School of Electronics and Computer Science
University of Southampton

COMP6247(2019/20): Reinforcement and Online Learning

Kalman and Particle Filters; Online PCA

Issue	26/02 2020
Deadline	16/03 (10:00 AM) ,
Feedback by	10/06/2020

Introduction

The aim of this assignment (worth 30% of your assessment for this module) is to study online learning with supervised and unsupervised learning problems. We will learn to implment Kalman and Prticle filters and study an online Principal Component Analysis (PCA) problem. Snippets of code are provided in Appendix to help you get started, but these should not be taken as complete working programs.

Generating Synthetic Data

Generating a Simple Autoregressive Time Series

An autoregressive time series (AR Process) of order p is given by the generating model:

$$s(n) = \sum_{k=1}^{p} a_k \, s(n-k) \; + \; v(n), \tag{1}$$

where n is index over time, $a_k, \;\; k = 1, .., p$ are the parameters of the process and $v(n)$ is Gaussian random noise, usually assumed to have zero mean and varince σ_v^2.
Figure 1 shows a random excitation signal and a second order autoregressive process obtained from Eqn. (1) for which $p = 2$.

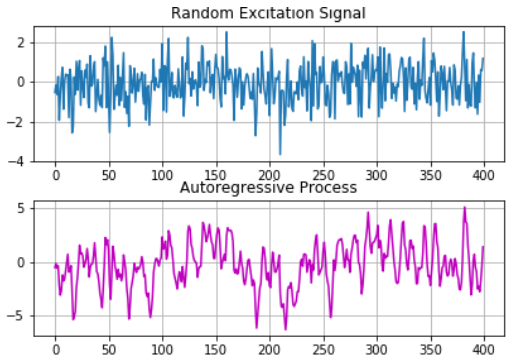

Figure 1: Random noise excitation signal and a second order autoregressive process.

Generating a Non-stationary Signal

We could study a time varying signal by slowly changing the parameters of the AR process over time. This generates a non-stationary signal. An example of slow variation in parameters and the resulting signal are shown in Fig. 2.

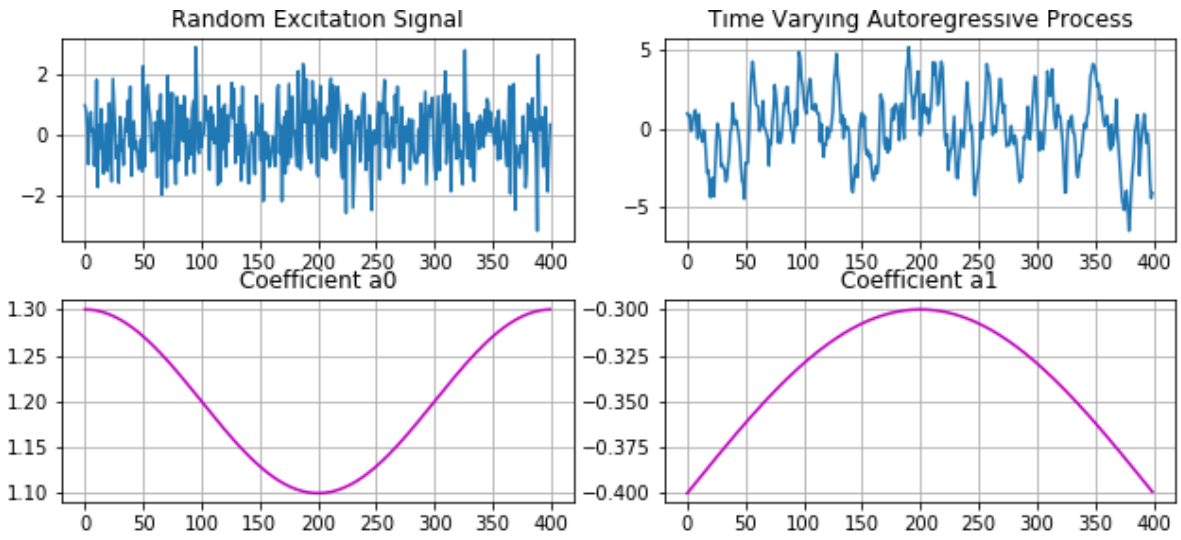

Figure 2: Slowly changing parameters of an AR process

Generating Data for Logistic Regression Classifier

If we considere a two-class classification problem in which the class conditional likelihoods are Gaussian distributed with distinct means and a common covariance matrix, the functional form of the posterior probability of class membership, computed from Bayes formula as

$$P\left[\omega_i \,|\, \boldsymbol{x}\right] \;=\; \frac{P\left[\omega_i\right]\; p\left(\boldsymbol{x} \,|\, \omega_i\right)}{\sum_{j=1}^{K}\; p\left(\boldsymbol{x} \,|\, \omega_j\right)},$$

where $\omega_j,\; j = 1, 2, ..., K$ denote K classes and $\boldsymbol{x}$ is the input feature vector has a sigmoidal form:

$$P\left[\omega_i \,|\, \boldsymbol{x}\right] \;=\; \frac{1}{1 \,+\, \exp(-\,\boldsymbol{\theta}^T\, \boldsymbol{x})}.$$

We will generate data from a two class problem in two dimensions with means set at $\left[\begin{array}{cc} -\alpha & \alpha \end{array}\right]$ and $\left[\begin{array}{cc} \alpha & -\alpha \end{array}\right]$ with common covariance marix $\left(\begin{array}{cc} 2 & 1 \\ 1 & 2 \end{array}\right)$.
This is set symmetric so there is no need to include a bias term and α could be tuned to simulate different slopes of the sigmoidal posterior. If α is set high, the means are far apart, data from the problem is linearly separable and easy to learn. Hence this could give us the flexibility of observing convergence of an extended Kalman filter at different problem compelxities.
Please see Lab Sheets of Foundations of Machine Learning for simulating such data.

Kalman Filter

A state space model for sequential estimation of parameters of a time series model is as follows:

$$\begin{aligned} \boldsymbol{\theta}(n) &= \boldsymbol{\theta}(n-1) \,+\, \boldsymbol{w}(n) \\ y(n) &= \boldsymbol{\theta}^T\, \boldsymbol{x}_n \,+\, v(n), \end{aligned}$$

where we have assumed a random walk model on the unknown parameters $\boldsymbol{\theta}$, a univariate observation and p past samples are held in the vector $\boldsymbol{x}(n) \,=\, [x_{n-1},\, x_{n-2},\, ...\, x_{n-p}]$. At every step in time, our task is to make the best estimate $\boldsymbol{\theta}$ from observation $y(n)$ and input $\boldsymbol{x}(n)$.

The Kalman filter equtions for the above are given below. For the random walk, our predictions of the state and uncertainty on the state (error covariance matrix) are:

$$\begin{aligned} \boldsymbol{\theta}(n-1|n) &= \boldsymbol{\theta}(n-1|n-1) \\ P(n-1|n) &= P(n-1|n) + Q \end{aligned}$$

At time n, we have new data $\{\boldsymbol{x}(n),\, y(n)\}$. We predict the target signal as $\hat{y}(n) = \boldsymbol{x}(n)^T \boldsymbol{\theta}(n|n-1)$, using the predicted parameters. The innovation (or error) signal is:

$$e(n) = y(n) - \boldsymbol{x}(n)^T \boldsymbol{\theta}(n|n-1)$$

We now make posterior updates to the state estimates and the corresponding error covariance matrix:

$$\begin{aligned} \boldsymbol{\theta}(n|n) &= \boldsymbol{\theta}(n|n-1) + \boldsymbol{k}(n)\, e(n) \\ P(n|n) &= \left(I - \boldsymbol{k}(n)\, \boldsymbol{x}(n)^T\right) P(n|n-1) \end{aligned}$$

where the Kalman gain $\boldsymbol{k}(n)$ is given by

$$\boldsymbol{k}(n) = \frac{R(n|n-1)\, \boldsymbol{x}(n)}{R + \boldsymbol{x}(n)^T P(n|n-1)\, \boldsymbol{x}(n)}$$

Extended Kalman Filter

We know that the Kalman filter is optimal in estimating $\boldsymbol{x}(n)$ when the models of dynamics and observations are linear and the noise processes are both Gaussian. When faced with nonlinear systems, the natural approach is to locally linearize by truncating the Taylor series expansion of the nonlinearities about the operating point. This leads to the extended Kalman filter (EKF) algorithm. In the machine learning literature, such approaches to online learning have been used effectively, for example, in training radial basis function and multi-layer perceptrons sequentially [1, 2].
The simple dynamical system we will consider to illustrate EKF algorithm for online estimation of parameters will again have a random walk state dynamics and Gaussian noise processes, but we will make the observation model nonlinear.

$$\begin{aligned} \boldsymbol{\theta}(n) &= \boldsymbol{\theta}(n-1) + \boldsymbol{w}(n) \\ y(n) &= f\left(\boldsymbol{\theta}, \boldsymbol{x}_n\right) + v(n), \end{aligned}$$

where $f(.,.)$ is a nonlinear function. With random walk dynamics imposed on $\boldsymbol{\theta}$, the unknown parameters of a model we wish to estimate, $\boldsymbol{x}(n)$ is now the input to the model.

Sequential Estimation of a Logistic Regression Model

We will use extended Kalman filter (EKF) to estimate parameters $\boldsymbol{\theta}$ of a logistic regression model defined as

$$y = \frac{1}{1 + \exp\left(-\boldsymbol{\theta}^T \boldsymbol{x}\right)}$$

in a sequential setting by propagating $\boldsymbol{\theta}$ and an error covariance matrix on its estimate sequentially (online).

Particle Filters

Particle filters (or Sequential Monte Carlo) are a family of online Bayesian inference tools in which a non-parametric density represented by samples drawn from it is propagated over time along with importance weights. In addition to several tracking and robot localization problems, these techniques have also been used in optimizing neural networks [3].
We will consider Algorithms 1 and 2 given in [5] for this exercise.

Online Principal Component Analysis

In class, we discussed the online algorithm for PCA due to Boutsidis *et al.* [4] which offers an algorithm whose reconstruction error performance is a compromise between a user specified error and a higher than required low dimensional representation, which is a penalty one pays for online processing. For the purposes of noise reduction and the convenience of working in a low dimensioanl space, we may wish to apply such an algorithm to preprocess a large dataset in the "big data" regime.

Tasks

5 Marks Following the snippets of code provided, implement Kalman filter to estimate the parameters of a synthetic second order AR process. Show how the parameters converge for five different initial conditions. Try various values for the process noise covariance and measurement noise variance and comment on their effects on convergence speeds.

10 Marks Implement a particle filter on the synthetic time varying AR problem. Show that sequential importance sampling (Algorithm 1 in [5]) suffers from degeneracy of particle weights and implement a resampling step to address this issue (Algorithm 2 in [5]).

10 Marks Write out the extended Kalman filter algorithm for the logistic regression problem described previously present it as pseudocode. You need to differentiate the logistic functionto do this step.

Generate synthetic data from a two-dimensional two-class distinct mean, common covariance problem as outlined in the section above and implement an extended Kalman filter for the estimation of its parameters.

Show in a graph how the class boundary converges to the true solution, starting from a random guess.

5 marks Briefly discuss the potential use of this algorithm on a large problem (large in terms of number of data points and/or dimensions) taken from a benchmark repository (e.g. the UCI Repository of Machine Learning Datasets or a `kaggle` competition dataset). Briefly discuss how you might specify the various parameters required by the algorithm, what low dimensional representations might result and what would the computational savings be?

Report

Upload a report of **no more than four pages** describing your work. Please make sure you include your name in the report. A report without the candidate's name on it runs the risk of attracting a mark of 0/30.

References

[1] Kadirkamanathan, V. and Niranjan, M. A Function Estimation Approach to Sequential Learning with Neural Networks,*Neural Computation* **5**(6): 954-975, 1993.

[2] Puskorius, G.V. and Feldkamp L.A., Decoupled extended Kalman filter training of feedforward layered networks, *International Joint Conference on Neural Networks (IJCNN91):* 771-777, 1991.

[3] Freitas, JFG de and Niranjan, M. and Gee, A.H. and Doucet, A., Sequential Monte Carlo methods to train neural network models, *Neural computation* **12**(4): 955-993, 2000.

[4] Boutsidis, C, Garber D, Karnin ZS and Liberty, E. Online principal components analysis, *Proceedings of the twenty-sixth annual ACM-SIAM symposium on Discrete algorithms, SODA-15*: 887–901, 2015.

[5] Arulampalam, MS, Maskell, S, Gordon, N &Tim Clapp, T. A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking, *IEEE Trans. Signal Processing* **50**(2), 174-188, 2002.

Appendix: Snippets of Code

Generating a Second Order Autoregressive Process

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

# Length of time series
#
N=400

# Gaussian random numbers as an excitation signal
#
ex = np.random.randn(N)

# Second order AR Process
#
a = np.array([1.2, -0.4])

S = ex.copy();
for n in range(2, N):
    x = np.array([S[n-1], S[n-2]])
    S[n] = np.dot(x, a) + ex[n]

fig, ax = plt.subplots(nrows=2, ncols=1, figsize=(6,4))
plt.tight_layout()

ax[0].plot(range(N), ex)
ax[0].grid(True)
ax[0].set_title("Random Excitation Signal")

ax[1].plot(range(N), S, color='m')
ax[1].grid(True)
ax[1].set_title("Autoregressive Process")
```

Generating a Time-varying AR process

```
%matplotlib inline
import numpy as np
import matplotlib.pyplot as plt

# Length of time series
#
N=400

# Gaussian random numbers as an excitation signal
#
ex = np.random.randn(N)

# Second order AR Process with coefficients slowly changing in time
#
a0 = np.array([1.2, -0.4])
A  = np.zeros((N,2))
omega, alpha = N/2, 0.1

for n in range(N):
    A[n,0] = a0[0] + alpha * np.cos(2*np.pi*n/N)
    A[n,1] = a0[1] + alpha * np.sin(np.pi*n/N)

S = ex.copy();
for n in range(2, N):
    x = np.array([S[n-1], S[n-2]])
    S[n] = np.dot(x, A[n,:]) + ex[n]

fig, ax = plt.subplots(nrows=2, ncols=2, figsize=(9,4))
plt.tight_layout()

ax[1,0].plot(range(N), A[:,0])
ax[1,0].grid(True)
ax[1,0].set_title("Coefficient a0", color='m')

ax[1,1].plot(range(N), A[:,1], color='m')
ax[1,1].grid(True)
ax[1,1].set_title("Coefficient a1", color='m')

ax[0,0].plot(range(N), ex)
ax[0,0].grid(True)
ax[0,0].set_title("Random Excitation Signal")

ax[0,1].plot(range(N), S, color='m')
ax[0,1].grid(True)
ax[0,1].set_title("Time Varying Autoregressive Process")

plt.savefig("arChange.png")
```

Kalman Filter Estimates of second order AR parameters

```
# Time series data  y
# th_n_n:  estimate at time n using all data upto time n
# th_n_n1: estimate at time n using all data upto time n-1
#

# Initialize
#
x = np.zeros((2,1))
th_n1_n1 = np.random.randn(2,1)
P_n1_n1 = 0.001*np.eye(2)

# Noise variances -- hyperparameters (to be tuned)
# Set measurement noise as fraction of data variance (first few samples)
# Guess for process noise
#
R    = 0.2*np.std(ex[0:10])
beta = 0.0001
Q    = beta*np.eye(2)

# Space to store and plot
#
th_conv = np.zeros([2, N])

# First two estimates are initial guesses
#
th_conv[0,0] = th_n1_n1[0]
th_conv[0,1] = th_n1_n1[1]
th_conv[1,0] = th_n1_n1[0]
th_conv[1,1] = th_n1_n1[1]

# Kalman Iteration Loop (univariate observation, start from time step 2)
#
for n in range(2, N):
    # Input vector contains past velues
    x[0] = y[n-1]
    x[1] = y[n-2]

    # Prediction of state and covariance
    th_n_n1 = th_n1_n1.copy()
    P_n_n1  = P_n1_n1 + Q

    yh = th_n_n1.T @ x
    en = y[n] - yh
    ePlot[n] = en

    # Kalman gain (kn) and innovation variance (den)
    #
```

```
        den = x.T @ P_n1_n1 @ x + R
        kn = P_n1_n1 @ x / den

        # Posterior update
        #
        th_n_n = th_n_n1 + kn * en
        P_n_n  = (np.eye(2) - kn @ x.T) @ P_n_n1

        # Save
        th_conv[0,n] = th_n_n[0]
        th_conv[1,n] = th_n_n[1]

        # Remember for next step
        #
        th_n1_n1 = th_n_n.copy()
        P_n1_n1  = P_n_n.copy()

print(a)
print(th_n_n)

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(8,3))
ax[0].plot(th_conv[0])
ax[0].set_xlim(0,500)
ax[0].axhline(y=a[0], color='r')

ax[1].plot(th_conv[1])
ax[1].set_xlim(0,500)
ax[1].axhline(y=a[1], color='r')
ax[1].set_title("R = %4.3f, Q = %6.5f I"%(R, beta))
```

Partile Filter Estimates of second order AR parameters

A distribution of random particles in 2D

```
Ns = 30

# Initial particles
#
parts = np.random.randn(Ns, 2)

fig, ax = plt.subplots(figsize=(3,3))
ax.scatter(parts[:,0], parts[:,1], color='b')
ax.grid(True)
ax.set_title("Initial Particles")
```

Algorithm 1: Sequential Monte Carlo

```
# Likelihood from Gaussian noise
# Needs estimate, true value and noise variance
def particle_likelihood(y0, yh, sig):
    lhood =  (1/(2*np.pi*sig))*np.exp(-0.5*(y0-yh)**2)
    return lhood

# Process noise covariance is beta*I
#
beta = 0.01

# Observation noise variance
#
sig = np.std(ex)

# Initial input vector and particle weights
#
x   = np.zeros((2,1))
pweights = np.ones((Ns,1)) / Ns

# Go over the data
#
for n in range(2, 10):
    x[0] = S[n-1]
    x[1] = S[n-2]

    # Sample particles from the prior
    # (in the simplest case here, random perturbation)
    #
    for p in range(Ns):
        parts[p,:] = parts[p,:] + beta*np.random.randn(1,2)

    # Predictions and likelihoods
    #
```

```
    sh = np.zeros((Ns, 1))
    lhoods = sh.copy()
    for particle in range(Ns):
        th = parts[p,:].T.copy()
        sh[particle] = x.T @ th
        lhoods[particle] = particle_likelihood(S[n], sh[p], sig)

    # multiply prior weigts by likelihoods and normalize
    #
    pweights = pweights * lhoods
    pweights = pweights / np.sum(pweights)

# Observe how the weights change as you update
#
    fig, ax = plt.subplots(figsize=(4,4))
    ax.bar(np.arange(len(pweights)), pweights[:,0])
```