Trading Simulation Based on MACD indicator and Locally Weighted Linear Regression algorithm

March 25, 2022

1 Implementation of MACD indicator and using it to construct a trading algorithm with locally weighted linear regression algorithm

1.1 General overview of $MACD$ implementation

$MACD$ and $SIGNAL$ indicators are calculated with usage of moving average EMA for N periods given by:

$$EMA_N = \frac{p_0 + (1-\alpha)p_1 + (1-\alpha)^2 p_2 + ... + (1-\alpha)^N p_N}{1 + (1-\alpha) + (1-\alpha)^2 + ... + (1-\alpha)^N}$$

where p_i is sample from i days and $\alpha = \frac{2}{N+1}$

Funkcja EMA przyjmuje jako parametr N elementowy wektor danych

[3]:
```
def EMA(v):
    n = len(v)
    a = 2 / (n + 1)
    numerator = 0
    denominator = 0
    for i in range(n):
        numerator += v[i] * ((1 - a) ** i)
        denominator += (1 - a) ** i
    return numerator / denominator
```

$MACD$ indicator is calculated as $EMA_{12} - EMA_{26}$ form data vector, and $SIGNAL$ indicator as EMA_9 z $MACD$

$MACD$ function takes data vector as a parameter (present day - the one for which we calculate the indicator is the last element of vector), similarly to $SIGNAL$ function

[4]:
```
def MACD(v):
    ema_12 = EMA(v[-12:])
    ema_26 = EMA(v[-26:])
    return ema_12 - ema_26
```

```python

def SIGNAL(macd):
    return EMA(macd[-9:])
```

[ ]:
```python
if __name__ == "__main__":
    column = -2
    data_dir = 'wig20.csv'
    data = pd.read_csv(data_dir)
    data_vector = data.iloc[:, column].to_numpy()
    main(data_vector[:1000], sim_type="basic")
```

Main function parameter are data vector and optionally parameters regarding simulation method. There are two methods for simulation: a simple one and a little more complicated including locally weighted linear regression model.

Main function iteratively calculates elements of $MACD$ and $SIGNAL$ vectos. ***last_action*** and ***prev_dif*** variables are used for simulations.

[ ]:
```python
def main(data, sim_type="advanced", dist=1.5, tau=0.8, amount=1):
    macd_arr = np.empty(0)
    signal_arr = np.empty(0)

    if sim_type == "basic":
        prev_dif = 0
    else:
        last_action = "none"

    if amount > 1:
        amount = 1
    # initialization of data for simulation, initial actions amount = 1000
    actions = 1000
    money = 0
    start_value = money + actions * data_vector[0]
```

First loop calculates first 9 elements of $MACD$ vector and first $SIGNAL$ value.

[ ]:
```python
     for i in range(26, 35):
        macd_arr = np.append(macd_arr, MACD(data[:i]))

    signal_arr = np.append(signal_arr, SIGNAL(macd_arr))
```

Before the main loop axes for plotting are prepared

[ ]:
```python
    fig, ax = plt.subplots(2, sharex=True)
    plt.xlabel("days")
    ax[1].set_ylabel("price")
    ax[0].set_ylabel("value")
    axes_1 = fig.gca()
```

```
    # plotting input data
    ax[1].bar(np.arange(0, len(data)), data - min(data), width=1,␣
↪color='#80ede2', bottom=min(data))
    # saving for later use
    y_min_1, y_max_1 = axes_1.get_ylim()
```

Plotting $MACD$ and $SIGNAL$

```
[ ]:     for i in range(35, len(data)):
             macd_arr = np.append(macd_arr,  MACD(data[:i]))
             signal_arr = np.append(signal_arr, SIGNAL(macd_arr))

             ax[0].plot([i - 1, i], [signal_arr[-2], signal_arr[-1]], color='b',␣
↪label='SIGNAL')
             ax[0].plot([i - 1, i], [macd_arr[-2], macd_arr[-1]], color='r',␣
↪label='MACD')
```

In every iteration (how it is done depends on choosen simulation type) actions and money values are reevaluated.

```
[ ]: if sim_type == "basic":
             prev_dif, actions, money = basicSim(actions, amount, ax, i,␣
↪macd_arr, money, prev_dif, signal_arr, y_max_1,
                                                   y_min_1)
         else:
             last_action, actions, money = advancedSim(actions, amount, ax,␣
↪dist, i, last_action, macd_arr, money,
                                                         signal_arr, tau, y_max_1,␣
↪y_min_1)
```

Print value of assets in last day of simulation and compare it to value from first day

```
[ ]: end_value = money + actions * data_vector[len(data_vector) - 1]
           print(f"start value: {start_value}, end value : {end_value}, diff:␣
↪{end_value - start_value} "
               f"( {100 * end_value /start_value}  % of start value)")
```

Show plots

```
[ ]: # remove repeating labels
    handles, labels = ax[1].get_legend_handles_labels()
    by_label = dict(zip(labels, handles))
    ax[1].legend(by_label.values(), by_label.keys(), loc='lower left')
    handles, labels = ax[0].get_legend_handles_labels()
    by_label = dict(zip(labels, handles))
    ax[0].legend(by_label.values(), by_label.keys(), loc='lower left')
```

```
    fig.tight_layout()
    plt.show()
```

1.2 Simulation functions overview

Both functions use buy and sell methods, also both algorithms always sell/buy 100% of stock. Algorithm that might determine amount of stock to sell is not subject of this project

[ ]:
```
def buy(actions, money, rate, price):
    amount = int(rate * money / price)
    return actions + amount, money - amount * price


def sell(actions, money, rate, price):
    amount = int(actions * rate)
    return actions - amount, money + price * amount
```

1.2.1 Simple decision function

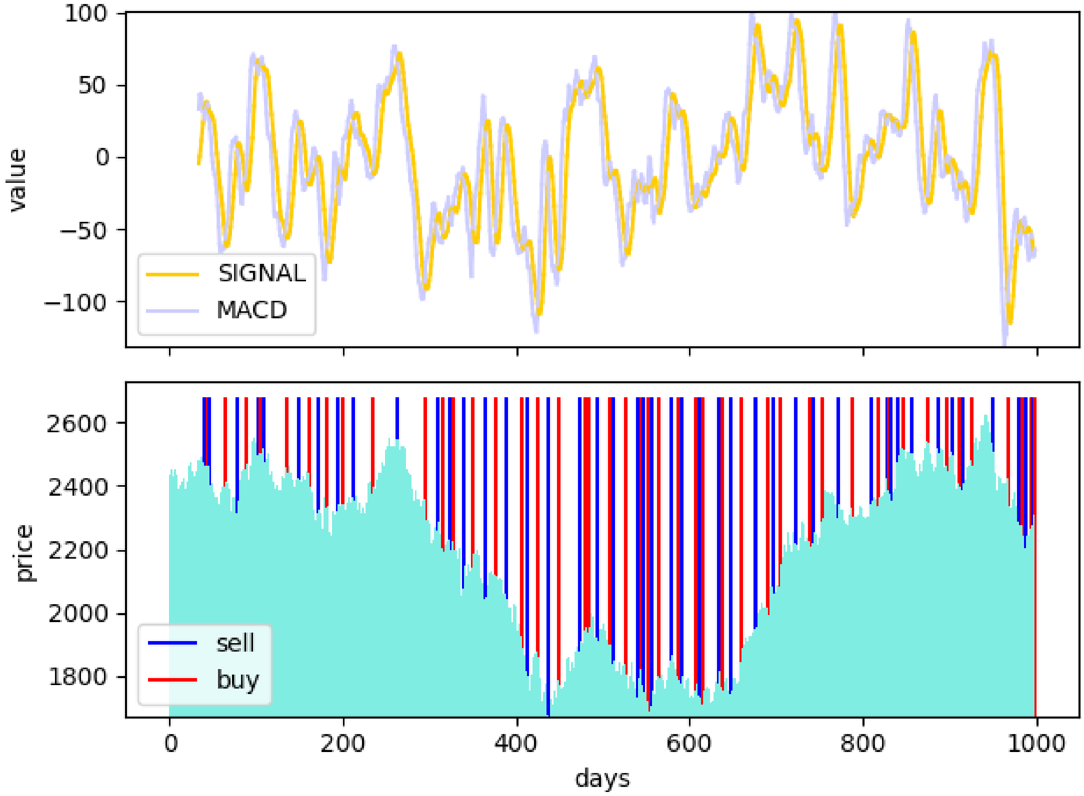
First (simple) function predicting buy / sell moment is based on following rule: point where $MACD$ and $SIGNAL$ intersect is the right moment to buy if $MACD$ crosses $SIGNAL$ from above and right moment to sell otherwise.

If plots intersect difference between their values changes its sign. At intersection point a line indicating right moment to buy / sell is drawn.

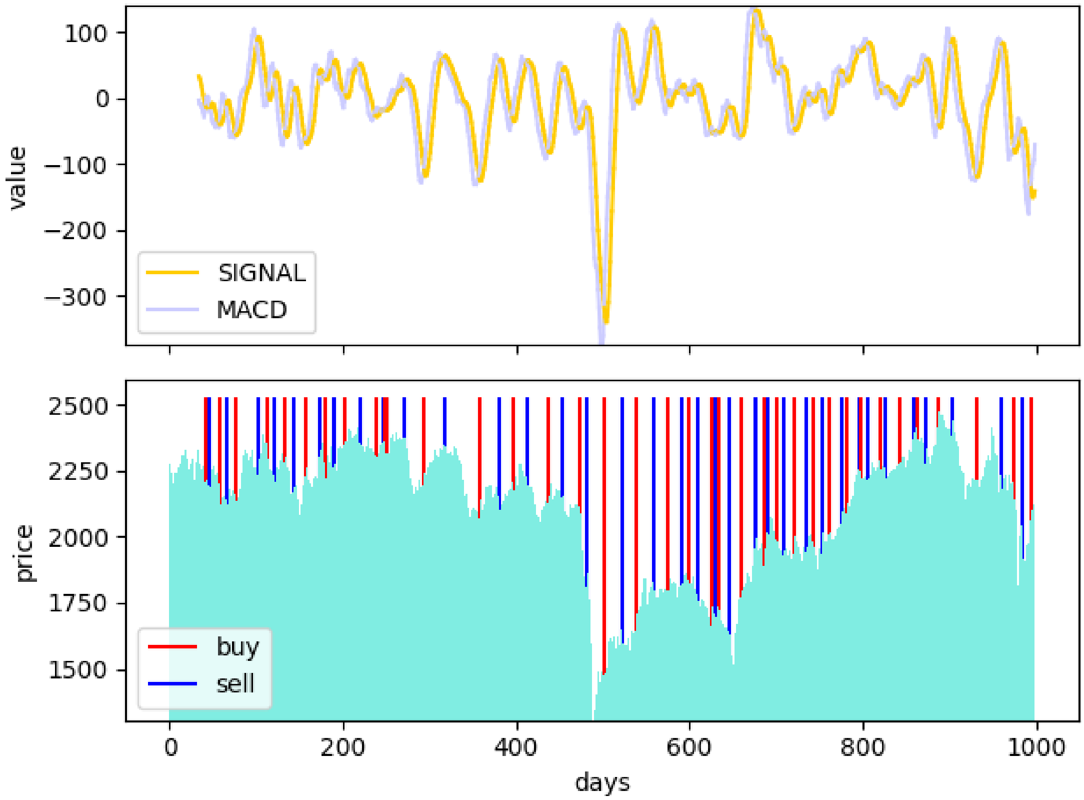
Function returns amounts of actions and money and difference between $MACD$ and $SIGNAL$ in current iteration.

[ ]:
```
def basicSim(actions, amount, ax, i, macd_arr, money, prev_dif, signal_arr,␣
 ↪y_max_1, y_min_1):
    dif = signal_arr[-1] - macd_arr[-1]
    if np.sign(dif) != np.sign(prev_dif) and i > 35:
        if dif - prev_dif > 0:
            ax[1].vlines(i, y_min_1, y_max_1, colors='b', zorder=1,␣
 ↪label="sell")
            actions, money = sell(actions, money, amount, data_vector[i])
        else:
            ax[1].vlines(i, y_min_1, y_max_1, colors='r', zorder=1, label="buy")
            actions, money = buy(actions, money, amount, data_vector[i])
    return dif, actions, money
```

Plots and results below are generated for two different datasets of length 1000.




```
start value: 2395550.0, end value : 1881655.0899999999, diff: -513894.91000000015 ( 78.5479363820417  % of start value)
```




```
start value: 2395550.0, end value : 1857823.1499999962, diff: -537726.8500000038 ( 77.55309427897544  % of start value)
```

1.2.2 Advanced decision function

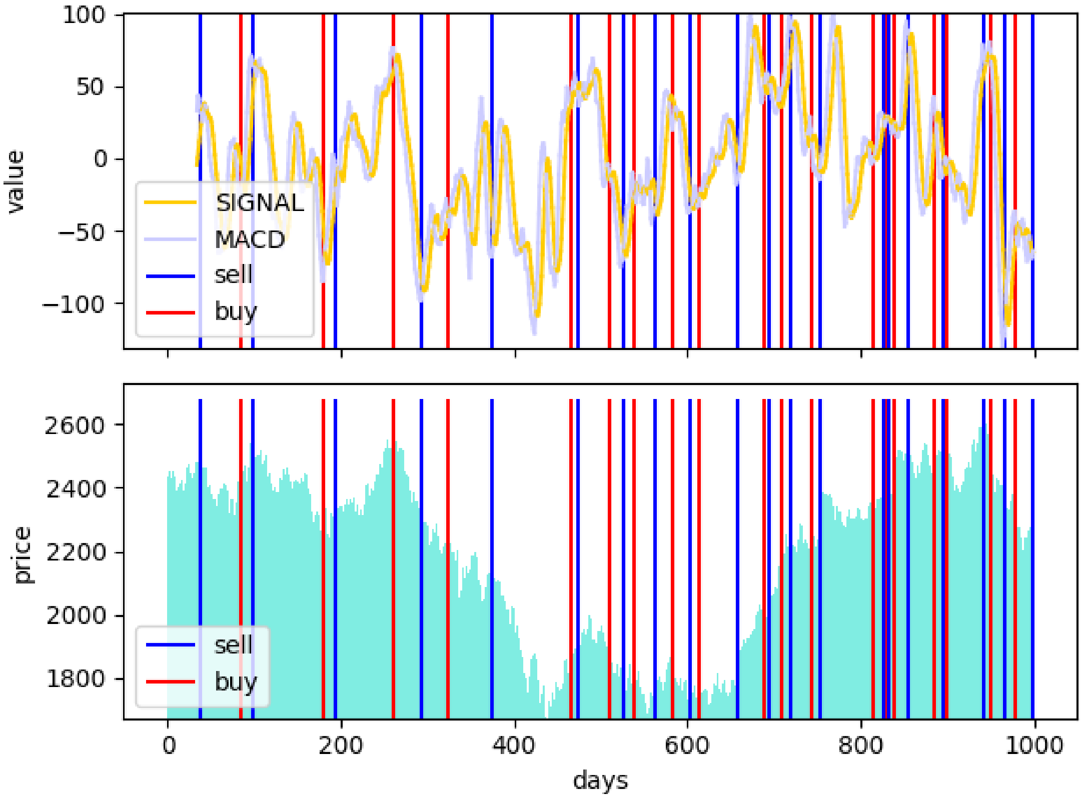
Second, more complicated decision function is based on the same assumptios, but instead of buying / selling in the exact moment of intersection it tries to predict whether intersection will soon occur and buy / sell before it.

Function works under assumption that purchase and sale alternate (e.g. there aren't 2 sales in row)

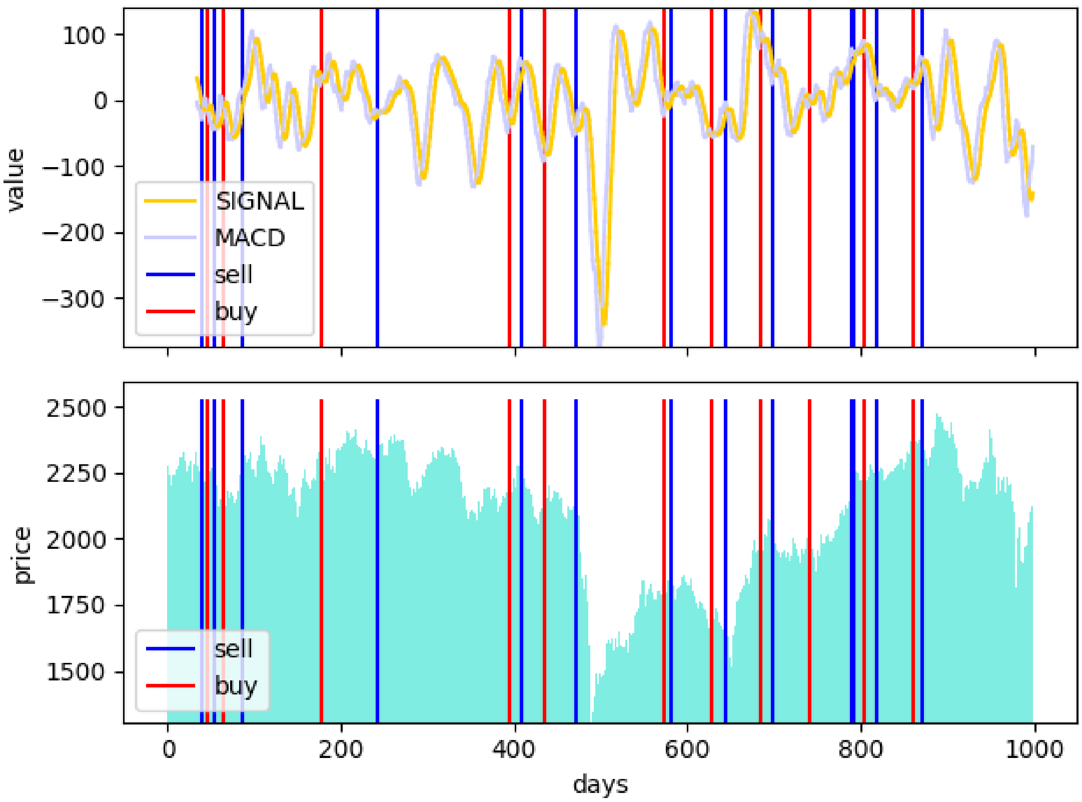
```
[ ]: def advancedSim(actions, amount, ax, dist, i, last_action, macd_arr, money,␣
     ↪signal_arr, tau, y_max_1, y_min_1):
         what = predict(macd_arr, signal_arr, dist, tau, plot=False, ax=ax[0],␣
     ↪point=i)
         if last_action != "buy" and what == "buy":
             ax[1].vlines(i, y_min_1, y_max_1, colors='r', zorder=1, label="buy")
             ax[0].vlines(i, -1000000, 1000000, colors='r', zorder=1, label="buy")
             actions, money = buy(actions, money, amount, data_vector[i])
             last_action = "buy"

         elif last_action != "sell" and what == "sell":
             ax[1].vlines(i, y_min_1, y_max_1, colors='b', zorder=1, label="sell")
             ax[0].vlines(i, -1000000, 1000000, colors='b', zorder=1, label="sell")
```

```
            actions, money = sell(actions, money, amount, data_vector[i])
            last_action = "sell"
    return last_action, actions, money
```

Plots and results below are generated for two different datasets of length 1000.




```
start value: 2395550.0, end value : 2740642.56, diff: 345092.56000000006 ( 114.40556698879172  % of start value)
```




```
start value: 2395550.0, end value : 3226596.860000001, diff: 831046.8600000008 ( 134.6912759074117  % of start value)
```

1.2.3 Prediction

Prediction is made with usage of locally weighted linear regression algorithm that fits a straight line to data based on selected point and weights relative to it. A prediction is made for latest known data point (to algorithm), what, with properly choosen τ parameter, leads to accurate prediction of intersection point between $MACD$ and $SIGNAL$.

(Regression algorithm is described later)

Predict function takes two parameters :

dist - distance describing how far away a prediction might be from current data point in order to algorithm label it as likely to occur

tau - a parameter for regression that determines rate at which wights drop when moving away from selected point

In shown below examples theese parameters were choosen experimentally

[ ]:
```python
def predict(macd, signal, dist, tau, plot=False, ax=None, point=34,␣
    ↪s_color='g', m_color='g', pred_color='g'):
```

Firstly a equal length of both vectors needs to be assured.

(also if vectors are larger than 100 elements they are clipped, for more efficient computation)

```
[ ]:     width = min(len(macd), len(signal))
        if width > 100:
            width = 100
        m = macd[-width:]
        s = signal[-width:]

        m_b, m_a = fit(m, tau)
        s_b, s_a = fit(s, tau)
        if s_a == m_a:
            return "pass"
```

Intersection point of graphs is given by

$$x_{intersection} = \frac{b_{MACD} - b_{SIGNAL}}{a_{SIGNAL} - a_{MACD}}$$

gwhere $a,\ b$ are paremeters of linear functions that approximate $MACD$ and $SIGNAL$

```
[ ]:    cross = (m_b - s_b)/(s_a - m_a)
```

If intersection has larger x coordinate than point for which a prediction is done and intersection is closer that ***dist*** , which function is rising/falling faster needs to be determined, and appropriate result is returned.

Eventually, depending on parameter ***plot*** a line that marks predicted intersection point is drawn (and also segments approximating $MACD$ and $SIGNAL$ from current data point to that point)

```
[ ]:    if cross >= width and cross - width <= dist:
            # plotting MACD and SIGNAL approximated by regression
            if plot:
                plot_prediction(m, s, m_a, m_b, s_a, s_b, dist)
                point_y = point - 34
                f_y_s_1 = s_a * point_y + s_b
                f_y_s_2 = s_a * (point_y + cross - width + 1) + s_b
                f_y_m_1 = m_a * point_y + m_b
                f_y_m_2 = m_a * (point_y+(cross - width) + 1) + m_b
                ax.plot([point, point+(cross - width) + 1], [f_y_s_1, f_y_s_2],␣
    ↪color=s_color, zorder=100)
                ax.plot([point, point +(cross - width) + 1], [f_y_m_1, f_y_m_2],␣
    ↪color=m_color, zorder=100)
                ax.vlines(point+(cross - width) + 1, -1000000, 1000000,␣
    ↪colors=pred_color, zorder=1,
                          label="predicted intersection ")
            if abs(s_a) > abs(m_a):
                return "sell"
            else:
                return "buy"
        else:
            return "pass"
```

1.2.4 Locally weigthed linear regression

In classical linear regression, a cost fuction is given by

$$J(\Theta) = \sum_{i=1}^{m}(y^{(i)} - \Theta^T x^{(i)})^2$$

It needs to be modified in order to take weights into account

$$J(\Theta) = \sum_{i=1}^{m} w^{(i)}(y^{(i)} - \Theta^T x^{(i)})^2$$

Function determining i-th element weight w.r.t x can be defined as a bell-shaped curve with standard deviation τ and mean x

$$w^{(i)} = exp(-\frac{(x^{(i)} - x)^2}{2\tau^2}) = exp(-\frac{(x^{(i)} - x)^T (x^{(i)} - x)}{2\tau^2})$$

Function below generates weight matrix for every x w.r.t ***point***

[ ]:
```
def weight_matrix(point, x, tau):
    n = x.shape[0]
    ret = np.eye(n)
    for i in range(n):
        ret[i, i] = np.exp(((x[i] - point).dot(np.transpose(x[i] - point))) /␣
 ↪(-2 * tau * tau))
    return ret
```

A classical linear regression can be fit using normal equation with Θ given by :

$$\Theta = (X^T X)^{-1}(X^T y)$$

The same method applies to locally weighted case

$$\Theta = (X^T W X)^{-1}(X^T W y)$$

where W is weight matrix

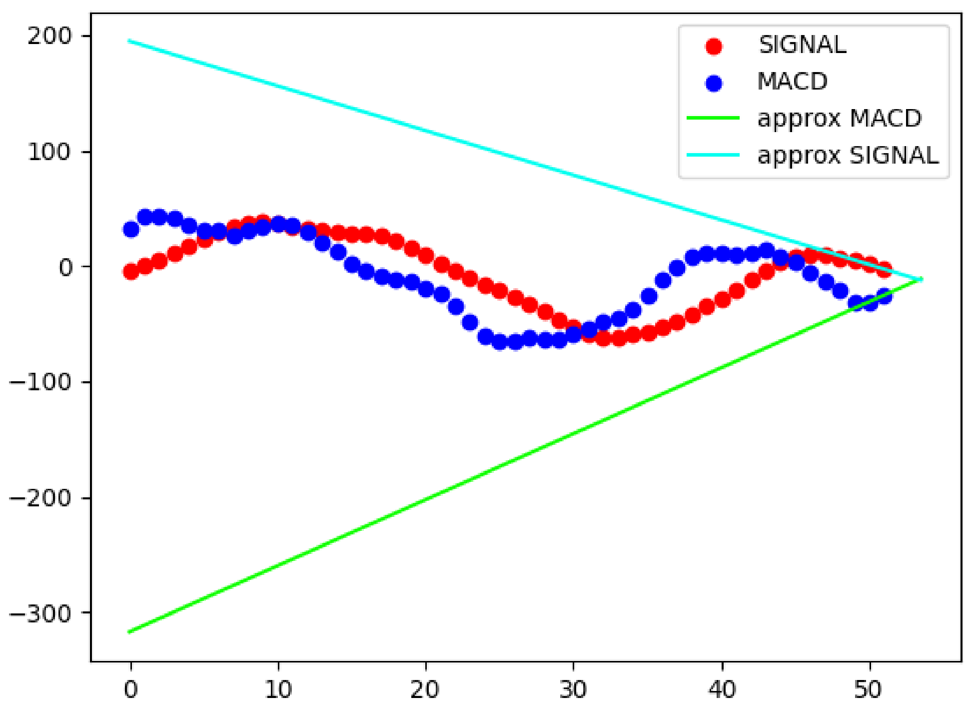
In this case as point with the highes weight we take last point in x vector

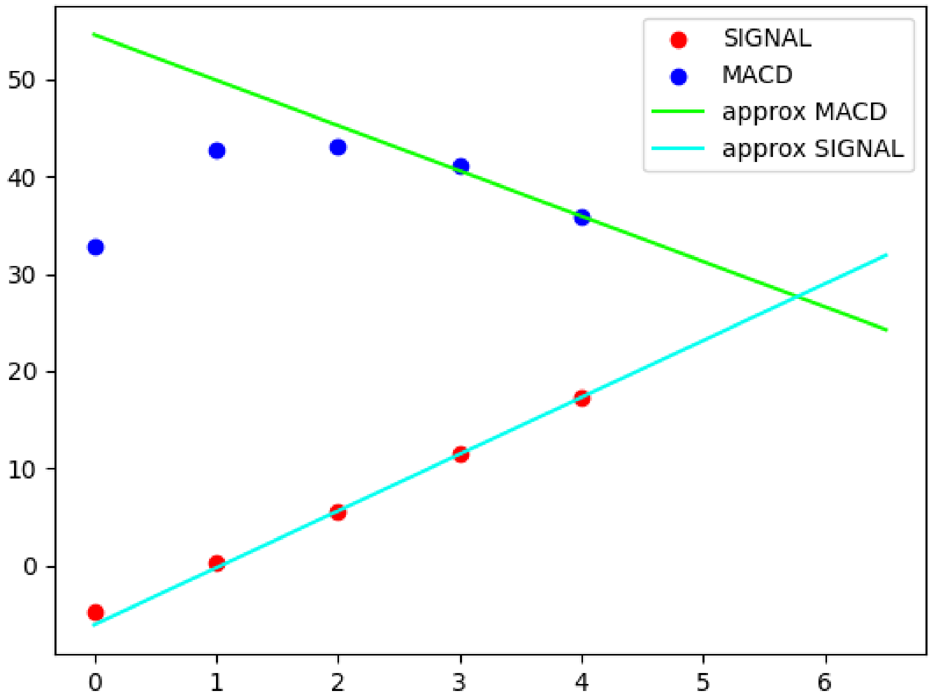
[ ]:
```
def fit(y, tau):
    n = y.shape[0]
    x = np.arange(0, n)

    # add ones in order to be able to compute bias term
    x_a = np.append(np.ones(n).reshape(n, 1), x.reshape(n, 1), axis=1)
    point = np.array([1, n - 1])
    w = weight_matrix(point, x_a, tau)
```

```
    theta = np.linalg.pinv(np.transpose(x_a).dot(w.dot(x_a))).dot(np.
 ↪transpose(x_a).dot(w.dot(y)))

    # b, a in y = ax + b
    return theta[0], theta[1]
```

[ ]:
```
def plot_prediction(m, s, m_a, m_b, s_a, s_b, dist):
    # zakładamy równą długość wektorów
    x_end = len(m) + dist
    x = np.arange(0, len(m))
    plt.figure()
    plt.scatter(x, s, color='#ff0000')
    plt.scatter(x, m, color='#0000ff')
    plt.plot([0, x_end], [m_b, m_a*x_end + m_b], color='#0fff00')
    plt.plot([0, x_end], [s_b, s_a*x_end + s_b], color='#00fff0')
```

Plots for two exaplmary approximations

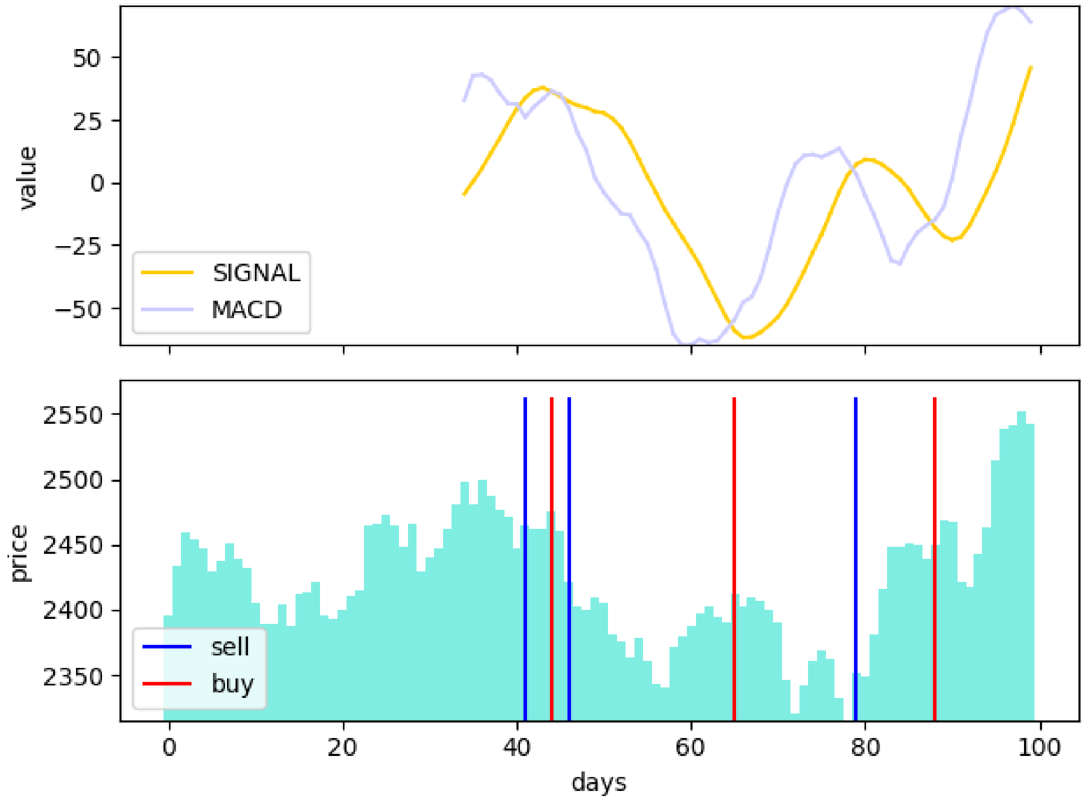




1.3 Evaluation of algorithms

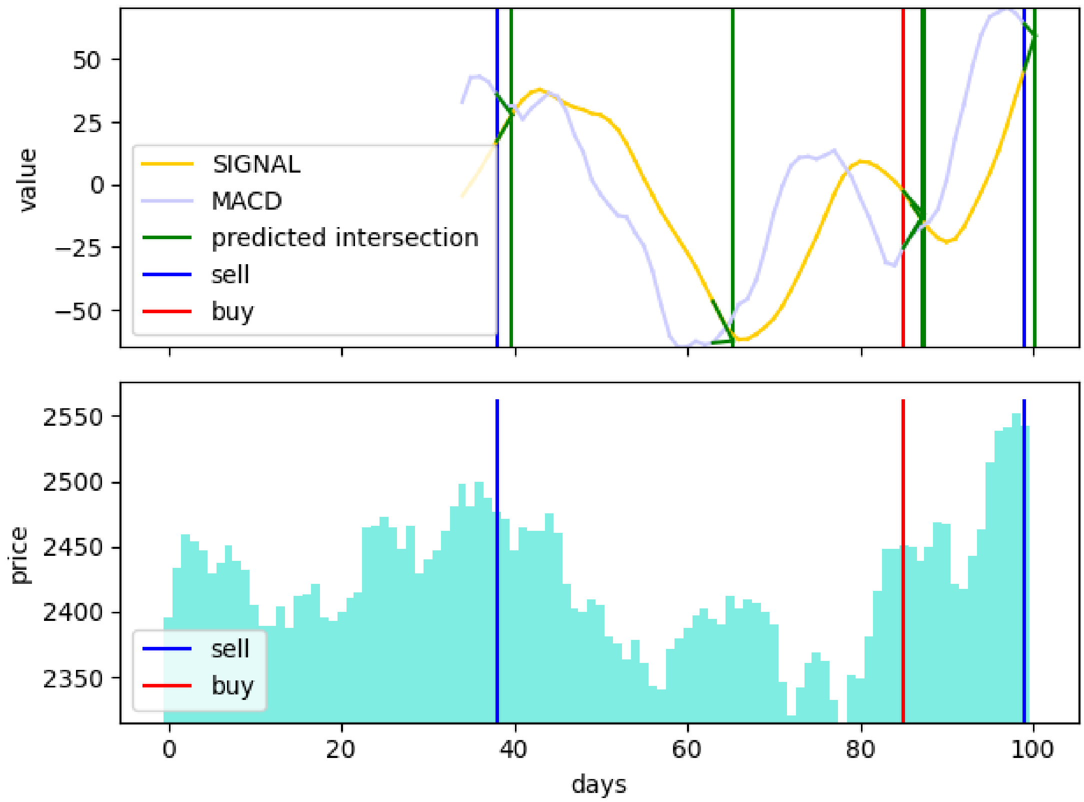
Plots for 100 element data along with simulation results are shown below (for better readability)

Basic algorithm.




```
start value: 2395550.0, end value : 2034850.12, diff: -360699.8799999999 ( 84.94291999749535  % of start value)
```

More advanced algorithm (with predictions plotted).




```
start value: 2395550.0, end value : 2569128.2, diff: 173578.2000000002 ( 107.24586003214294  % of start value)
```

A conclusion is that predicting algorithm manages to score a little gain both in short and long term (but only with correctly choosen parameters, that is another problem, maybe solution can be found with another machine learning algorithm). Basic algorithm can score gains only if data is constantly growing, in other case it looses money.