Πανεπιστήμιο Ιωαννίνων-Τμήμα Μηχανικών Υπολογιστών

Ακαδήμαϊκο έτος 2021-2022

1η Εργασία Βελτιστοποίησης

Αγαμέμνων Κυριαζής 4400

Περγαμηνέλης Χρήστος 4474

Περιεχόμενα

Μερικά λόγια για την άσκηση

Αρχικά, αυτό που ζητείται στην άσκηση είναι οι βέλτιστες τιμές των βαρών που αντιστοιχίζονται στην κάθε τιμή του κάθε forecaster για τις χρονικές στιγμές 1 έως 100. Στην συνέχεια χρησιμοποιούμε τις μ τελευταίες τιμές για να λάβουμε ένα διάνυσμα βαρών το οποίο αξιολογούμε με βάση τις τελευταίες 20 τιμές της χρονοσειράς. Για να ξεκινήσουμε τον υπολογισμό των βαρών με την χρήση των αλγορίθμων πρέπει να περάσουμε για αρχή τους forecasters. Στο αρχείο forecast.py περνάμε τους forecasters όπως φαίνονται και παρακάτω.

```
class Forecaster:

    def __init__(self, y_array):
        self.y_array = np.array(y_array)
        self.y_hat_s = y_array[0]
        self.y_hat_l = y_array[0]
        self.Lt = 0
        self.Rt = 0
        return

    def simple_moving_average(self, xi, t):
        return 1/len(self.y_array[max([0, t-xi]):t]) * sum(self.y_array[max([0, t-xi]):t])

    def linear_exponential_smoothing(self, alpha, t):
        self.y_hat_l = alpha * self.y_array[t] + (1 - alpha) * self.y_hat_l
        return self.y_hat_l

    def simple_exponential_smoothing(self, alpha, beta, t):
        lt = self.Lt
        self.Lt = alpha * self.y_array[t] + (1 - alpha) * (self.Lt + self.Rt)
        self.Rt = beta * (self.Lt - lt) + (1 - beta) * self.Rt
        self.y_hat_s = self.Lt + self.Rt
        return self.y_hat_s
```

Το αρχεία main.py περιέχει την συνάρτηση f που είναι η συνάρτηση σφάλματος και το αρχείο optimization.py περιέχει τις μεθόδους που αναλύονται παρακάτω.

Το αρχείο main περιέχει επιπλέον το seed το οποίο είναι το random.seed(10)

Όταν τρέξετε το αρχείο main θα σας εμφανιστούν 3 νούμερα με τα ονόματα των αλγορίθμων. Πατώντας το αντίστοιχο νούμερο ξεκινάει να τρέχει ο αντίστοιχος αλγόριθμος.

Εξήγηση κώδικα αλγορίθμου BFGS με wolf conditions line search

Στο αρχείο optimization, λοιπόν, υπάρχουν οι συναρτήσεις steepest descent (που υλοποιήσαμε για να τεστάρουμε αρχικά την line search), bfgs, newton, dogleg, trust region και bfgs trust region στην κλάση Gradient Based Algorithms και οι συναρτήσεις init, phyobj, dphyobj, zoom, line search στην κλάση LineSearch.

Για την συνάρτηση bfgs της κλάσης Gradient Based Algorithms υλοποιήσαμε για αρχή την κλάση Line search της οποίας οι συναρτήσεις καλούνται στην bfgs. Αρχικά οι phyobj και dphyobj ορίζουν την συνάρτηση φ και φ' που αξιοποιεί η συνάρτηση zoom

```
def phyobj(self, xk, a, pk):
    return self.fobj(xk + np.multiply(a, pk))

def dphyobj(self, xk, a, pk):
    h = 10 ** (-16)
    return (self.phyobj(xk, a + h, pk) - self.phyobj(xk, a, pk)) / h
```

(Για την dphyobj επειδή χρειαζόμαστε την παράγωγο σε συγκεκριμένο σημείο βρήκαμε την παράγωγο της φ με βάση τον ορισμό της παραγώγου

$\varphi'(\alpha) = \lim_{h \to 0} \frac{\varphi(\alpha+h)-\varphi(\alpha)}{h}$)

```
def zoom(self, a_low, a_high, xk, pk):
    phy_0 = self.phyobj(xk, 0, pk)
    dphy_0 = self.dphyobj(xk, 0, pk)
    aj = 0
    k = 1
    while k <= 50:
        aj = (a_low + a_high) / 2
        phy_aj = self.phyobj(xk, aj, pk)

        if phy_aj > phy_0 + (self.c1 * aj * dphy_0) or phy_aj >= self.phyobj(xk, a_low, pk):
            a_high = aj
        else:
            dphy_aj = self.dphyobj(xk, aj, pk)

            if np.abs(dphy_aj) <= -self.c2 * dphy_0:
                return aj

            if dphy_aj * (a_high - a_low) >= 0:
                a_high = a_low

            a_low = aj
        k = k+1
    return aj
```

Στη συνέχεια η συνάρτηση line_search καλεί την zoom όπως φαίνεται παρακάτω

```
def line_search(self, xk, a1, pk, a_max):
    a0 = 0
    i = 1
    while True:
        phy_a1 = self.phyobj(xk, a1, pk)
        phy_0 = self.phyobj(xk, 0, pk)
        dphy_0 = self.dphyobj(xk, 0, pk)

        if phy_a1 > phy_0 + self.c1 * a1 * dphy_0 or (phy_a1 >= self.phyobj(xk, a0, pk) and i > 1):
            return self.zoom(a0, a1, xk, pk)

        d_phy_a1 = self.dphyobj(xk, a1, pk)
        if np.abs(d_phy_a1) <= -self.c2 * dphy_0:
            return a1

        if d_phy_a1 >= 0:
            return self.zoom(a1, a0, xk, pk)

        a0 = a1
        a1 = (a1 + a_max) / 2
        i = i + 1
```

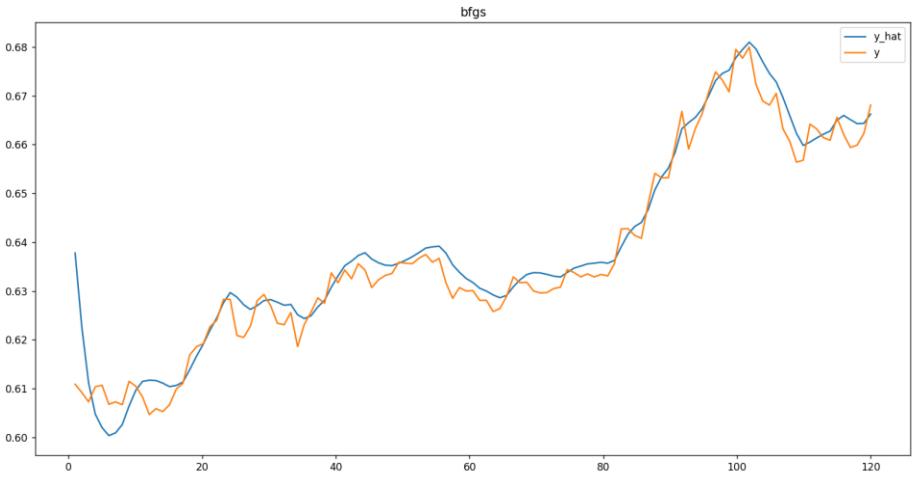
Τέλος, η συνάρτηση bfgs καλεί την συνάρτηση line_search όπως φαίνεται και στον κώδικα παρακάτω.

```
def bfgs(self, xk, Hk, epsilon, nstep):
    k = 1
    ln = LineSearch(self.fobj, 10 ** (-4), 0.9)
    Hk = np.linalg.inv(Hk)
    while np.linalg.norm(self.gradfobj(xk)) >= epsilon and k <= nstep:
        pk = -np.matmul(Hk, self.gradfobj(xk))
        a_step = ln.line_search(xk, 0.5, pk, 1)
        xk_old = xk
        xk = xk + np.multiply(a_step, pk)

        sk = xk - xk_old
        if sk.all() == 0:
            return [k, xk, self.fobj(xk)]

        yk = np.array(self.gradfobj(xk)) - np.array(self.gradfobj(xk_old))
        rk = 1/np.matmul(np.transpose(yk), sk)
        temp = np.multiply((1 - (rk * np.matmul(sk, np.transpose(yk)))), Hk)
        temp = np.multiply(temp, (1 - (rk * np.matmul(yk, np.transpose(sk)))))
        Hk = temp + np.multiply(rk, np.matmul(sk, np.transpose(sk)))

        k = k+1
    return [k, xk, self.fobj(xk)]
```

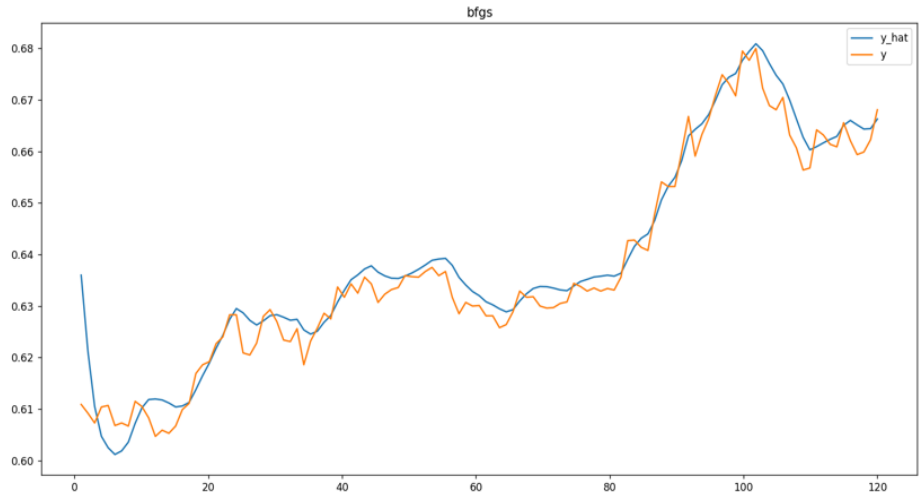
Διαγράμματα αλγορίθμου BFGS με wolf conditions line search

Στα παρακάτω διαγράμματα παρατηρούμε την συνάρτηση των πραγματικών τιμών (πορτοκαλί) στο ίδιο διάγραμμα με τις τιμές που προέβλεψε ο αλγόριθμος bfgs με wolf conditions line search (μπλε). Πήραμε επίσης την πρωτοβουλία να μην εκτυπώσουμε μόνο τα βάρη τις τιμές N=101 έως 120 άλλα θεωρήσαμε ενδιαφέρον να εκτυπωθούν και για τις προηγούμενες τιμές της χρονοσείρας για να δούμε σε σχέση και με αυτές τις τιμές πως συγκρίνονται τα επιλεγόμενα βάρη (τιμές N=1 έως 100 και 101 έως 120 που είναι το ζητούμενο της άσκησης)

Για μ=10:



Για μ=5:



Παρατηρούμε ότι για μ=5 και μ=10 ο αλγόριθμος κάνει περίπου τις ίδιες προσεγγίσεις.

Εξήγηση κώδικα αλγορίθμου BFGS με trust region

Για να υλοποιήσουμε την συνάρτηση bfgs trust region αρχικά έπρεπε να υλοποιήσουμε τις συναρτήσεις dogleg και trust region. Πρώτα υλοποιήσαμε την συνάρτηση dogleg η οποία φαίνεται παρακάτω

```
def dogleg(self, xk, Dk, Hk):
    gk = self.gradfobj(xk)
    pB = -np.matmul(Hk, gk)

    if np.linalg.norm(pB) <= Dk:
        return pB

    pU = - np.dot((np.dot(np.transpose(gk), gk) / np.dot(np.transpose(gk), np.dot(Hk, gk))), gk)
    if np.linalg.norm(pU) >= Dk:
        return -np.dot(Dk / np.linalg.norm(gk), gk)

    tau = np.divide((Dk - pU), (pB - pU))
    return pU + np.matmul((tau - 1), (pB - pU))
```

Στην συνέχεια κατασκευάσαμε την συνάρτηση trust region η οποία καλεί την dogleg

```
def trust_region(self, xk, Hk, epsilon, nstep):
    k = 1
    eta = 0.05
    Dk = 0.5
    Hk2 = Hk
    Hk = np.linalg.inv(Hk)
    while np.linalg.norm(self.gradfobj(xk)) >= epsilon and k <= nstep:
        pk = self.dogleg(xk, Dk, Hk)
        gk = np.array(self.gradfobj(xk))

        r_top = self.fobj(xk) - self.fobj(xk + pk)
        r_bot = -(np.dot(np.transpose(pk), gk) + 0.5 * np.dot(np.transpose(pk), np.dot(Hk2, pk)))
        rk = r_top/r_bot

        if rk < 1/4:
            Dk = (1/4)*Dk
        else:
            if rk > 3/4 and np.linalg.norm(pk) == Dk:
                Dk = 2*Dk
            else:
                Dk = Dk

        if rk > eta:
            xk = xk + pk
        else:
            xk = xk

        k = k+1

    return [k, xk, self.fobj(xk)]
```

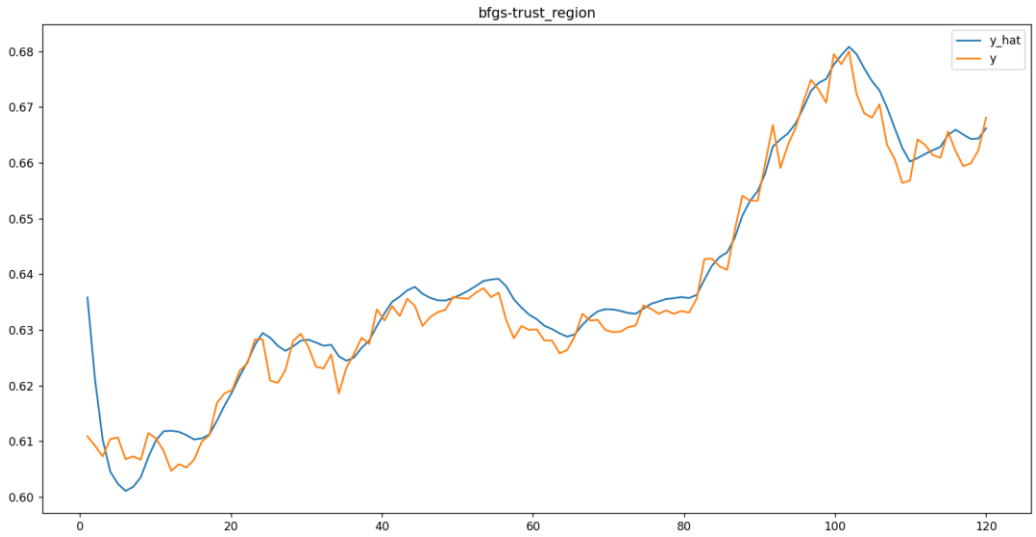
Τέλος η συνάρτηση bfgs trust region καλεί την trust region η οποία υπολογίζει ένα καινούργιο xk και το επιστρέφει στην bfgs όπως φαίνεται παρακάτω

```
def bfgs_trust_region(self, xk, Hk, epsilon, nstep):
    k = 1
    Hk2 = Hk
    Hk = np.linalg.inv(Hk)
    ln = LineSearch(self.fobj, 10 ** (-4), 0.9)
    while np.linalg.norm(self.gradfobj(xk)) >= epsilon and k <= nstep:
        xk_old = xk
        xk = self.trust_region(xk, Hk2, epsilon, nstep)[1]

        sk = xk - xk_old
        if sk.all() == 0:
            return [k, xk, self.fobj(xk)]

        yk = np.array(self.gradfobj(xk)) - np.array(self.gradfobj(xk_old))
        rk = 1/np.matmul(np.transpose(yk), sk)
        temp = np.multiply( (1 - (rk * np.matmul(sk, np.transpose(yk)))), Hk )
        temp = np.multiply(temp, (1 - (rk * np.matmul(yk, np.transpose(sk))))  )
        Hk = temp + np.multiply(rk, np.matmul(sk, np.transpose(sk)))

        k = k+1
    return [k, xk, self.fobj(xk)]
```

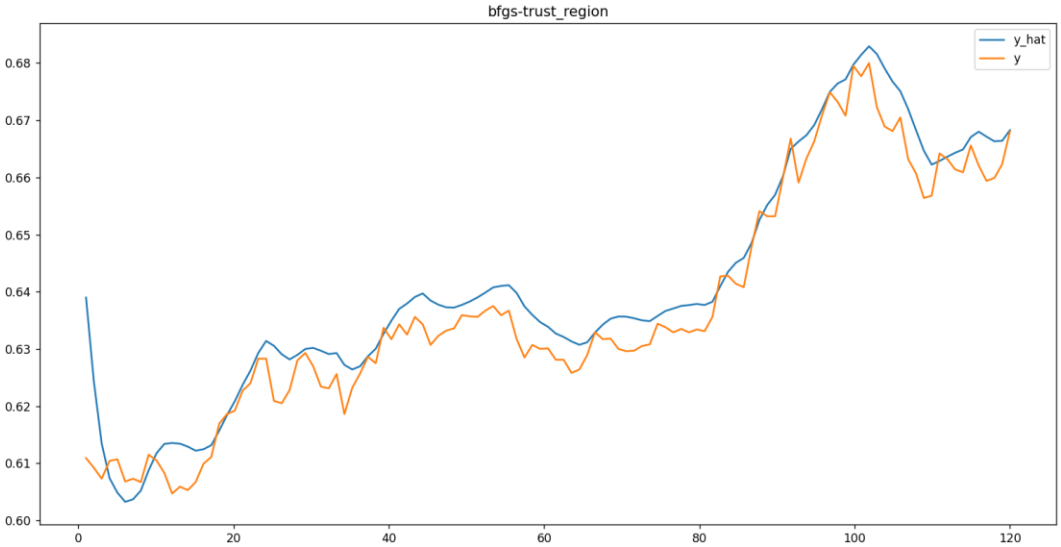
(μερικές φορές εμφανιζεταί ένα warning "RuntimeWarning: overflow encountered in double_scalars rk = r_top/r_bot" διοτι η τιμή r_bot τινει να παίρνει τιμες πολύ κοντα στο 0, μπορείτε να το αγνοήσετε)

Διαγράμματα αλγορίθμου BFGS με trust region

Για μ=5:



Για μ=10:



Παρατηρούμε ότι για μ=5 ο αλγόριθμος προσεγγίζει καλύτερα τις πραγματικές τιμές για N=101 έως N=120.

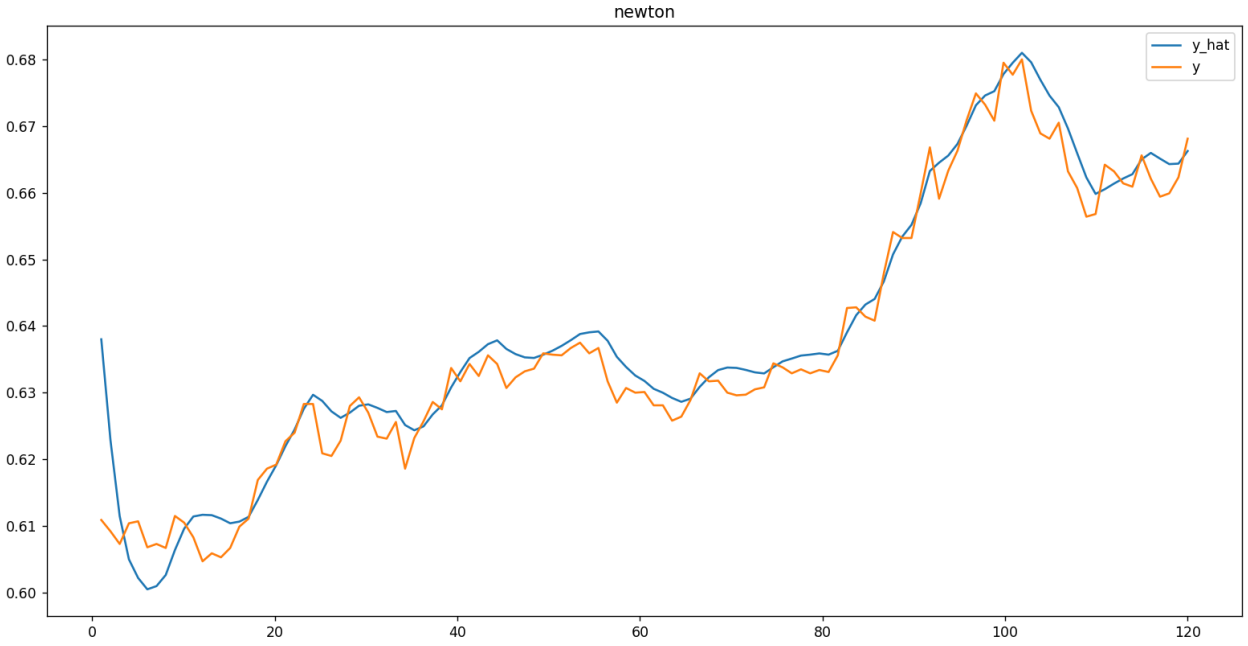
Εξήγηση κώδικα αλγορίθμου Newton με wolf conditions line search

Για να υλοποιήσουμε την συνάρτηση newton της κλάσης gradient based algorithms θα χρειαστεί να έχουν κατασκευαστεί αρχικά οι συναρτήσεις line search και zoom οπού προαναφέρθηκαν στον αλγόριθμο bfgs με wolf conditions line search. Η συνάρτηση newton καλεί την line search όπως φαίνεται παρακάτω

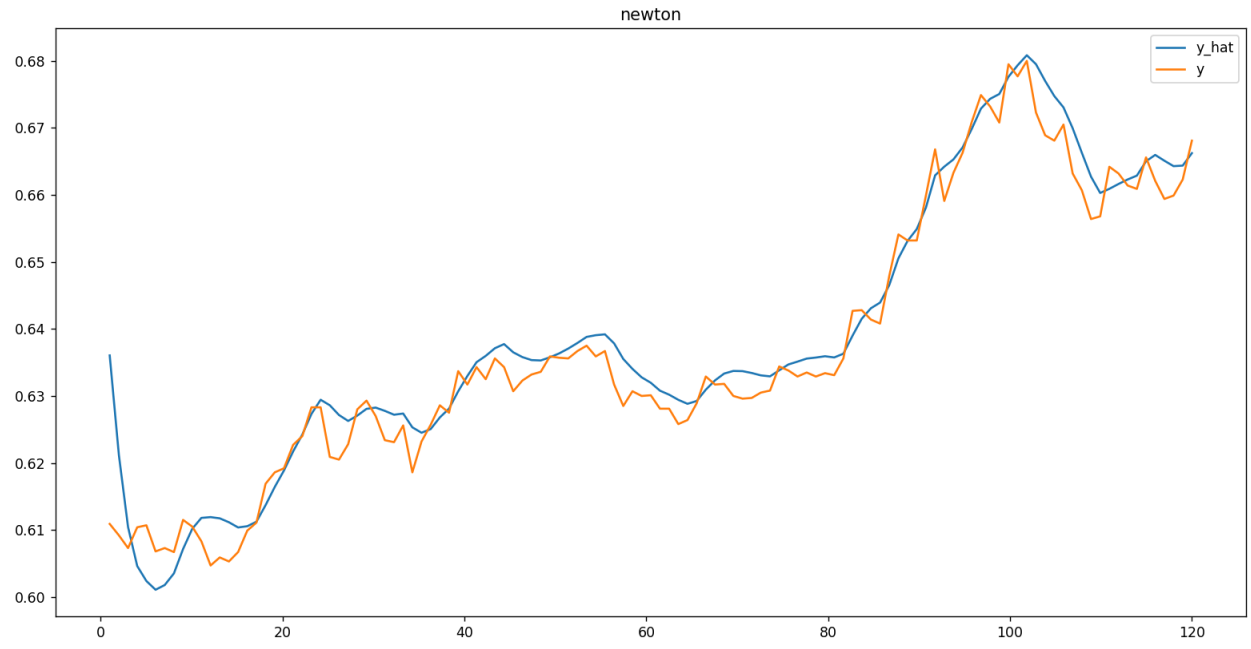
```
def newton(self, xk, Hk, epsilon, nstep):
    k = 1
    ln = LineSearch(self.fobj, 10 ** (-4), 0.9)
    Hk = np.linalg.inv(Hk)
    while np.linalg.norm(self.gradfobj(xk)) >= epsilon and k <= nstep:
        pk = -np.matmul(Hk, self.gradfobj(xk))
        a_step = ln.line_search(xk, 0.5, pk, 1)
        xk = xk + np.multiply(a_step, pk)
        k = k+1
    return [k, xk, self.fobj(xk)]
```

Διαγράμματα αλγορίθμου Newton με wolf conditions line search

Για μ=10:



Για μ=5:



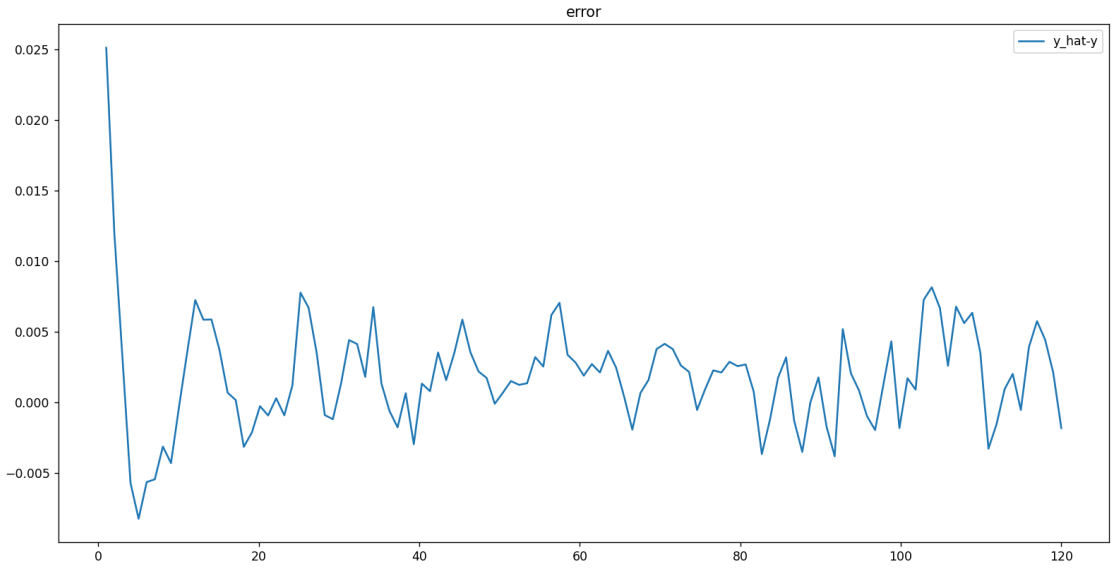
Παρατηρούμε ότι για μ=5 και μ=10 ο αλγόριθμος κάνει περίπου τις ίδιες προσεγγίσεις.

Τι παρατηρούμε

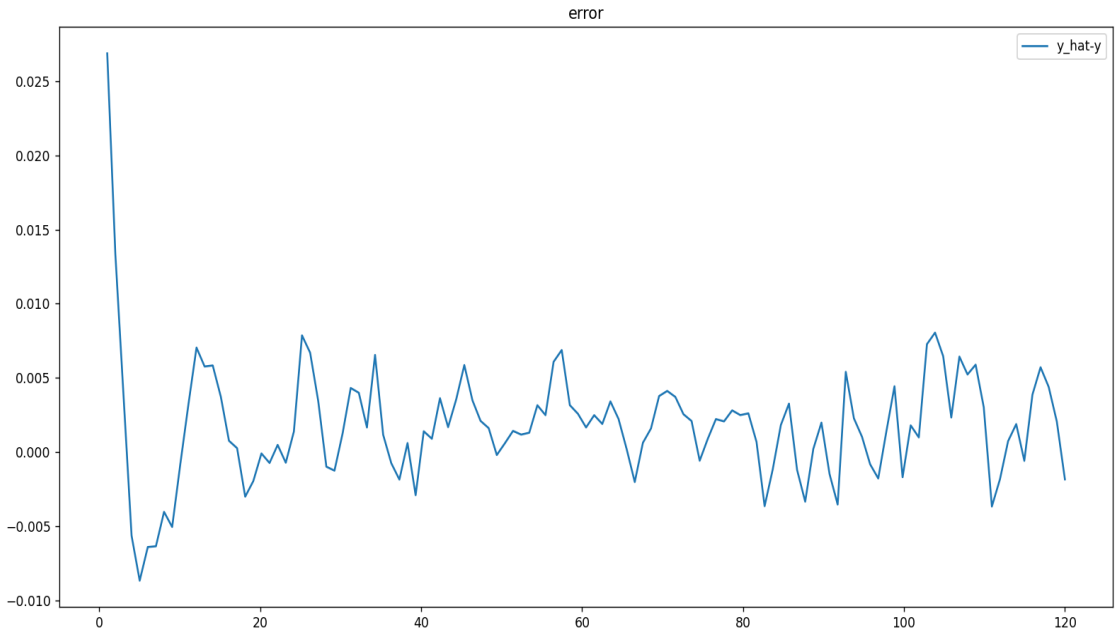
Παρακάτω βλέπουμε διαγράμματα τα σφάλματος κάθε αλγορίθμου σε σχέση με τα πραγματικά δεδομένα (y_hat-y):

Για τον αλγόριθμο bfgs με wolf conditions line search:

Για μ=5:

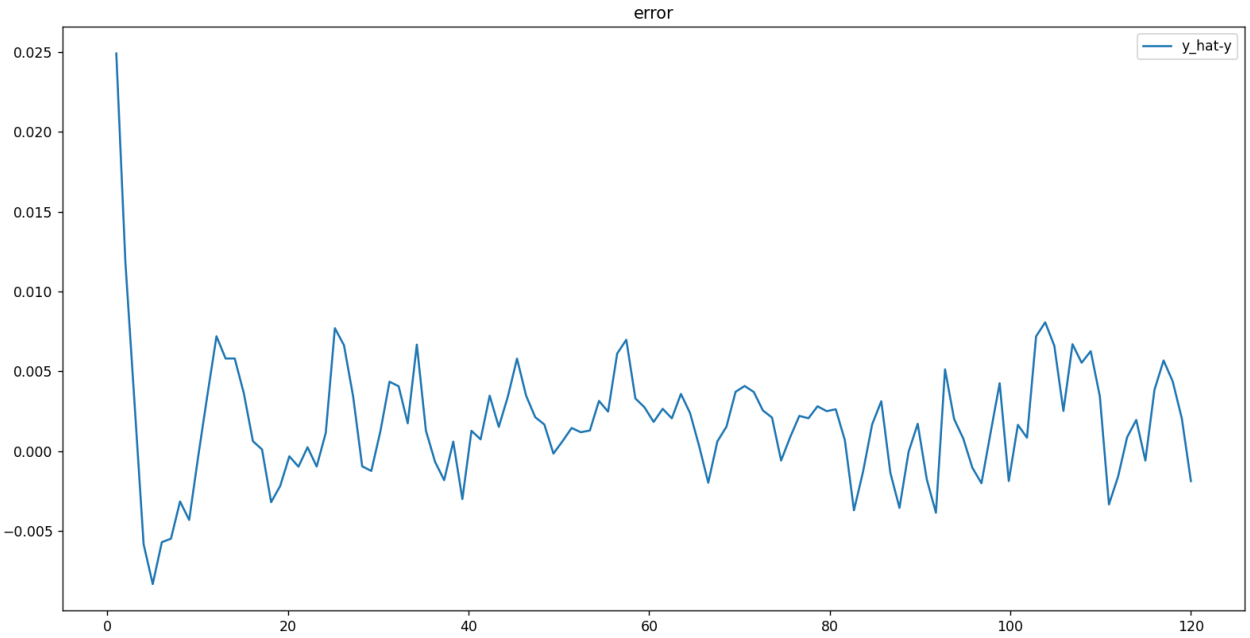


Για μ=10:

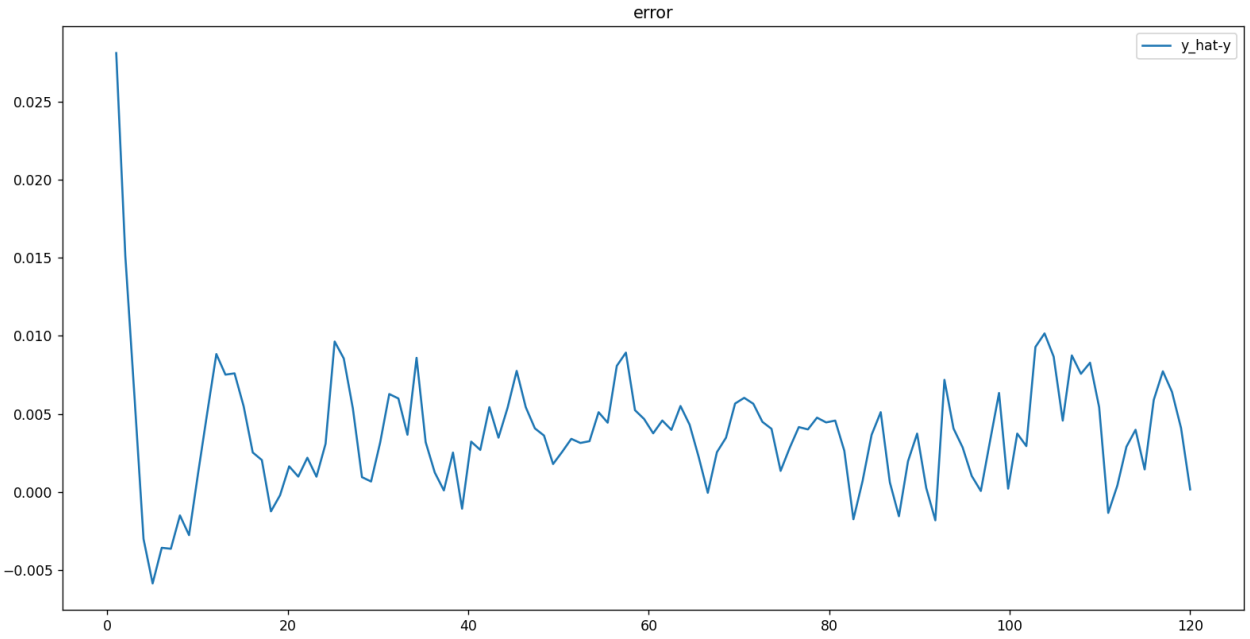


Για τον αλγόριθμο bfgs με trust region:

Για μ=5:

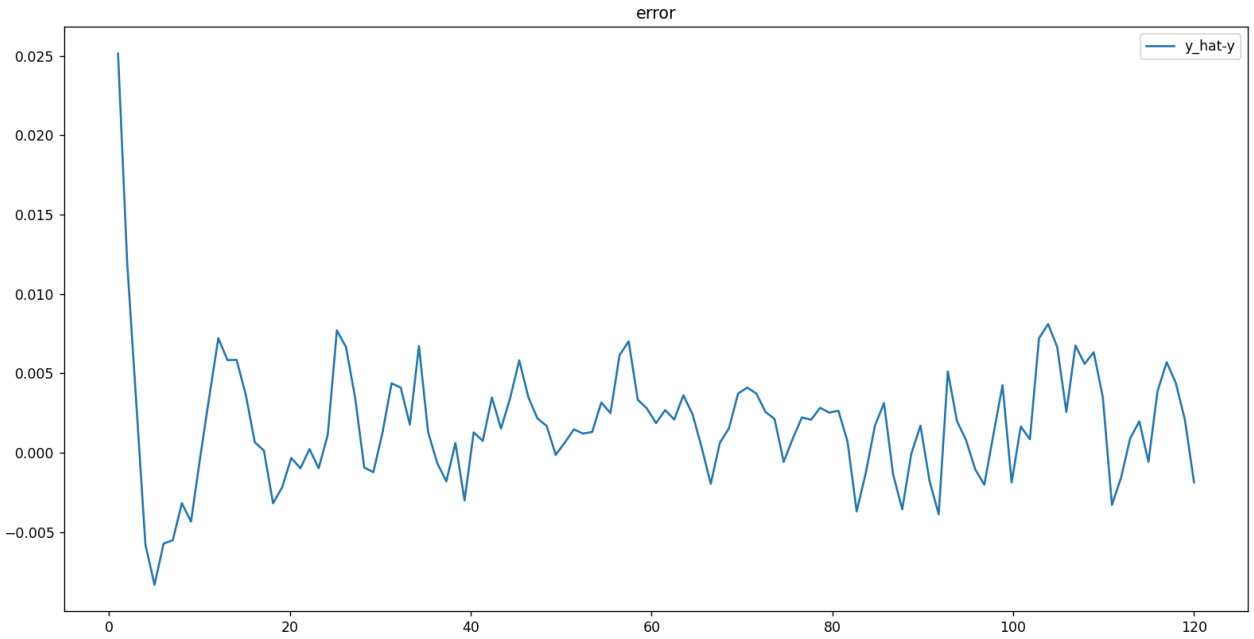


Για μ=10:

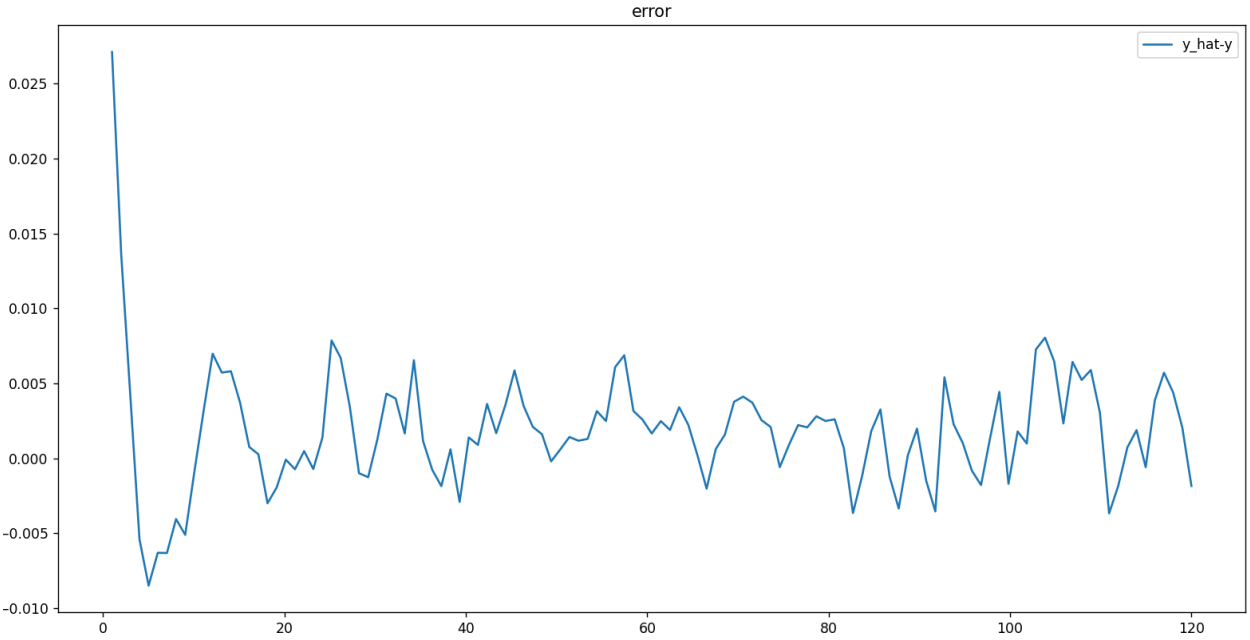


Για τον αλγόριθμο newton με wolf conditions line search:

Για μ=5:



Για μ=10:



Παρατηρούμε ότι και οι 3 αλγόριθμοι κάνουν πολύ καλή προσέγγιση των πραγματικών δεδομένων με σφάλμα που κυμαίνεται κυρίως στο 3ο δεκαδικό ψηφίο.

Τι θα μπορούσαμε να είχαμε κάνει καλύτερα

Για να έχουμε μια καλύτερη ιδέα του πόσο γρήγορά προσέγγισαν οι αλγόριθμοι τα πραγματικά δεδομένα θα μπορούσαμε να τρέξουμε για πολλές διαφορετικές αρχικές τιμές τους αλγορίθμους (πχ 1000 διαφορετικές αρχικές τιμές) και να βγάζαμε έναν μέσω χρόνο που χρειάζεται ο κάθε αλγόριθμος για να τρέξει ώστε να μπορούσαμε και χρονικά να τους συγκρίνουμε μεταξύ τους.