yunlinz assignment 2 dee924a 6 minutes ago

1 contributor

187 lines (161 sloc) 7.85 KB

Raw Blame History

```python
import tensorflow as tf
import numpy as np
import numpy.random as rn

# Define some constants
POS_1 = tf.constant(1.0)
NEG_1 = tf.constant(-1.0)
DEBUG = False


class TfBayes(object):
    def __init__(self, p0, n, tau=0.005, n_chains=1, t=np.arange(0, 3, 0.003, dtype=float), size=1, data_gen=None):
        """
        Initializes the class for doing Bayesian inference
        :param p0: the initial points in the form of [[A] (n_chains x size), [lambda] (n_chains x size), [sigma] (n_chains x 1)]
        :param n: number of samples to take per chain
        :param tau: step size in Metropolized Langevin process
        :param n_chains: number of parallel MCMC chains to run
        :param t: times where the observations are made
        :param size: number of components in the sum
        :param data_gen: a function that takes in a vector of t and returns the fake data
        """
        self.session = tf.Session()
        self.mcmc_chain = []
        self.prob_chain = []
        self.prop_chain = []
        self.t = t
        self.t_tf = tf.constant(t, shape=[1, len(t)], dtype=tf.float32)
        if data_gen is not None:
            self.observations = data_gen(t)
            self.observations_tf = tf.constant(self.observations, dtype=tf.float32)
        else:
            self.observations = np.zeros((1, len(t)))
            self.gen_data()
        self.n_chains = n_chains
        self.accepted = []
        self.size = size
        self.tau = tau
        self.n = n
        a = p0[0]
        l = p0[1]
        s = p0[2]

        self.sigma = tf.Variable(s, dtype=tf.float32, expected_shape=[self.n_chains, 1])
        self.coeff = tf.Variable(a, dtype=tf.float32, expected_shape=[self.n_chains, self.size])
        self.decay = tf.Variable(l, dtype=tf.float32, expected_shape=[self.n_chains, self.size])

        coeff = tf.reshape(self.coeff, [self.n_chains, 1, self.size])
        decay = tf.reshape(self.decay, [self.n_chains, self.size, 1])

```

```
        ideal = tf.matmul(coeff, tf.exp(NEG_1 * decay * self.t_tf))

        diff_sq = tf.reshape(tf.reduce_sum((ideal - self.observations_tf) ** 2, [1, 2]), [self.n_chains, 1])

        sigma_recip = tf.reshape(tf.div(POS_1, self.sigma), [self.n_chains, 1])
        likelihood = tf.reshape(tf.exp(NEG_1 * tf.reduce_sum(self.sigma ** 2, 1)), [self.n_chains, 1]) * \
                     tf.reshape(
                         tf.exp(NEG_1 * tf.reduce_sum(self.coeff ** 2, 1)), [self.n_chains, 1]) * \
                     tf.reshape(
                         tf.exp(NEG_1 * tf.reduce_sum(self.decay ** 2, 1)), [self.n_chains, 1]) * \
                     tf.reshape(
                         tf.exp(NEG_1 / 2.0 * diff_sq * sigma_recip), [self.n_chains, 1])

        self.log_l = tf.log(likelihood)
        self.grad_log_l = tf.gradients(self.log_l, [self.coeff, self.decay, self.sigma])
        self.mcmc_chain.append([
            np.zeros((self.n_chains, self.size, n)),
            np.zeros((self.n_chains, self.size, n)),
            np.zeros((self.n_chains, 1, n)),
        ])
        self.mcmc_chain[-1][0][:, :, 0] = p0[0]
        self.mcmc_chain[-1][1][:, :, 0] = p0[1]
        self.mcmc_chain[-1][2][:, :, 0] = p0[2]
        self.accepted.append(np.zeros(n))
        self.prob_chain.append(self.loglikelihood(p0))

    def gen_data(self, coeff=None, decay=None, s=0.1):
        """
        Simple data generator for fake data
        :param coeff: list of A
        :param decay: list of lambda
        :param s: noise in the fake data
        :return:
        """
        if decay is None:
            decay = [1.0]
        if coeff is None:
            coeff = [1.0]
        for a, l in zip(coeff, decay):
            self.observations += a * np.exp(-1.0 * l * self.t)
        self.observations += s * rn.randn(len(self.t))
        self.observations_tf = tf.constant(self.observations, dtype=tf.float32)

    def loglikelihood(self, state):
        """
        Calculate likelihood and the gradient
        :param state: State [[A], [lambda], [sigma]]
        :return: [likelihood (double), gradient ([vector, vector, vector])]
        """
        a = state[0]
        l = state[1]
        s = state[2]

        return self.session.run([self.log_l, self.grad_log_l], feed_dict={self.sigma: s, self.coeff: a, self.decay: l})

    def mcmc_sample(self):
        """
        Take MCMC samples based on n defined in initialization of the class
        :return: Nothing
        """
        for i in range(1, self.n):
            if DEBUG:
                print('Running: {}'.format(i))
            last_state = [self.mcmc_chain[-1][component][:, :, i - 1] for component in range(3)]
            last_prob = self.prob_chain[-1][0]
            last_deriv = self.prob_chain[-1][1]
            new_state = [value + self.tau * deriv + np.sqrt(2 * self.tau) *
```

```python
                         (rn.randn(value.shape[0], value.shape[1]))
                         for value, deriv in zip(last_state, last_deriv)]

            result = self.loglikelihood(new_state)

            new_prob = result[0]
            new_deriv = result[1]

            last_to_new, new_to_last = self.transition_prob(last_state, last_deriv, new_state, new_deriv)

            u = rn.rand(self.n_chains)
            prob = (np.exp(new_prob) * new_to_last) / (np.exp(last_prob) * last_to_new)
            accepted = np.less(u, prob.flatten())
            accepted &= (new_state[-1].flatten() >= 0)

            if self.size > 1:
                # we want to enforce lambda to be in increasing order
                for k in range(self.size - 1):
                    accepted &= new_state[1][:, k] <= new_state[1][:, k+1]

            self.prob_chain[-1][0][accepted.flatten()] = new_prob[accepted.flatten()]
            for component in range(3):
                self.mcmc_chain[-1][component][accepted.flatten(), :, i] = new_state[component][accepted.flatten(), :]
                self.mcmc_chain[-1][component][~accepted.flatten(), :, i] = self.mcmc_chain[-1][component][
                                                                            ~accepted.flatten(), :, i - 1]
                self.prob_chain[-1][1][component][accepted.flatten(), :] = new_deriv[component][accepted.flatten(), :]

            self.accepted[-1][i] = sum(accepted)

    def transition_prob(self, last_state, last_deriv, new_state, new_deriv):
        """
        Calculate the transition probabilities between new state and last, and vice versa
        :param last_state: previous MCMC state
        :param last_deriv: previous list of gradients
        :param new_state: new MCMC state
        :param new_deriv: new list of gradients
        :return: (last to new, new to last)
        """

        def trans_prob_helper(state1, deriv1, state2):
            return np.reshape(np.exp(- 1.0 / (4 * self.tau) *
                                     np.sum([np.sum(np.square(item2 - item1 - self.tau * d1), 1)
                                             for item1, d1, item2 in zip(state1, deriv1, state2)], 0)
                                     ),
                              [self.n_chains, 1])

        return (trans_prob_helper(last_state, last_deriv, new_state),
                trans_prob_helper(new_state, new_deriv, last_state))

    def get_state(self):
        import pandas as pd
        result = np.zeros((self.n_chains, 2 * self.size + 1))
        result[:, :self.size] = self.mcmc_chain[-1][0][:, :, -1]
        result[:, self.size:(2 * self.size)] = self.mcmc_chain[-1][1][:, :, -1]
        result[:, -1] = self.mcmc_chain[-1][-1][:, :, -1].flatten()
        headers = ['A_{}'.format(i) for i in range(self.size)] + \
                  ['Lambda_{}'.format(i) for i in range(self.size)] + \
                  ['Sigma']
        return pd.DataFrame(data=result, columns=headers)

    def reset(self):
        """
        Reset all the states in the MCMC chain
        :return: Nothing
        """
        self.mcmc_chain = []
        self.prob_chain = []
```