CS 6190: Probabilistic Machine Learning Spring 2023

Midterm Report

Handed out: 30 March, 2023
Due: 11:59pm, 05 May, 2023

1. Problem definition and motivation
   The final term report is about earthquake prediction. I expect to use the dataset from Kaggle and knowledge from the class to predict the earthquake magnitude. Many countries are engaging in earthquake prediction. Especially at the beginning of this year, Turkey experienced several enormous disasters which caused many casualties. I believe there are relationships between magnitude and the dataset from the observation stops. Hence, this project aims to use machine learning to find the connection. The mid-term report also shows that the dataset and magnitude have a non-linear relationship, which motivates me to use machine learning skills to deal with the prediction.
   There are many ways to predict earthquakes, but none only use the dataset from the observation stops but not from seismicity.

2. My solution
   In this project, I built three models to solve the problem. They are linear model (Built in the mid-term report), Gaussian distribution, and Dirichlet distribution. On another side, I used Laplace approximation to get the result to make things easy because Gaussian and Dirichlet are hard to compute the evidence. Then, use mean square error as my evaluation. I used the linear model as my baseline and tried to build other models to compete with it. Overall, Dirichlet has the best performance among competitors. (Mean square error: Dirichlet 2.75/ Gaussian 9.07/ linear 9.45)
   In the linear model, I used Bayesian linear regression taught in the class. Equations are shown below:
   $p(\omega|D) \propto p(D|\omega)p(\omega)$
   $=> N(\omega|m_N, S_n) \propto N(t|\Phi\omega, \beta^{-1}I)N(\omega|m_0, S_0)$
   Then, we can get $m_N = S_N(S_0^{-1}m_0 + \beta\Phi^T t)$ and $S_N^{-1} = S_0^{-1} + \beta\Phi^T\Phi$
   Or, simplify it as $m_N = \beta S_N\Phi^T t$ and $S_N = \alpha I + \beta\Phi^T\Phi$
   The codes are shown in fig 1

```
40          """
41          SN = inv(inv(S0) + beta * Phi.T @ Phi)
42          mN = SN @ (inv(S0) @ m0 + beta * Phi @ t)
43          return SN, mN
```

Figure 1: Conjugate linear regression

In Gaussian distribution, I used factorized model taught in the class. Equations are shown below:
$p(\alpha) = Gam(a_0, b_0)$
Prior: $p(\omega|\alpha) = N(\omega|0, \alpha I)$
Likelihood: $p(t|\omega) = \Pi_{n=1}^{N} N(t_n|\omega\Phi_n, \beta^{-1})$
Joint: $p(t, \omega, \alpha) = p(t|\omega)p(\omega|\alpha)p(\alpha)$
The task I want is to get the posterior $p(\omega|t, \alpha)$. The most significant difference from the linear model is that the gamma function influences the prior. I used Laplace approximation to get the features. Here are the equations:

$f(\theta) = log(p(\theta, D))$ $f(\theta) \approx f(\theta_0) + \nabla f(\theta_0)^T(\theta - \theta_0) + \frac{1}{2}(\theta - \theta_0)A(\theta - \theta_0)$ Hence, $p(\theta|D) = N(\theta|\theta_0, A)$
In this Gaussian model, the logarithm joint is shown below:
$log(p(w, t, \alpha)) = log(p(t|\omega, \alpha)) + log(p(\omega|\alpha)) + log(p(\alpha))$
$=> log(gamma(\alpha)) + \frac{-1}{2}\omega\alpha I\omega^T + \frac{-1}{2}(t_n - \Phi_n\omega^T)\beta(t_n - \Phi_n\omega^T)$. The codes are shown in fig 2

```
def compute_log_gamma(alpha):
    """ Compute the expected value of every log pi_k, i.e., compute ∀k. E[log pi_k] """
    #return digamma(alpha) - digamma(alpha.sum())
    return np.log(gamma(alpha))
def compute_log_pw(alpha, w):
    w = w.reshape(1,8)
    #return -0.5 * w @ (alphanp.eye(8)) @ w.T
    #return np.diagonal(0.5 * (w @ (alpha*np.eye(8))).T @ w)
    return 0.5 * w @ (alpha*np.eye(8)) @ w.T

def compute_log_pt_w(w, phi, t, beta):
    return np.sum(0.5 * np.diagonal((t- phi @ w.T) * beta * ((t- phi @ w.T)).T))

def log_posterior(w, phi, t, alpha, beta,i):
    #optimize w[0]w[i]
    #print("compute_log_gamma: ",compute_log_gamma(alpha))
    #print("compute_log_pw: ",compute_log_pw(alpha, w).T)
    #print("compute_log_pt_w: ",compute_log_pt_w(w, phi, t, beta))
    result = -1 * compute_log_gamma(alpha) + compute_log_pw(alpha, w).T + compute_log_pt_w(w, phi, t, beta)
    return np.sum(result)
```

Figure 2: Logarithm joint prob. in factorized Gaussian

Then, we need to find the argmax of ω. Because it is just a derivative of the logarithm toward ω, I use Scipy optimize minimize library to find the mode.(shown in fig 3)

```
solution = scipy.optimize.minimize(log_posterior, w0_laplace, args=(phi_laplace, tn_laplace, alpha, beta,0),method='BFGS')
w_map = solution.x
print("w_map")
print(w_map)
```

Figure 3: Find Gaussian logarithm joint mode

In Dirichlet distribution, I changed the prior from Gaussian distribution to Dirichlet distribution. The main goal is to see the performance difference from Gaussian.
Prior: $Dir(\omega, \alpha) = \frac{\Gamma(\alpha_0)}{\Gamma(\alpha_1)\Gamma(\alpha_2)...\Gamma(\alpha_k)}\Pi_{k=1}^{K}\omega_k^{\alpha_k - 1}$
Likelihood: $p(t|\omega) = \Pi_{n=1}^{N}N(t_n|\omega\Phi_n, \beta^{-1})$.
The logarithm joint: $log(p(t|\omega)) + log(Dir(\omega, \alpha))$. I divided it into a couple of items:
$log(p(t|\omega)) = \frac{-1}{2}(t_n - \Phi_n\omega^T)\beta(t_n - \Phi_n\omega^T)$ and
$log(Dir(\omega, \alpha)) = log(\Gamma(\alpha_0)) - log(\Gamma(\alpha_1)) - log(\Gamma(\alpha_2))... - log(\Gamma(\alpha_k)) + (\alpha_0 - 1)log(\omega_0) + (\alpha_1 - 1)log(\omega_1) + ...(\alpha_k - 1)log(\omega_k)$
The codes are shown in fig 4 (prior) and fig 5 (likelihood). Afterward, I use Scipy library to get the mode of the logarithm.(shown in fig 6)

3. Experimental evaluation
To evaluate the model performance, I use mean square error on the test dataset to confirm how far the model is from the ground truth. The result is shown in fig 7. The result, linear model(9.44), Gaussian model(9.07), and Dirchlet model(2.74), shows how the different models influence performance. For example, the linear and Gaussian models are similar, except the prior is limited by another distribution or not. With the restriction, the performance starts improving (from 9.44 to 9.07). On the other side, we can tell how important a model is in regression. By change the prior from Gaussian to Dirichlet, the performance improves enormously (from 9.07 to 2.74). Though there still is some improvement space (2.74 is too large), I believe trying more distributions can improve the performance to an acceptable score.

4. Future plan
We already saw how much important a model influences performance. The next step would be to find the most proper model(try other models, e.g., Poisson or Wishart). Then, using more accurate computing approximation ways to get the result (e.g., Variational inference).
The source code is put on the GitHub: link

```
def compute_log_pw2(alpha, w): #dirichlet prior
    #left side (not log yet): gamma(a0)/(gmma(a1)*(gmma(a2)...(gmma(ak))
    #right side (not log yet): w0^(a0-1) * w1^(a1-1) * w1^(a2-1) ... * wk^(ak-1)

    #left side log:
    log_gamma_alpha = np.log(gamma(alpha))
    log_gamma_alpha[0] = -1 * log_gamma_alpha[0]
    log_gamma_alpha = -1 * log_gamma_alpha
    leftside = log_gamma_alpha

    #right side:
    alpha_minus_1 = (alpha -1).reshape(1,8)
    #print("alpha_minus_1 shape: ",alpha_minus_1.shape)
    w_normalize = w/np.sum(w)
    #print("w: ",w)
    #print("w_normalize: ",w_normalize)
    log_w = np.log(w_normalize)
    #print("log_w shape: ",log_w.shape)

    rightside = alpha_minus_1 @ log_w.T
    result = leftside + rightside
    return np.sum(result)
```

Figure 4: Dirichlet prior

```
def compute_log_pt_w2(w, phi, t, beta):
    return np.sum(0.5 * np.diagonal((t- phi @ w.T) * beta * ((t- phi @ w.T)).T))

def log_posterior2(w, phi, t, alpha, beta):
    #print("compute_log_pw2: ",compute_log_pw2(alpha, w))
    #print("compute_log_pt_w2: ",compute_log_pt_w2(w, phi, t, beta))
    result = compute_log_pw2(alpha, w) + compute_log_pt_w2(w, phi, t, beta)
    return result
```

Figure 5: Dirichlet likelihood and posterior

```
solution = scipy.optimize.minimize(log_posterior2, w0_laplace, args=(phi_laplace, tn_laplace, alpha, beta),method='BFGS')
w_map = solution.x
print("w_map")
print(w_map)
```

Figure 6: Find logarithm Dirichlet distribution mode

```
#validation
def Get_predic(coef,input):
        input_numpy=input.to_numpy()
        output=np.arange(len(input_numpy), dtype=float)
        for idx in range(len(input_numpy)):
            output[idx] = f(input_numpy[idx], coef)
        return output
test_prediction = testY
test_prediction=Get_predic(mN,testX)
print("Baysian linear Mean Square Error:")
print(np.square(np.subtract(testY,test_prediction)).mean())
#validation end
```

Figure 7: Evaluation

5. Reference to literature

   - Laplace approximation: https://james-brennan.github.io/posts/laplace_approximation/
   - Kaggle data set: https://www.kaggle.com/datasets/grigol1/earthquakes-2000-2023
   - Columans meaning: https://earthquake.usgs.gov/data/comcat/data-eventterms.php
   - Baysien linear learning: https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf
   - Regular linear learning: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html