    def __init__(self, max_signal, max_n_steps = 8):
        """
        http://www.investopedia.com/articles/trading/09/pyramid-trading.asp

        Inverted Pyramid Model to scale the signal.

        As our investment produces positive returns
        we scale up our signal by an equal amount at
        each step until we reach our max_signal

        Args:
            max_signal (float): A maximum signal so that the strategy cannot request the
                                portfolio to invest more than 'max_signal' times the amount of
                                dollars into this trade as the portfolio would let it
            max_n_steps (int, optional): The maximum number of intervals with which to increase our signal
        """
        self.name = 'IPM'
        self.max_signal = max_signal
        self.max_steps = max_n_steps
        # determine all possible signals
        self._possible_signals = self.determine_possible_signals()

    def determine_possible_signals(self):
        """
        This determines all possible signal values so that, given
        the current signal, we can determine easily what the next signal
        will be if we are looking to increase the signal in our trade.

        Returns:
            list: list of all possible signal values
        """
        # set initial current signal
        curr_signal = 1.0
        # find the difference between the maximum signal and the current signal
        diff_max_signal = self.max_signal - curr_signal
        # calculate the incrementations of the signal for each step
        signal_increment = diff_max_signal / self.max_steps
        # perform the signal incrementing
        all_sigs = [round(curr_signal + (n * signal_increment), 4) for n in range(9)]
        return all_sigs

    def scale_signal(self, current_signal):
        """
        Scales the signal of the strategy by some factor

        Args:
            current_signal (float): The current signal of the ticker

        Returns:
            float: A scaled signal value to pass to the portfolio
        """
        # set the signal index from list to increment equal to 0 (no increment)
        signal_idx_inc = 0
        # get the sign of the current signal
        curr_signal_sign = np.sign(current_signal)
        # get the index from the _possible_signals list of the current signal
        idx_curr_signal = self._possible_signals.index(abs(round(current_signal, 4)))
        # if I am not at the last possible signal then set the increment to 1 position
        if (idx_curr_signal < (len(self._possible_signals) - 1)):
            signal_idx_inc = 1
        # get the next signal
        new_signal = self._possible_signals[idx_curr_signal + signal_idx_inc]
        return round(new_signal * curr_signal_sign, 4)
```

2.2.3 Reflective Pyramid Model (RPM)

The Reflective Pyramid Model allows us to add to our current position when rebalancing, if our trade is profitable so far, by adding to the current position an amount at each point in time that is a function of the amount previously added. We only do this until we reach the half-way point of our profit target. This means that at our half-way point we have the maximum position on. As the trade continues we slowly trim positions so that once we are at our full profit target level we have the same amount invested as we did when we began. We also set a maximum level for the signal so as not to become overly concentrated in just one stock.

```python
class ReflectingPyramidModel(PyramidModel):
    def __init__(self, max_signal):
        """
        http://www.investopedia.com/articles/trading/09/pyramid-trading.asp

        Reflecting Pyramid Model to scale the signal.

        As our investment produces positive returns
        we scale up our signal by an equal to half
        the amounnt it was increased previously. We do this
        until we reach the 'half-way' point of expected profit
        (where our position is maximized) and then we begin
        to decrement the amount in the signal as we approach
        our 'full profit target'. We only increase our position when the current
        cumulative profits are increasing.

        Args:
            max_signal (float): A maximum signal so that the strategy cannot request the
                                portfolio to invest more than 'max_signal' times the amount of
                                dollars into this trade as the portfolio would let it
        """
        self.name = 'RPM'
        self.max_signal = max_signal
        # determine all the possible signals
        self._possible_signals = self.determine_possible_signals()

    def determine_possible_signals(self):
        """
        This determines all possible signal values so that, given
        the current signal, we can determine easily what the next signal
        will be if we are looking to increase the signal in our trade.

        Returns:
            list: list of all possible signal values
        """
        # set current signal
        curr_signal = 1.0
        all_sigs = [curr_signal]
        while True:
            # if current signal is 1 then the amount it has been incremented by is equal to 1
            if curr_signal == 1.0:
                signal_diff = curr_signal
            else:
                # find the amount the signal has been incremented by
                signal_diff = all_sigs[-1] - all_sigs[-2]
            # add on to the signal value
            signal_addon = round(max(signal_diff / 2.0, 0.15), 4)
            # calculate the new signal
            new_signal = min(curr_signal + signal_addon, self.max_signal)
            all_sigs.append(round(new_signal, 4))
            curr_signal = new_signal
            if new_signal == self.max_signal:
                break
        return all_sigs

    def scale_signal(self, current_signal, cum_profit_up = True):
        """
        Scales the signal of the strategy by some factor

        Args:
            current_signal (float): The current signal of the ticker

        Returns:
            float: A scaled signal value to pass to the portfolio
        """
        # get the current signal sign
        curr_signal_sign = np.sign(current_signal)
        # get the index of the current signal in the list of _possible_signals
        idx_curr_signal = self._possible_signals.index(abs(round(current_signal, 4)))
        # set the index increment to 0
        signal_idx_inc = 0
        # if my cumulative profit has increased and my current signal is less than the maximum signal
        # then set the index increment to 1 for a higher signal
        if (cum_profit_up) & (abs(current_signal) < self.max_signal):
            signal_idx_inc = 1
        # else set it to -1 if my cumulative profit is not increasing and my
        # current signal is greater than 1.0, so I move back one index slot
        elif (not cum_profit_up) & (abs(current_signal) > 1.0):
            signal_idx_inc = -1
        # calculate the new signal
        new_signal = self._possible_signals[idx_curr_signal + signal_idx_inc]
        return round(new_signal * curr_signal_sign, 4)
```

2.3 Position Size Models

The Portfolio accepts the signals from the Strategy through the SignalEvent. The Portfolio then takes these signals and, utilizing a position sizing model, creates orders to be passed on to the ExecutionHandler. This backtest utilizes three different Position Size Models; Percent Volatility Model, Markets Money Model, Multi-Tier Model. For purposes of this backtest we always invest (in the beginning) 1% of our equity in a trade, and then scale this up and down based on the Pyramid Model and the Position Size Model.

```python
class PositionSizer(object):
    """
    Abstract Base Class of Position Sizers to be used by the Portfolio
    """

    __metaclass__ = ABCMeta

    @abstractmethod
    def determine_position_size(self):
        """
        Abstract Method for determining the position size of the trade
        based on the portfolio state and the intended signal
        """
        raise NotImplementedError("Should Implement determine_position_size()")
```

2.3.1 Percent Volatility Models (PVM)

The Percent Volatility Model for position sizing utilizes a target volatility and, based on the realized volatility of the asset calculated with an EMA, scales the investment decision to reach the target volatility. For purposes of this backtest we target a volatility of an annualized rate of 20%.

```python
class PercentVolatilityModel(PositionSizer):
    def __init__(self, st_equity_risk_pct, target_vol):
        """
        Position Sizing Model where we calculate the realized volatility
        of the asset and compare it to a target realized volatility. We
        invest a percentage of our equity. We start with an allocation
        of st_equity_risk_pct (a percentage) percent of our equity and
        then we scale this up or down based on the ratio of the realized
        volatility compared to the targeted volatility.

        Args:
            st_equity_risk_pct (float): percentage of equity to risk on the trade. ie: 0.01 (1%% of equity)
            target_vol (float): The targeted annualized volatility we want. ie: 0.15 (15%% annualized volatility)
        """
        self.name = 'PVM'
        self.target_vol = target_vol
        self.st_equity_risk_pct = st_equity_risk_pct
        # set a maximum number of days to use when calculating realized volatility
        self._max_size_roll_window = 63
        # set a minimum number of days to use when calculating realized volatility
        self._min_size_roll_window = 21

    def determine_position_size(self, ohlc_df, current_nav):
        """
        Determine the position size of the trade

        Args:
            ohlc_df (pd.DataFrame): DataFrame containing 'open', 'high' 'low', 'close' data of ticker
            current_nav (float): current nav state of the portfolio. ie: 150 ($150 of NAV)

        Returns:
            float: the number of dollars to invest in this trade
        """
        # get the initial position size in dollars
        pos_size_dollars = self.st_equity_risk_pct * current_nav

        # get the shape of the dataframe that was passed
        df_size = ohlc_df.shape[0]
        # determine what half of its size is to use for the volatility window
        half_size = int(float(df_size) / 2.0)
        # use a window size equal to the minimum of the _max_size_roll_window and the maximum of half the size of the
        # dataframe or the _min_size_roll_window. This ensures that our window size is never less than 21 days (1 month)
        # and never greater than 63 days (1 quarter)
        window_size = min(self._max_size_roll_window, max(half_size, self._min_size_roll_window))
        # calculate the realized volatility
        realized_vol = ohlc_df['return'].ewm(com=window_size).std()
        # annualize the realized volatility
        realized_vol = realized_vol.multiply(np.sqrt(252))
        # scale the number of dollars to invest in this trade by the target volatility
        # as compared to the realized volatility
        position_size_scaler = round(self.target_vol / realized_vol.iloc[-1], 6)
        pos_size_dollars *= position_size_scaler
        return pos_size_dollars
```

2.3.2 Market Money Models (MMM)

The Market Money Model for position sizing invests a fixed percentage of your initial starting equity in the security, and then invests a different percentage of our overall profits in the trade. For purposes of this backtest we invest our usual 1% of initial equity in the trade and invest 50% of our current profits in any given trade as well.

```python
class MarketMoneyModel(PositionSizer):
    def __init__(self, st_equity_risk_pct, profit_risk_pct):
        """
        Position Sizing Model where we calculate our position size based on some
        percentage of our initial starting capital and then we increase our
        position size by some number of dollars that represents
        a percentage (profit_risk_pct) of our total profit so far

        Args:
            st_equity_risk_pct (float): percentage of initial equity to risk on the trade. ie: 0.01 (1%% of initial equity)
            profit_risk_pct (float): percentage of total profits to risk on the trade. ie: 0.5 (50%% of total profits)
        """
        self.name = 'MMM'
        self.st_equity_risk_pct = st_equity_risk_pct
        self.profit_risk_pct = profit_risk_pct

    def determine_position_size(self, initial_capital, current_nav):
        """
        Determine the position size of the trade

        Args:
            initial_capital (float): Number of dollars we initially started with. ie: 150 ($100 of initial equity)
            current_nav (float): current nav state of the portfolio. ie: 150 ($150 of NAV)

        Returns:
            float: the number of dollars to invest in this trade
        """
        # set up an initial position size in dollars based on my initial capital
        pos_size_dollars = self.st_equity_risk_pct * initial_capital
        # calculate my current profits
        curr_profits = current_nav - initial_capital
        # if my current profits are positive then lets look to increase our position size
        if curr_profits > 0:
            # increase our position size by the number of dollars equal to
            # our profit * our profit_risk_pct (percentage)
            pos_size_dollars += (self.profit_risk_pct * curr_profits)
        return pos_size_dollars
```

2.3.3 Multi-Tier Models (MTM)

The Multi-Tier Model for position sizing invests different percentages of your equity in trades based on how much your portfolio has increased in value since your initial investment. As we have more money to invest we invest higher percentages of our equity in each trade. For purposes of this backtest we have the following thresholds where the value to the right of the ":" represents a scaling factor of the 1% initial investment amount.

3% > current_cumulative_return > 0% : 1.0

5% > current_cumulative_return > 3% : 1.1

10% > current_cumulative_return > 5% : 1.25

15% > current_cumulative_return > 10% : 1.5

20% > current_cumulative_return > 15% : 2.0

current_cumulative_return > 20%: 2.5

```python
class MultiTierModel(PositionSizer):
    def __init__(self, st_equity_risk_pct, pct_return_tier_dict):
        """
        Position Sizing Model where we calculate our position size
        based on a set of "threshold levels" of cumulative return for the strategy.
        As our cumulative return exceeds these "threshold levels" we increase
        our position size, otherwise we don't

        Args:
            st_equity_risk_pct (float): percentage of equity to risk on the trade. ie: 0.01 (1%% of equity)
            pct_return_tier_dict (dict): dictionary containing the "threshold levels" of cumulative return and their
                                         corresponding increases in percentage of equity to trade.
                                         ie:
                                         pct_return_tier_dict = {0.03 : 1.1,   # if 0.05 (5 pct) > cumulative return > 0.03 (3 pct) then increase our position size by 10%
                                                                 0.05 : 1.25,  # if 0.1 (10 pct) > cumulative return > 0.05 (5 pct) then increase our position size by 25%
                                                                 0.1 : 1.5,    # if 0.15 (15 pct) > cumulative return > 0.1 (10 pct) then increase our position size by 50%
                                                                 0.15 : 2.0,   # if 0.2 (20 pct) > cumulative return > 0.15 (15 pct) then increase our position size by 100%
                                                                 0.2 : 2.5}    # if cumulative return >= 0.2 (20 pct) then increase our position size by 150%
        """
        self.name = 'MTM'
        self.st_equity_risk_pct = st_equity_risk_pct
        self.pct_return_tier_set = sorted(pct_return_tier_dict.items(), key = lambda x: x[0])

    def determine_position_size(self, initial_capital, current_nav):
        """
        Determine the position size of the trade

        Args:
            initial_capital (float): Number of dollars we initially started with. ie: 150 ($100 of initial equity)
            current_nav (float): current nav state of the portfolio. ie: 150 ($150 of NAV)

        Returns:
            float: the number of dollars to invest in this trade
        """
        pos_size_dollars = self.st_equity_risk_pct * current_nav
        profit_pct = (current_nav / initial_capital) - 1.0
        scalers = [x[1] for x in self.pct_return_tier_set if profit_pct > x[0]]
        if bool(scalers):
            pos_size_dollars *= scalers[-1]
        return pos_size_dollars
```

3. Trading and Execution

All simulated trading incurs a slippage cost of 5bps per trade (half-turn) and a commission of $0.01 per share. All trades are executed through the ExecutionHandler which can be seen below.

```python
class ExecutionHandler(object):
    """
    Abstract Base Class for the Execution Handling
    which is done with the broker
    """

    __metaclass__ = ABCMeta

    @abstractmethod
    def process_order_event(self):
        """
        Abstract method for processing order events
        """
        raise NotImplementedError("Should Implement process_order_event()")


class SimulatedExecutionHandler(ExecutionHandler):
    def __init__(self, event_queue, DataHandler):
        """
        This is a simulated relationship with a broker for
        executing securities.

        Args:
            event_queue (Queue.queue): queue object which allows for events to be processed
            DataHandler (DataHandler.DataHandler): DataHandler class which contains the latest ticker data
        """
        self._event_queue = event_queue
        self.DataHandler = DataHandler
        # slippage in bps for each trade
        self._slippage = 0.0005
        # commission in $ for unit that is traded
        self._commission = 0.01

    def process_order_event(self, event):
        """
        Processes an OrderEvent from the portfolio and sends the
        order to a simulated broker whereby we receive the fill
        information from and then push a FillEvent on to the
        events queue

        Args:
            event (Event.Event): OrderEvent Class
        """
        ticker = event.ticker
        quantity = event.quantity
        side = event.side

        # get the latest data from the DataHandler
        timestamp, latest_bar = self.DataHandler.get_latest_bars(ticker)[-1]
        latest_price = latest_bar['close']

        # calculate the executed price to determine the slippage amount
        if side == 'BUY':
            price_impact = 1.0 + self._slippage
        elif side == 'SELL':
            price_impact = 1.0 - self._slippage

        # Calculate the fill price based on the latest price
        fill_price = (latest_price * price_impact)
        # calculate the commissions paid
        commission = abs(quantity * self._commission)
        # calculate the slippage impact
        slippage = abs(fill_price - latest_price) * abs(quantity)
        # create a fill event
        fill_event = FillEvent(timestamp, ticker, quantity, fill_price, commission, slippage, side)
        # put the FillEvent on to the queue
        self._event_queue.put(fill_event)
```

3. Analysis

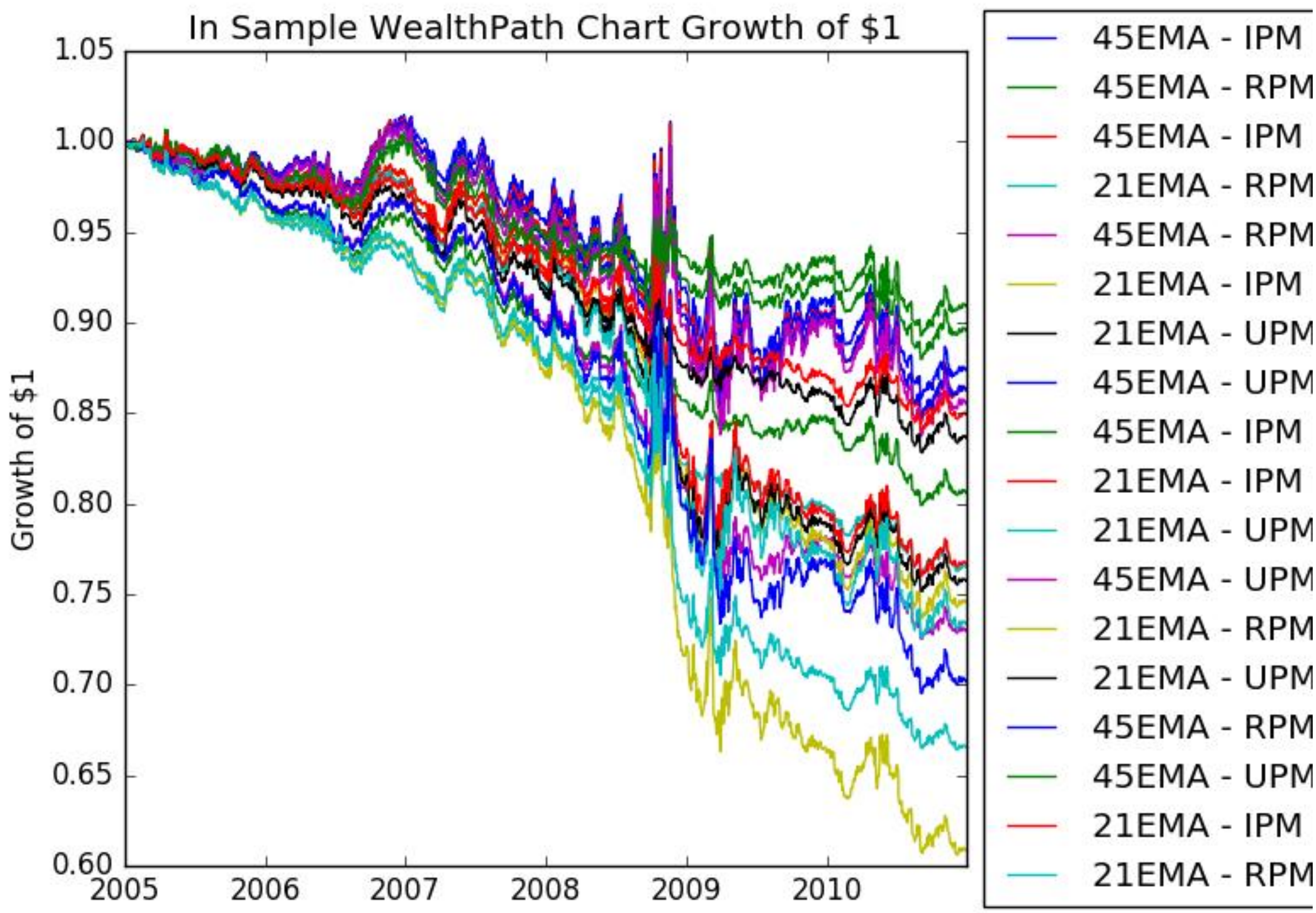
The Analysis is broken up into a number of different sections due to the large volume of information and data. We will begin with an In-Sample Analysis which will contain data on all 18 strategies. We will select just the best performing strategy of the 18 that are tested for Out Of Sample testing. The naming convention for all strategies is as follows;

# of Days for EMA Lookback – Pyramid Model Name – Position Size Model Name

Ie: For a 21 Day EMA strategy utilizing the Upright Pyramid Model and the Percent Volatility Position Size model the name would be "21EMA – UPM – PVM".

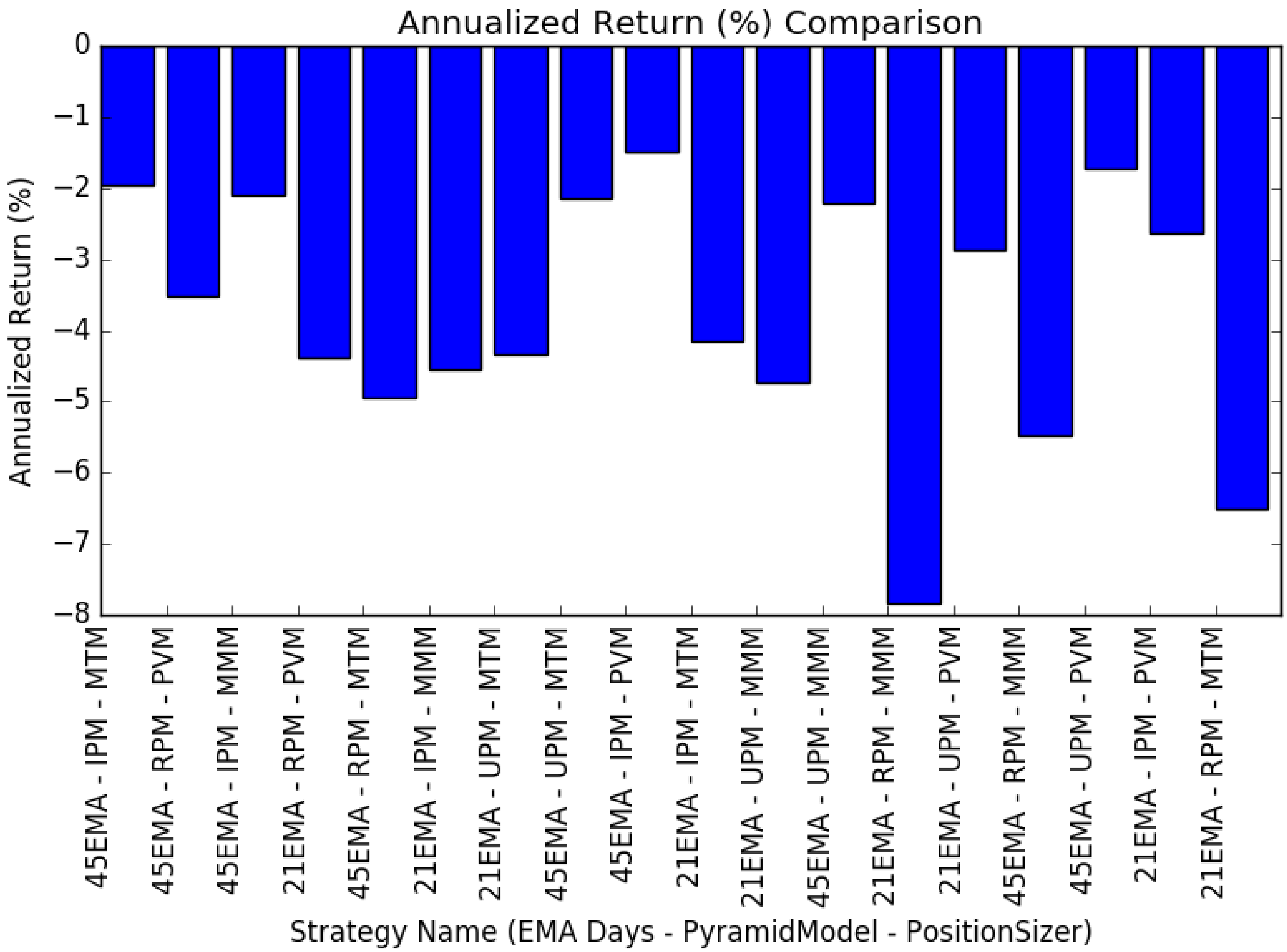
3.1.1 In-Sample Wealth Paths

Unfortunately it is very difficult to fit this large amount of information on a single chart for comparison. As a result some of the names do not fit in the chart legend. What we can take away from this chart though is that we don't have a single strategy which produces a positive return over this time period. Every single strategy managed to lose money over the course of the entire In-Sample test. This does not bode well for the Out-Of-Sample tests.



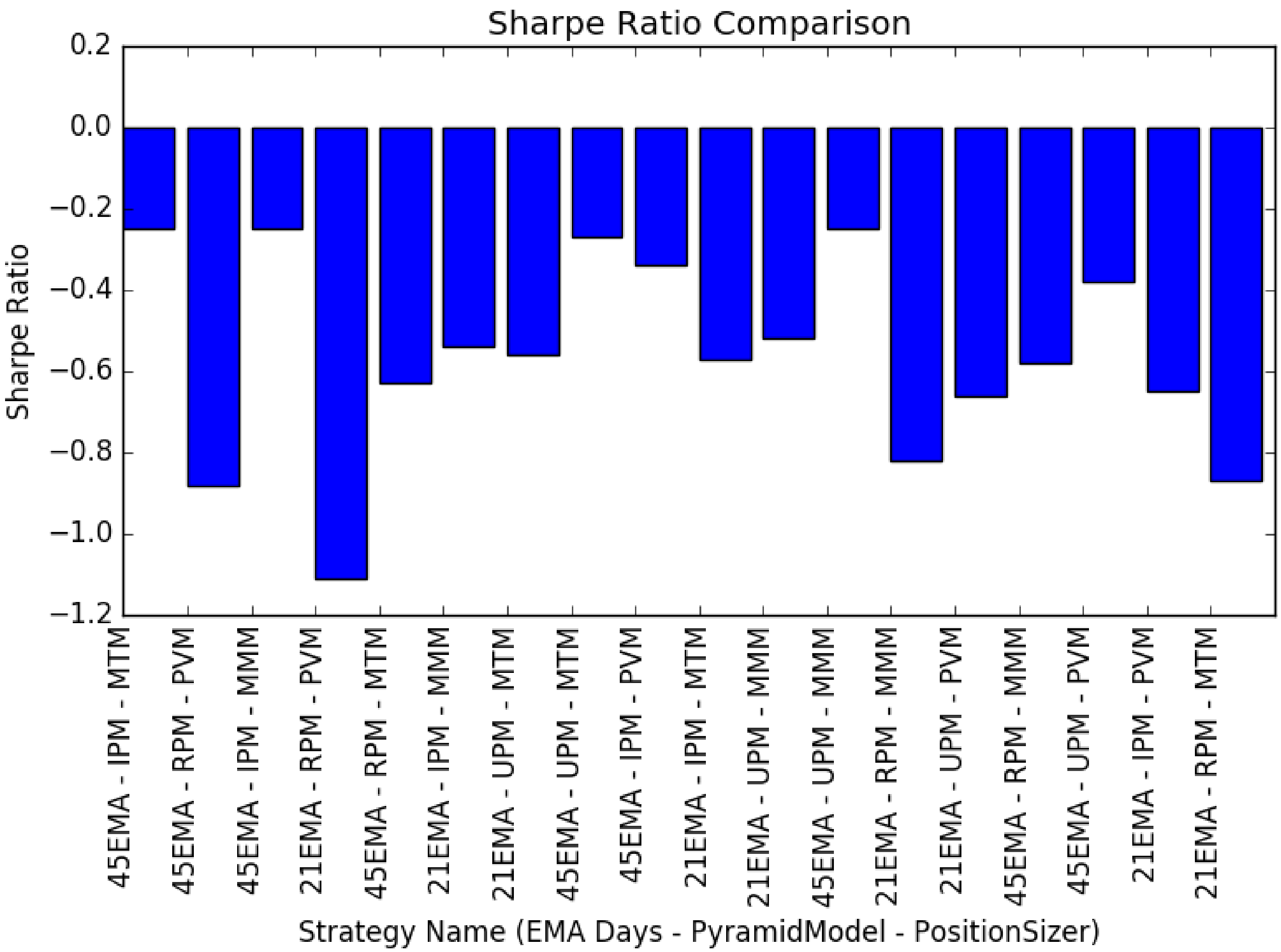
3.1.2 In-Sample Annualized Return

As we can see from the chart below, all strategies have a negative annualized return. Some good information we can glean from this analysis is that the strategies utilizing the 45 Day EMA all tended to do better than their 21 Day EMA counterpart. No other information is blatantly clear from the results.



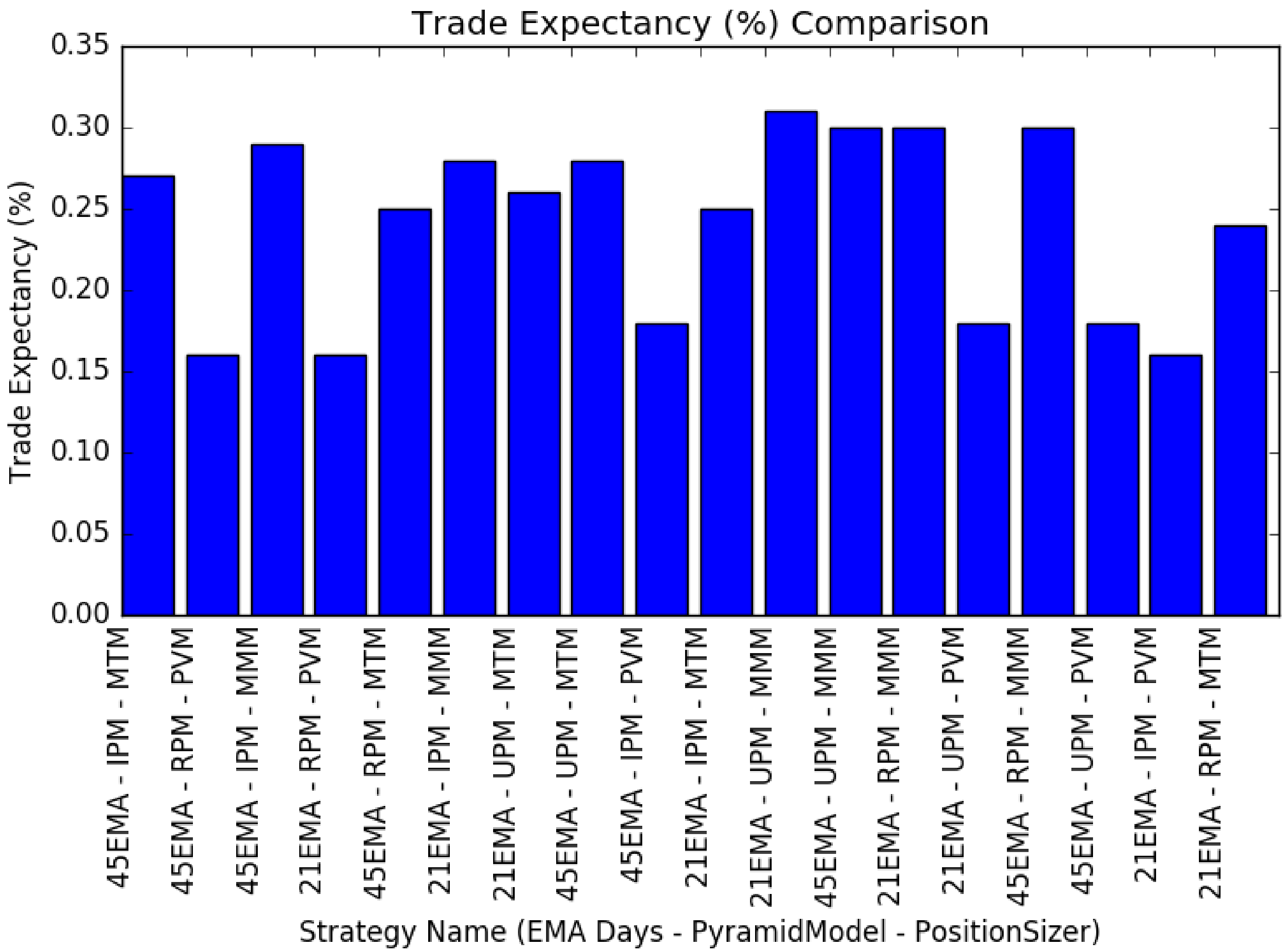
3.1.3 In-Sample Sharpe Ratio

Given the poor annualized returns and the image of the wealth path, it is no surprise that most strategies have pretty terrible Sharpe Ratios. We can get similar information from the chart below as we did from the annualized return chart, namely that the 45 Day EMA tends to have a Sharpe which is “less worse” than their corresponding 21 Day counterpart. The other piece of information that I find useful here is to note that, in most cases, the Percent Volatility Model (PVM) for Position Sizing ended up with a lower Sharpe Ratio on average as compared to the other two (MMM, MTM).



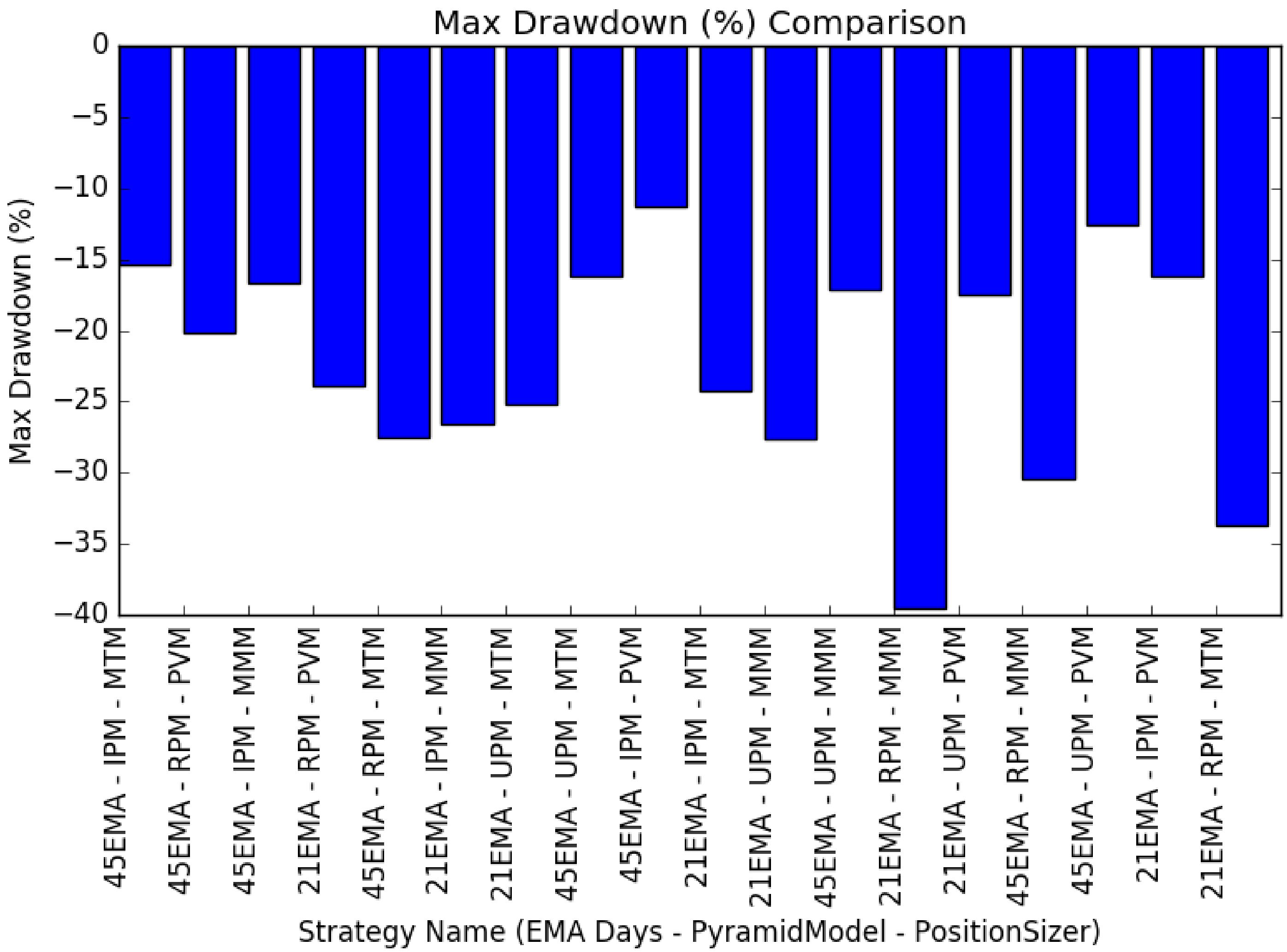
3.1.4 In-Sample Trade Expectancy

Surprisingly each model has a positive trade expectancy. Unfortunately, because each trade has a trade expectancy less than 0.3% on average, simply trading the strategy is enough to remove all the positive returns from the portfolio. I don't find the same amount of clearly defined difference between the 45 Day and 21 Day strategies, but I do find that the models using the Money Market Model for Position Sizing tend to have higher trade expectancies than their counterparts.



3.1.5 In-Sample Max Drawdown

Most strategies have a Maximum Drawdown which we could live with. The 21 Day strategies typically have a worse Max Drawdown than their 45 Day counter parts. It is also interesting to see that the Inverted Pyramid Model tends to outperform the others.



3.1.6 In-Sample Return Statistics

When looking at the best return statistics of the strategies we can still see how disappointing their overall performance truly is. Below are the top 6 performing strategies, all of which are a 45 Day Strategy. All have a Max Drawdown between -10% and -20%, an annualized return between -1.5% and -2.3%, and a loss rate that is basically 50%. These results lean towards the argument that this strategy, regardless of how it is constructed, is not a profitable one.

	45EMA - IPM - MTM	45EMA - IPM - MMM	45EMA - UPM - MTM	45EMA - IPM - PVM	45EMA - UPM - MMM	45EMA - UPM - PVM
Annualized Downside Dev	7.1%	7.6%	7.3%	3.7%	7.9%	3.9%
Annualized Return	-2.0%	-2.1%	-2.1%	-1.5%	-2.2%	-1.7%
Annualized Std Dev	7.8%	8.4%	8.0%	4.4%	8.8%	4.6%
Avg Loss Return	-0.3%	-0.3%	-0.3%	-0.2%	-0.3%	-0.2%
Avg Win Return	0.3%	0.3%	0.3%	0.2%	0.3%	0.2%
Best Month Return	3.1%	3.6%	3.5%	2.0%	4.1%	2.3%
Gain to Pain Ratio	-0.02	-0.02	-0.02	-0.03	-0.02	-0.03
Lake Ratio	0.06	0.07	0.07	0.05	0.07	0.06
Loss Rate	49.5%	49.7%	49.6%	49.5%	49.5%	49.1%
Max Drawdown	-15.5%	-16.7%	-16.2%	-11.3%	-17.1%	-12.6%
Percent Profitable Months	52.8%	52.8%	51.4%	50.0%	52.8%	48.6%
Sharpe Ratio	-0.25	-0.25	-0.27	-0.34	-0.25	-0.38
Sortino Ratio	-0.28	-0.28	-0.29	-0.41	-0.28	-0.45
Trade Expectancy	0.3%	0.3%	0.3%	0.2%	0.3%	0.2%
Win Rate	50.5%	50.2%	50.3%	50.4%	50.4%	50.8%
Worst Month Return	-5.9%	-6.5%	-5.8%	-2.8%	-6.6%	-3.0%

3.1.7 In-Sample Strategy Ranking

In order to not put any unnecessary bias in the analysis and selection of the strategy to use for Out-Of-Sample testing I decided to rank each strategy where the best strategy would get a score of 18 and the worst strategy would get a score of 1. At the end we sum up the points each strategy has and we select the strategy with the top score. We rank each strategy based upon the following KPI's; Sharpe Ratio, Sortino Ratio, Win Rate, Trade Expectancy, and Max Drawdown.

```
def sort_and_score_series(series):
    # get the series values and sort them
    sorted_series = series.sort_values()
    # score the values
    new_series = pd.Series(index = sorted_series.index, data = range(1, len(sorted_series) + 1), name = series.name)
    return new_series
```

```
    print "%s - Selecting The Best In Sample Model" % (dt.datetime.now().strftime('%Y-%m-%d %H:%M:%S'))
    # sort and score the in sample strategies based on their metrics
    MetricsToScore = ['Sharpe Ratio', 'Sortino Ratio', 'Max Drawdown', 'Trade Expectancy', 'Win Rate']
    ScoredMetrics = list()
    for Metric in MetricsToScore:
        ScoredSeries = sort_and_score_series(InSampleOutput_Df.loc[Metric, :])
        ScoredMetrics.append(ScoredSeries)

    # select the backtest to use for out of sample
    InSampleScoredBacktests = pd.concat(ScoredMetrics, axis=1).sum(axis=1).sort_values()
    BestInSampleBacktestName = InSampleScoredBacktests.index[-1]
    BestInSampleParameters = InSampleOutputDict[BestInSampleBacktestName]['components']
    BestInSampleParameters['SampleStartDt'] = out_sample_s_dt
    BestInSampleParameters['SampleEndDt'] = out_sample_e_dt
```

4. Results

Below are the output results for the optimally selected in-sample strategy compared to an equal weight of the 10 least correlated stocks and a simple buy and hold approach. To get these results we also had to run these strategies so they are shown below. We have Out-Of-Sample results for the strategy we selected based on In-Sample statistics and then we run the best strategy with the 10 least correlated stocks and a simple Buy & Hold of the index over the entire sample period.

```
# create some new parameter groups for backtesting
FullPeriodParameters = [{'TickerList' : ['DIA'],
                         'InitialCapital' : init_capital,
                         'SampleStartDt' : in_sample_s_dt,
                         'SampleEndDt' : out_sample_e_dt,
                         'PositionSizers' : (PositionSizers.PercentEquityModel, tuple([1.0])),
                         'BacktestName' : 'Buy & Hold DIA Index'
                         },
                         {'TickerList' : LeastCorrelatedTickers,
                          'InitialCapital' : init_capital,
                          'SampleStartDt' : in_sample_s_dt,
                          'SampleEndDt' : out_sample_e_dt,
                          'PositionSizers' : (PositionSizers.PercentEquityModel, tuple([1.0 / len(LeastCorrelatedTickers)])),
                          'BacktestName' : 'Buy & Hold 10 Least Correlated'
                          }]
```

```
# get full period results for the Buy and Hold Strategies
FullPeriod = dict()
for ParameterCombo in FullPeriodParameters:
    BacktesterObj, BacktestName = run_buy_hold_backtest(ParameterCombo)
    FullPeriod[BacktestName] = {'nav_df' : BacktesterObj.nav_df,
                                'trades_df' : BacktesterObj.trades_df,
                                'components' : None}

# get full period results for the Best In Sample Strategy
BestInSampleParameters['SampleStartDt'] = in_sample_s_dt
FullPeriodBacktesterObj, BacktestName = run_ema_backtest(BestInSampleParameters)
FullPeriod[BacktestName] = {'nav_df' : FullPeriodBacktesterObj.nav_df,
                            'trades_df' : FullPeriodBacktesterObj.trades_df,
                            'components' : None}

print "%s - Creating Full Period Wealth Paths and Return Stats" % (dt.datetime.now().strftime('%Y-%m-%d %H:%M:%S'))
# Create Full Period Wealthpaths
FullPeriodWealthPathList = list()
for BacktestName, BacktestOutputDict in FullPeriod.items():
    nav_df = BacktestOutputDict['nav_df'].set_index('date')
    wealthpath = nav_df['nav'].divide(nav_df['nav'].iloc[0])
    wealthpath.name = BacktestName
    FullPeriodWealthPathList.append(wealthpath)

FullPeriodWealthPath_Df = pd.concat(FullPeriodWealthPathList, axis=1)
# Create Wealthpath Plot
FullPeriodWealthPathFig, FullPeriodWealthPathAx = create_line_plot(FullPeriodWealthPath_Df,
                                                                  'Growth of $1',
                                                                  'Full Period WealthPath Chart Growth of $1')
FullPeriodWealthPathFig.savefig('FullPeriod - WealthPaths.png')

# generate output statistics for full sample period
FullPeriodOutput_Obj = Output.Output(FullPeriodWealthPath_Df.pct_change())
FullPeriodOutput_Df = FullPeriodOutput_Obj.generate_output()
FullPeriodOutput_Df.to_csv('FullPeriodStats.csv')
```

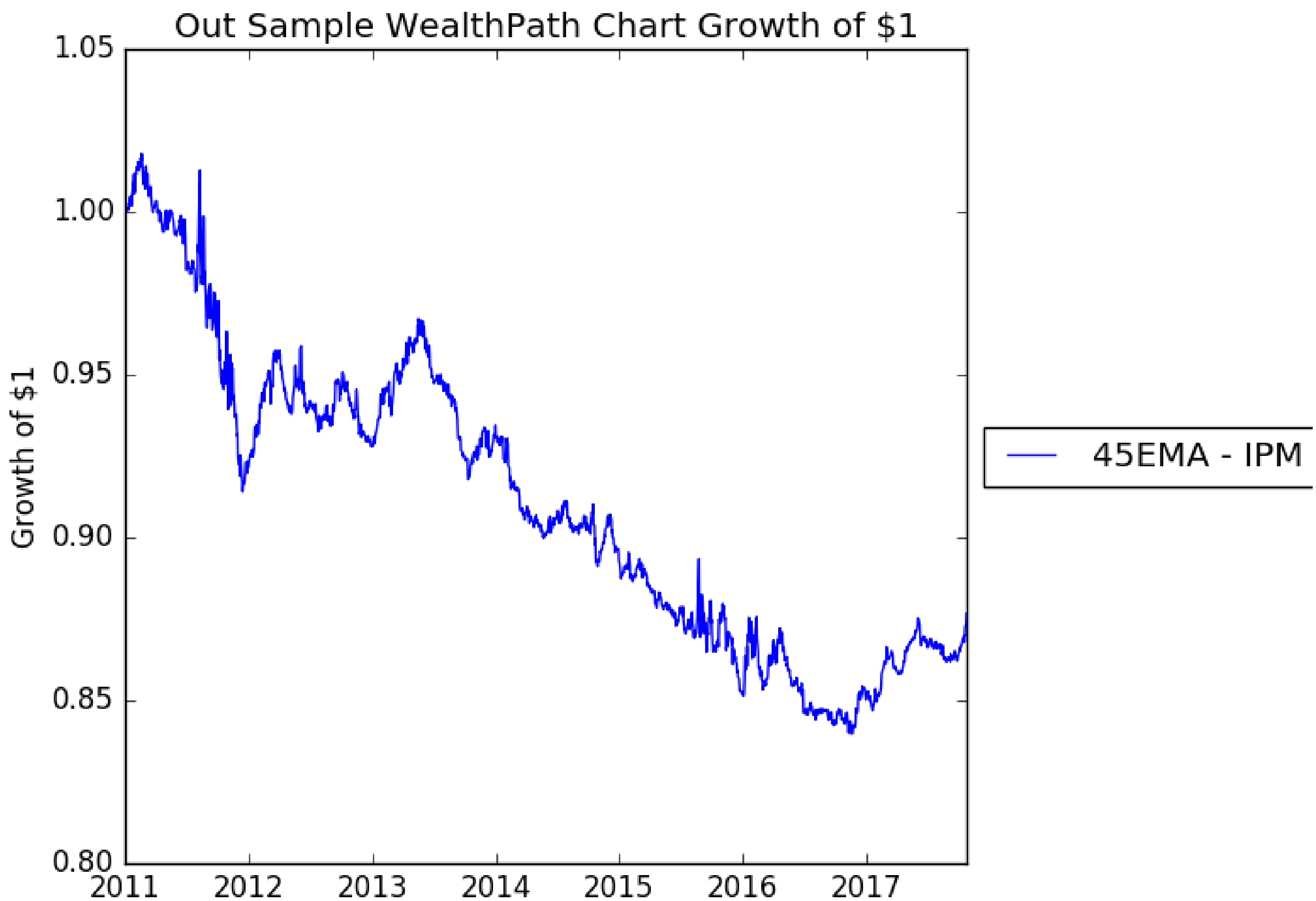
4.1 Out-Of-Sample Strategy Return Statistics and Wealth Path

We utilize our backtesting engine to run the best In-Sample strategy over the Out-Of-Sample Period.

```
# select the backtest to use for out of sample
InSampleScoredBacktests = pd.concat(ScoredMetrics, axis=1).sum(axis=1).sort_values()
BestInSampleBacktestName = InSampleScoredBacktests.index[-1]
BestInSampleParameters = InSampleOutputDict[BestInSampleBacktestName]['components']
BestInSampleParameters['SampleStartDt'] = out_sample_s_dt
BestInSampleParameters['SampleEndDt'] = out_sample_e_dt

print "%s - Running Out Of Sample with Best In Sample Model" % (dt.datetime.now().strftime('%Y-%m-%d %H:%M:%S'))
# run the out of sample backtest
BestInSampleBacktesterObj, BestInSampleBacktestName = run_ema_backtest(BestInSampleParameters)
BestInSampleBacktestNav = BestInSampleBacktesterObj.nav_df.set_index('date')['nav']
BestInSampleBacktestNav.name = BestInSampleBacktestName

# generate output statistics for out of sample period
OutSampleOutput_Obj = Output.Output(BestInSampleBacktestNav.pct_change())
OutSampleOutput_Df = OutSampleOutput_Obj.generate_output()
OutSampleOutput_Df.to_csv('OutSampleStats.csv')

# create out of sample wealthpath to plot
OutSampleWealthPath_Df = pd.DataFrame(BestInSampleBacktestNav)
OutSampleWealthPath_Df.columns = [BestInSampleBacktestName]
OutSampleWealthPath_Df = OutSampleWealthPath_Df.divide(OutSampleWealthPath_Df.iloc[0])
OutSampleWealthPath_Df = pd.DataFrame(OutSampleWealthPath_Df)

OutSampleWealthPathFig, OutSampleWealthPathAx = create_line_plot(OutSampleWealthPath_Df,
                                                                 'Growth of $1',
                                                                 'Out Sample WealthPath Chart Growth of $1')
OutSampleWealthPathFig.savefig('OutSample - WealthPaths.png')
```

4.1.1 Out-Of-Sample Strategy Wealth Path

Not surprisingly, the strategy that performed best In-Sample (which it had poor results even In-Sample) continues to have poor performance in the Out-Of-Sample period. We see that the strategy did not manage to generate any positive returns except for a brief period in 2013 and currently YTD for 2017. This results match our expectations from the In-Sample period and suggest that this is not a true Alpha signal.



4.1.2 Out-Of-Sample Strategy Return Statistics

The Out-Of-Sample return statistics confirm the poor performance we can see in the wealth path. The strategy has a loss rate of almost 50%, a negative annualized return, and a max drawdown of almost -20%. Overall the results are very disappointing.

	45EMA - IPM - MTM
Annualized Downside Dev	3.5%
Annualized Return	-1.8%
Annualized Std Dev	4.2%
Avg Loss Return	-0.2%
Avg Win Return	0.2%
Best Month Return	1.8%
Gain to Pain Ratio	-0.04
Lake Ratio	0.12
Loss Rate	49.2%
Max Drawdown	-17.5%
Percent Profitable Months	41.5%
Sharpe Ratio	-0.44
Sortino Ratio	-0.53
Trade Expectancy	0.2%
Win Rate	50.7%
Worst Month Return	-3.0%

4.2 Full Period Ten (10) Least Correlated Stocks

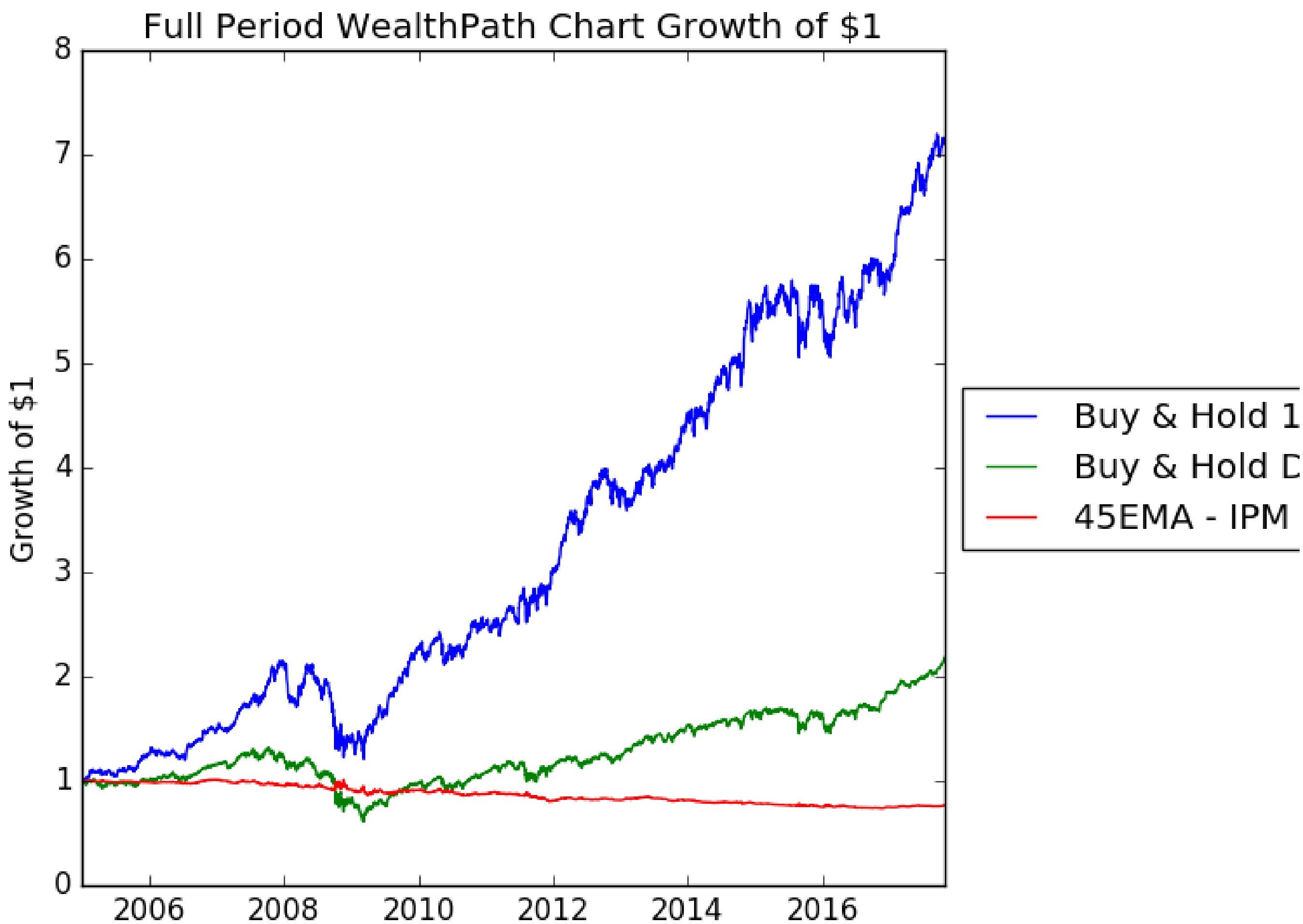
As part of our analysis we must create a portfolio of the 10 least correlated stocks and use a Buy & Hold approach to check their performance.

```
def get_ticker_correlations(backtester_object):
    # extract ticker pair by pair correlations
    returns_df_list = list()
    for ticker in backtester_object.ticker_list:
        # get the ticker data from DataHandler
        ticker_df = backtester_object.DataHandler.get_latest_dataframe(ticker, 0)
        # get the returns
        ticker_returns = ticker_df['return']
        ticker_returns.name = ticker
        returns_df_list.append(ticker_returns)
    # create a DataFrame for the returns
    returns_df = pd.concat(returns_df_list, axis=1)
    # return the correlated returns
    return returns_df.corr()


def find_N_least_correlated_tickers(correlation_df, N):
    # get the combination of tickers
    ticker_pairs = itertools.combinations(correlation_df.index.tolist(), 2)
    # create a dictionary of all the pairs and their values
    correlation_dict = {pair : correlation_df.loc[pair[0], pair[1]] for pair in ticker_pairs}
    # convert the dictionary to a pandas series
    correlation_series = pd.Series(correlation_dict).drop_duplicates().sort_values()
    # select the least correlated tickers
    tickers_to_select = list()
    for ticker_pair in correlation_series.index:
        ticker1, ticker2 = ticker_pair
        if ticker1 not in tickers_to_select:
            tickers_to_select.append(ticker1)

        if len(tickers_to_select) < 4:
            if ticker2 not in tickers_to_select:
                tickers_to_select.append(ticker2)
        else:
            break
    return tickers_to_select
```

```
    # now we get the in sample ticker correlations
    Correlation_Df = get_ticker_correlations(InSampleBacktesterObj)
    # now we find the 10 least correlated tickers
    LeastCorrelatedTickers = find_N_least_correlated_tickers(Correlation_Df, 10)
```

4.4 Full Period Return Statistics & Wealth Path

The Blue line is the Buy & Hold of the 10 Least Correlated Stocks. The performance is truly astonishing compared to just the regular Buy and & Hold model (Green line). The Strategy continues unabated in falling and losing money consistently.



We see that the 10 Least Correlated portfolio has some very impressive stats, with a Sharpe ratio close to 1, a win rate slightly above Buy & Hold the index, and a drawdown that is marginally better than Buy & Hold the index. The Strategy performance is terrible in comparison.

	Buy & Hold 10 Least Correlated	**Buy & Hold DIA Index**	**45EMA - IPM - MTM**
Annualized Downside Dev	13.3%	14.0%	5.5%
Annualized Return	17.0%	7.7%	-1.9%
Annualized Std Dev	18.2%	17.7%	6.2%
Avg Loss Return	-0.8%	-0.7%	-0.2%
Avg Win Return	0.8%	0.7%	0.2%
Best Month Return	12.5%	9.6%	3.1%
Gain to Pain Ratio	0.00	0.00	-0.01
Lake Ratio	0.04	0.10	0.16
Loss Rate	45.3%	45.6%	49.4%
Max Drawdown	-43.8%	-53.8%	-27.4%
Percent Profitable Months	64.9%	63.0%	46.8%
Sharpe Ratio	0.93	0.43	-0.30
Sortino Ratio	1.27	0.55	-0.34
Trade Expectancy	0.8%	0.7%	0.2%
Win Rate	54.7%	54.3%	50.6%
Worst Month Return	-16.0%	-13.7%	-5.9%

4.5 Conclusion

This was an interesting exercise to test a simple trading strategy but as we can see from the evidence, markets are not so simple. The strategy performs extremely poorly in a majority of cases and even in the best case the performance is still significantly worse than a simple Buy & Hold of the Index. I think this assignment tries to get too “crafty” (if you will…) with the different Pyramid Models and the different Position Sizing methods. I think if we can find a slight edge to exploit, with say a 55% win rate, then we should make that bet as often as we can with a portion of our portfolio so as to minimize the risk of ruin (something like Kelly).

I think the strategy could be greatly improved by changing the strategy completely. For starters, if we had some notion of Jump Density for each stock, along with what the Implied Volatility surface says about the stock, a notion of trend using short term and longer term moving averages, and some features on how the stock performs relative to its peers and the index in conjunction with their respective correlations, we could utilize some simple machine learning algorithms to determine the probability of our investment being profitable (such as a Random Forest Classifier). With all these enhanced features and a probability of obtaining a profit we could then apply notions like the Pyramid Models and the Position Size models. Allowing a human to determine whether or not to buy because a stock is above its EMA and above 0.5 ATR of its EMA seems too arbitrary and does not separate our data enough into useful trades.