Off-Policy Policy Evaluation: Algorithms, Theory, and Python Implementations

Key Takeaway: Off-policy evaluation (OPE) enables estimating the value of a target policy πₑ using trajectories generated under a behavior policy π_b. The three principal approaches—**Importance Sampling**, **Direct Method**, and **Doubly Robust**—balance bias and variance differently.

1. Theoretical Background

1.1. Problem Setting

We assume an episodic Markov decision process (MDP) with finite horizon H, discount factor γ, and state–action trajectories

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_H)$$

logged under a **behavior policy** π_b. Our goal is to estimate the **expected discounted return**

$$V^{\pi_e} = \mathbb{E}_{\tau \sim \pi_e}\Big[\sum_{t=0}^{H-1} \gamma^t \, r_t\Big]$$

using only the logged data $\{\tau^{(i)}\}$.

1.2. Importance Sampling (IS)

Importance Sampling re-weights returns from π_b to unbiasedly estimate π_e's value:

$$\hat{V}_{\mathrm{IS}} = \frac{1}{N} \sum_{i=1}^{N} \Big(\prod_{t=0}^{H-1} \omega_t^{(i)}\Big) \sum_{t=0}^{H-1} \gamma^t \, r_t^{(i)}, \quad \omega_t^{(i)} = \frac{\pi_e(a_t^{(i)} \mid s_t^{(i)})}{\pi_b(a_t^{(i)} \mid s_t^{(i)})}.$$

While unbiased, **IS** often suffers high variance when π_e diverges from π_b.

1.3. Direct Method (DM)

The Direct Method fits a model of the **action-value function** $Q(s, a)$ (e.g., via regression) from the logged data, then computes:

$$\hat{V}_{\mathrm{DM}} = \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{H-1} \gamma^t \sum_{a} \pi_e(a \mid s_t^{(i)}) \, \widehat{Q}(s_t^{(i)}, a).$$

DM can yield low variance but incurs **bias** if $\widehat{Q}$ is misspecified.

1.4. Doubly Robust (DR)

Doubly Robust marries IS and DM to achieve unbiasedness and variance reduction. For each trajectory:

$$\hat{V}_{\mathrm{DR}} = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=0}^{H-1}\gamma^t\Big[\omega_{0:t}^{(i)}\big(r_t^{(i)} - \widehat{Q}(s_t^{(i)}, a_t^{(i)})\big) + \sum_a \pi_e(a \mid s_t^{(i)})\,\widehat{Q}(s_t^{(i)}, a)\Big],$$

where $\omega_{0:t} = \prod_{u=0}^{t}\omega_u$. DR is **consistent**, **unbiased**, and often lower-variance than plain IS.

2. Python Implementations

All three modules expect as input a JSON file containing a list of episodes. Each episode is a list of time-step records, each record a dictionary with the fields:

- `"state"`: any serializable representation of the state,
- `"action"`: the taken action (e.g., integer),
- `"reward"`: the scalar reward,
- `"pi_b"`: the behavior policy probability π_b(a|s),
- `"pi_e"`: the evaluation policy probability π_e(a|s).

RLLib's PPO policy can supply `"pi_e"` via `policy.compute_action(state, full_fetch=True)`.

2.1. importance_sampling.py

```
import json
import numpy as np

def load_episodes(path):
    with open(path, 'r') as f:
        return json.load(f)

def importance_sampling(path, gamma=1.0):
    episodes = load_episodes(path)
    returns = []
    for ep in episodes:
        rho = 1.0
        G = 0.0
        for t, step in enumerate(ep):
            rho *= step['pi_e'] / step['pi_b']
            G += (gamma ** t) * step['reward']
        returns.append(rho * G)
    return np.mean(returns), np.std(returns) / np.sqrt(len(returns))

if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("file", help="JSON episodes file")
    parser.add_argument("--gamma", type=float, default=1.0)
    args = parser.parse_args()
    mean, stderr = importance_sampling(args.file, args.gamma)
    print(f"IS estimate: {mean:.4f} ± {1.96*stderr:.4f}")
```

2.2. direct_method.py

```
import json
import numpy as np
from sklearn.ensemble import RandomForestRegressor

def load_episodes(path):
    with open(path, 'r') as f:
        return json.load(f)

def fit_q(episodes, gamma=1.0):
    X, Y = [], []
    for ep in episodes:
        G = 0.0
        for step in reversed(ep):
            G = step['reward'] + gamma * G
            X.append((step['state'], step['action']))
            Y.append(G)
    # Feature engineering: flatten state and action
    Xf = [np.concatenate([np.ravel(s), [a]]) for s,a in X]
    model = RandomForestRegressor()
    model.fit(Xf, Y)
    return model

def direct_method(path, gamma=1.0):
    episodes = load_episodes(path)
    model = fit_q(episodes, gamma)
    values = []
    for ep in episodes:
        V = 0.0
        for t, step in enumerate(ep):
            feats = np.concatenate([np.ravel(step['state']), []])
            # For discrete actions, enumerate possible actions from data
            # Here assume action space {0,...,A-1}
            qs = []
            for a in range(model.n_estimators):  # placeholder for action set
                feat = np.concatenate([np.ravel(step['state']), [a]])
                qs.append(model.predict([feat])[^0] * step.get('pi_e',1.0))
            V += (gamma**t) * sum(qs)
        values.append(V)
    return np.mean(values), np.std(values)/np.sqrt(len(values))

if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("file", help="JSON episodes file")
    parser.add_argument("--gamma", type=float, default=1.0)
    args = parser.parse_args()
    mean, stderr = direct_method(args.file, args.gamma)
    print(f"DM estimate: {mean:.4f} ± {1.96*stderr:.4f}")
```

2.3. doubly_robust.py

```
import json
import numpy as np
from direct_method import fit_q

def load_episodes(path):
    with open(path, 'r') as f:
        return json.load(f)

def doubly_robust(path, gamma=1.0):
    episodes = load_episodes(path)
    model = fit_q(episodes, gamma)
    dr_values = []
    for ep in episodes:
        dr = 0.0
        rho = 1.0
        for t, step in enumerate(ep):
            s, a, r = step['state'], step['action'], step['reward']
            # Q-model prediction
            feat = np.concatenate([np.ravel(s), [a]])
            q_sa = model.predict([feat])[^0]
            # Expected Q under π_e
            # assume discrete actions 0..A-1
            q_exp = 0.0
            for a2 in range(model.n_estimators):  # placeholder
                feat2 = np.concatenate([np.ravel(s), [a2]])
                q_exp += step['pi_e'] * model.predict([feat2])[^0]
            dr += (gamma**t) * (rho * (r - q_sa) + rho * q_exp)
            rho *= step['pi_e'] / step['pi_b']
        dr_values.append(dr)
    return np.mean(dr_values), np.std(dr_values)/np.sqrt(len(dr_values))

if __name__ == "__main__":
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument("file", help="JSON episodes file")
    parser.add_argument("--gamma", type=float, default=1.0)
    args = parser.parse_args()
    mean, stderr = doubly_robust(args.file, args.gamma)
    print(f"DR estimate: {mean:.4f} ± {1.96*stderr:.4f}")
```

3. Explanation of Algorithms

- **Importance Sampling** re-weights each trajectory by the ratio of likelihoods under π_e vs π_b, providing an unbiased but potentially high-variance estimator.
- **Direct Method** fits a regression model for the return (or Q-function) from logged data, then uses it to predict π_e's performance. It has low variance but is biased if the model is misspecified.
- **Doubly Robust** incorporates both IS and DM: it corrects the DM estimate by adding an IS-weighted residual term. DR remains unbiased even if the DM model is incorrect, and often enjoys much lower variance than plain IS.

These implementations assume discrete action spaces and require filling in the action set size. To integrate with RLLib 2.11's PPO policy, record `"pi_e"` by querying the trained policy's action probability for each state prior to logging trajectories.

⁂