Tutorial: Testing MCMC code

Roger Grosse and David Duvenaud

December 13, 2014
Motivation

- Why is it hard to test machine learning algorithms?
  - algorithms are stochastic—no single correct output
Motivation

- Why is it hard to test machine learning algorithms?
  - algorithms are stochastic—no single correct output
  - have to distinguish bugs from, e.g., local optima
Motivation

- Why is it hard to test machine learning algorithms?
  - algorithms are stochastic—no single correct output
  - have to distinguish bugs from, e.g., local optima
  - good performance is a matter of degree (if it gets 85.3% on the test set, is it “working?”)
Why is it hard to test machine learning algorithms?
- algorithms are stochastic—no single correct output
- have to distinguish bugs from, e.g., local optima
- good performance is a matter of degree (if it gets 85.3% on the test set, is it “working?”)
- algorithms are designed to handle noisy inputs
  - math mistakes are just another source of noise!
  - buggy algorithms can perform well and make sensible predictions
Motivation

- Why do we care about correctness?
  - Bugs may prevent you from getting state-of-the-art performance
Motivation

- Why do we care about correctness?
  - Bugs may prevent you from getting state-of-the-art performance
  - Experimental manipulations turn out the wrong way
    - maybe some parameter settings better compensate for the bugs
    - results not reproducible
    - screws with your intuition
Why do we care about correctness?

- Bugs may prevent you from getting state-of-the-art performance
- Experimental manipulations turn out the wrong way
  - maybe some parameter settings better compensate for the bugs
  - results not reproducible
  - screws with your intuition
- Partition function estimates are very sensitive to bugs
Overview

- **Unit testing**
  - test a single function at a time
  - fast, deterministic
Overview

- **Unit testing**
  - test a single function at a time
  - fast, deterministic

- **Integration testing**
  - test high-level behaviors
  - often slower
Overview

- **Unit testing**
  - test a single function at a time
  - fast, deterministic

- **Integration testing**
  - test high-level behaviors
  - often slower

- **Common pipeline:**
  1. write integration tests based on a high-level specification
  2. write unit tests
  3. write code to make the unit tests pass
  4. make sure the integration tests pass
Running example: Gibbs sampler for mixture of Gaussians with spherical covariance

$$\pi \sim \text{Dirichlet}(\alpha)$$
$$\sigma^2_\mu \sim \text{InverseGamma}(a_\mu, b_\mu)$$
$$\sigma^2_n \sim \text{InverseGamma}(a_n, b_n)$$
$$z_i | \pi \sim \text{Multinomial}(\pi)$$
$$\mu_{kj} | \sigma^2_\mu \sim \text{Normal}(0, \sigma^2_\mu)$$
$$x_{ij} | z_i, \mu_{z_i,j}, \sigma^2_n \sim \text{Normal}(\mu_{z_i,j}, \sigma^2_n)$$

Essentially, a Bayesian analogue of K-means
Overview

- Straightforward implementation

```python
def sample_from_posterior(X, alpha, K, a_mu, b_mu, a_n, b_n, num_iter):
    # ===== (random initialization) ======

    for it in range(num_iter):
        # update assignments
        prior = np.log(pi)
        evidence = -np.sum((mu[nax, :, :] - X[:, nax, :]) ** 2, 2) * 0.5 / sigma_sq_n
        odds = prior[nax, :] + evidence
        p = np.exp(odds - np.logaddexp.reduce(odds, 1)[nax])
        z = sample_multinomial(p)

        # update centers
        for k in range(K):
            idxs = np.where(z == k)[0]
            h = X[idxs, :].sum(0) / sigma_sq_n
            lam = idxs.size / sigma_sq_n + 1. / sigma_sq_mu
            mu[:, k] = np.random.normal(h / lam, np.sqrt(1. / lam))
```

Roger Grosse and David Duvenaud
Tutorial: Testing MCMC code
December 13, 2014 6 / 29
Overview

- Straightforward implementation

```python
def sample_from_posterior(X, alpha, K, a_mu, b_mu, a_n, b_n, num_iter):
    # ===== (random initialization) ======

    for it in range(num_iter):
        ...

    # update mixture probabilities
    counts = np.bincount(z)
    counts.resize(K)
    pi = sample_dirichlet(alpha + counts)

    # update variance parameters
    a = a_mu + 0.5 * K * ndim
    b = b_mu + 0.5 * np.sum(mu ** 2)
    sigma_sq_mu = 1. / np.random.gamma(a, 1. / b)

    a = a_n + 0.5 * ndata * ndim
    b = b_n + 0.5 * np.sum((X - mu[z, :]) ** 2)
    sigma_sq_n = 1. / np.random.gamma(a, 1. / b)

    return mu, z, pi, sigma_sq_n, sigma_sq_mu
```

Roger Grosse and David Duvenaud
Tutorial: Testing MCMC code
December 13, 2014 7 / 29
Modularity

- Hard to retrofit existing code to be unit testable
- Needs to be modular
  - the requirement of testability can force you to write cleaner and more maintainable code
Modularity

- Hard to retrofit existing code to be unit testable
- Needs to be modular
  - the requirement of testability can force you to write cleaner and more maintainable code
- One possible decomposition for MCMC:
  - probability distributions (know how to sample, evaluate log probabilities)
  - functions to compute gradients, conditional probabilities, etc.
  - the MCMC routines themselves
Classes representing the model definition and state:

class Model:
    def __init__(self, alpha, K, sigma_sq_mu_prior, sigma_sq_n_prior):
        self.alpha = alpha  # Parameter for Dirichlet prior
        self.K = K  # Number of components
        self.sigma_sq_mu_prior = sigma_sq_mu_prior
        self.sigma_sq_n_prior = sigma_sq_n_prior

class State:
    def __init__(self, z, mu, sigma_sq_mu, sigma_sq_n, pi):
        self.z = z  # Assignments
        self.mu = mu  # Cluster centers
        self.sigma_sq_mu = sigma_sq_mu  # Between-cluster variance
        self.sigma_sq_n = sigma_sq_n  # Within-cluster variance
        self.pi = pi  # Mixture probabilities
Classes representing probability distributions:

class GaussianDistribution:
    def __init__(self, mu, sigma_sq):
        self.mu = mu
        self.sigma_sq = sigma_sq

    def log_p(self, x):
        return -0.5 * np.log(2*np.pi) + \
        -0.5 * np.log(self.sigma_sq) + \
        -0.5 * (x - self.mu)**2 / self.sigma_sq

    def sample(self):
        return np.random.normal(self.mu, np.sqrt(self.sigma_sq))
Modularity

Computing conditional distributions

```python
class Model:
    ...

    def cond_sigma_sq_mu(self, state):
        ndim = state.mu.shape[1]
        a = self.sigma_sq_mu_prior.a + \
            0.5 * self.K * ndim
        b = self.sigma_sq_mu_prior.b + \
            0.5 * np.sum(state.mu ** 2)
        return InverseGammaDistribution(a, b)

    def cond_sigma_sq_n(self, state, X):
        ndata, ndim = X.shape
        a = self.sigma_sq_n_prior.a + \
            0.5 * ndata * ndim
        b = self.sigma_sq_n_prior.b + \
            0.5 * np.sum((X - state.mu[state.z, :]) ** 2)
        return InverseGammaDistribution(a, b)
```
Modularity

### Computing conditional distributions

class Model:

...  

    def cond_pi(self, state):
        counts = np.bincount(state.z)
        counts.resize(self.K)
        return DirichletDistribution(self.alpha + counts)

    def cond_z(self, state, X):
        prior = np.log(state.pi)
        evidence = GaussianDistribution(state.mu[nax, :, :],
                                         state.sigma_sq_n).log_p(X[:, nax, :]).sum(2)
        return MultinomialDistribution.from_log_odds(prior[nax, :] + evidence)

    def cond_mu(self, state, X):
        ndata, ndim = X.shape
        h = np.zeros((self.K, ndim))
        lam = np.zeros((self.K, ndim))
        for k in range(self.K):
            h[k, :] = X[idxs, :].sum(0) / state.sigma_sq_n
            lam[k, :] = (state.assignments==k).sum() / state.sigma_sq_n + 1. / state.sigma_sq_mu
        return GaussianDistribution(h / lam, 1. / lam)
**Modularity**

- Gibbs sampling

```python
class Model:
    ...

def gibbs_step(self, state, X):
    state.pi = self.cond_pi(state).sample()
    state.z = self.cond_z(state, X).sample()
    state.mu = self.cond_mu(state, X).sample()
    state.sigma_sq_mu = self.cond_sigma_sq_mu(state).sample()
    state.sigma_sq_n = self.cond_sigma_sq_n(state, X).sample()
```

Roger Grosse and David Duvenaud
Unit tests

- Testing conditional probabilities

\[
\frac{p(X'|Z)}{p(X|Z)} = \frac{p(X', Z)}{p(X, Z)}
\]  \hspace{1cm} (1)

- Check that this holds to high precision for random \(X, X', Z\)
- Fails with high probability if your math is wrong
Unit tests

Computing joint probability (needed for unit tests):

class Model:
    ...

def joint_log_p(self, state, X):
    return DirichletDistribution(self.alpha * np.ones(self.K)).log_p(
        state.pi) + \\n    MultinomialDistribution.from_probabilities(state.pi).log_p(
        state.z).sum() + \\n    self.sigma_sq_mu_prior.log_p(state.sigma_sq_mu) + \\n    self.sigma_sq_n_prior.log_p(state.sigma_sq_n) + \\n    GaussianDistribution(0., state.sigma_sq_mu).log_p(
        state.mu).sum() + \\n    GaussianDistribution(state.mu[state.z, :],
        state.sigma_sq_n).log_p(X).sum()
Testing the conditional distributions:

```python
def test_cond_mu():
    np.random.seed(0)
    model = random_model()
    state, X = model.forward_sample(N, D)
    new_state = copy.deepcopy(state)
    new_state.mu = np.random.normal(size=(K, D))
    cond = model.cond_mu(state, X)
    assert np.allclose(cond.log_p(new_state.mu).sum() -
                        cond.log_p(state.mu).sum(),
                        model.joint_log_p(new_state, X) -
                        model.joint_log_p(state, X))
```
Most languages provide convenient unit testing frameworks (e.g. `nose` in Python)
  - easily and quickly run your whole suite of tests
  - keep your testing code in a separate directory to avoid clutter
Let’s sanity check our tests by adding a small typo:

```python
class Model:
    ...

    def cond_sigma_sq_n(self, state, X):
        ndata, ndim = X.shape
        a = self.sigma_sq_n_prior.a + \n            0.5 * ndata * ndim
        b = self.sigma_sq_n_prior.b + \n            0.51 * np.sum((X - state.mu[state.z, :]) ** 2)  # oops!
        return InverseGammaDistribution(a, b)
```

Unit testing

- Only the test corresponding to this function fails:

  rgrosse:~/code/testing$ nosetests
  ....F
  ==============================================================
  FAIL: test_mog.test_cond_sigma_sq_n
  ==============================================================
  Traceback (most recent call last):
    File "/Users/rgrosse/anaconda/lib/python2.7/site-packages/nose/case.py", line 197, in runTest
      self.test(*self.arg)
    File "/Users/rgrosse/code/testing/test_mog.py", line 74, in test_cond_sigma_sq_n
      model.joint_log_p(new_state, X) - model.joint_log_p(state, X))
  AssertionError
  
  ==============================================================
  Ran 5 tests in 0.217s

  FAILED (failures=1)
Unit testing

We fix the typo, and everything passes:

```
rgrosse:~/code/testing$ nosetests
.....
Ran 5 tests in 0.231s

OK
```
Unit testing

- Some other unit tests:
  - Gradient checking (e.g. `scipy.optimize.check_grad`)
  - Check optimality conditions in coordinate descent (e.g. mean field)
  - Check a fast, optimized function against a slow, simple one
  - Maintain invariants for sufficient statistics, etc.
  - List various mathematical properties your algorithm should satisfy
Integration testing: the Geweke test

- Most of the bugs I’ve caught, I’ve caught with unit testing.
- But notice we still haven’t tested the Gibbs sampler itself.

Powerful method for testing MCMC code: the Geweke Test


Roger Grosse and David Duvenaud
Tutorial: Testing MCMC code
December 13, 2014
Integration testing: the Geweke test

- Most of the bugs I’ve caught, I’ve caught with unit testing.
  - But notice we still haven’t tested the Gibbs sampler itself.

**Integrating testing:** test how everything fits together to produce the desired behavior

- Powerful method for testing MCMC code: **the Geweke Test**
  - He caught multiple bugs in his own previously published results.
Integration testing: the Geweke test

- Two ways of sampling from $p(\theta, x)$:
  1. Forward sampling: sample from $p(\theta)$, then $p(x | \theta)$
  2. Start with a forward sample, then alternate between:
     1. take an MCMC step which preserves $p(\theta | x)$
     2. resample $x$ from $p(x | \theta)$
Integration testing: the Geweke test

- Two ways of sampling from $p(\theta, x)$:
  1. Forward sampling: sample from $p(\theta)$, then $p(x | \theta)$
  2. Start with a forward sample, then alternate between:
     1. take an MCMC step which preserves $p(\theta | x)$
     2. resample $x$ from $p(x | \theta)$

- Both methods yield a sequence of exact samples from $p(x | \theta)$, so the two distributions should be indistinguishable
  - Compare lots of statistics
  - Geweke does frequentist hypothesis tests, but this is difficult because of correlations in the samples
  - I just look at P-P plots
Integration testing: the Geweke test
Integration testing: the Geweke test

- **Geweke test code:**

  ```python
  def geweke(model, num_samples):
      forward_samples = []
      for i in range(num_samples):
          state, X = model.forward_sample(N, D)
          forward_samples.append(state.sigma_sq_n)

      gibbs_samples = []
      state, X = model.forward_sample(N, D)
      for i in range(num_samples):
          model.gibbs_step(state, X)
          X = model.cond_X(state).sample()
          gibbs_samples.append(state.sigma_sq_n)

      pp_plot(forward_samples, gibbs_samples)
  ```
Integration testing: the Geweke test

Let’s re-introduce our 0.51 bug:

class Model:
    ...

    def cond_sigma_sq_n(self, state, X):
        ndata, ndim = X.shape
        a = self.sigma_sq_n_prior.a + \
            0.5 * ndata * ndim
        b = self.sigma_sq_n_prior.b + \
            0.51 * np.sum((X - state.mu[state.z, :]) ** 2)  # oops!
        return InverseGammaDistribution(a, b)

• estimated variance is about 2% too large
• this is subtle enough that you might never notice it normally
Integration testing: the Geweke test

- What happens in the Geweke test?
  - estimate variance 2% too large
  - re-generate data with 2% too large variance
  - estimate variance 2% too large on top of that (4% total)
  - pretty soon, the variance explodes...
- The Geweke test can amplify some very subtle bugs.
Integration testing: the Geweke test

- One drawback: it gives you no indication what you did wrong
  - No point in running it until all your unit tests pass
Integration testing: the Geweke test

- One drawback: it gives you no indication what you did wrong
  - No point in running it until all your unit tests pass
- Using this test, I caught a subtle bug in the original Indian buffet process sampler
  - must resample each row of binary variables in a *random* order, not the canonical order
I showed a toy model; how might we want to extend it?

- Replace the isotropic covariance with a general covariance matrix (with an inverse Wishart prior)
- Untie the covariance between different clusters
- Collapse out the mixture probabilities and the cluster centers in order to speed up mixing
- Replace the finite Dirichlet mixture with a Chinese restaurant process
- Slice sample the hyperparameters
- Use split-merge proposals to speed up mixing
Discussion

- I showed a toy model; how might we want to extend it?
  - Replace the isotropic covariance with a general covariance matrix (with an inverse Wishart prior)
  - Untie the covariance between different clusters
  - Collapse out the mixture probabilities and the cluster centers in order to speed up mixing
  - Replace the finite Dirichlet mixture with a Chinese restaurant process
  - Slice sample the hyperparameters
  - Use split-merge proposals to speed up mixing

- All of these modifications can be tested in the same way
- Having a modular, testable organization makes it easier to build sophisticated models and samplers incrementally