Project 2: Reinforcement Learning

Shengtong Zhang stzh1555@stanford.edu
AA228/CS238, Stanford University

1. Algorithm Description

For small.csv, I use maximal likelihood to estimate the transition functions (which is ok I guess since we have 100000 data points for 400 state-action pairs) and the reward function, and run value iteration. After less than 1000 iterations, the algorithm converges.

For medium.csv, I first manually estimate the transition model, which is very annoying since everything is discretized. After a few hours of statistical analysis, my best guess for the transition/reward model is, for state $s = (p, v)$ and action a

$$\begin{aligned} v'(p, v, a) &= v + 0.25 * (a - 4) + 2.0 * \cos(2\pi(p - 50)/590), \\ p'(p, v, a) &= p + 0.353 * (v' - 50), \\ R(p, v, a) &= 100000 * \mathbf{1}_{p \geq 454} - 25(a - 4)^2. \end{aligned}$$

I then use value iteration with approximate value function, which I take to be a degree 5 polynomial in v and p. I normalize v and p to have absolute value at most 1 to resolve scaling issues. After less than 2000 iterations the strategy converges to something ok. An observation is that it is useful to first train a simpler model with less parameters then add more parameters, instead of training the complicated model from scratch.

To fit the model, I used Julia's GLM package.

For large.csv most of my effort is spent figuring out the model. The best model I can find is this.

- The state space is secretly 5 layers of 10×10 gridworlds.
- Action $1, 2, 3, 4$ take you to an adjacent grid on the same layer. Each action has a "principal direction"; you have $(1 - p)$ probability of taking the principal direction and p of taking another random direction. The value of p seems to depend on the layer you are at. I estimate p to be $0.20, 0.30, 0.20, 0.25, 0.10$ on the five layers.
- At the four corners of the gridworlds there are teleporters. Action $5, 6, 7, 8, 9$ controls these teleporters. If you stand at a teleporter on layer ℓ and take action $a \geq 5$ with $a - 4 \neq \ell$, you have roughly $1 - 136/450$ probability of being transported to a uniformly random square on layer $(a - 4)$, and $136/450$ probability of being transported to a uniformly random square on a different layer that is not $(a - 4)$ or ℓ. Otherwise, the action does nothing.
- The reward function is a deterministic function of (s', a) instead of (s, a).

With this model I can simply do value iteration, and the policy converges after less than 2000 iterations.

2. Runtime

I spent most of my time on the project manually interpolating the parameters for medium and large. Here are the runtime for learning.

Small: 3 seconds.

Medium: It takes about 5 minutes to train a degree 3 model, and 15 minutes to refine it into a degree 5 model.

Large: About 40 seconds.

3. Code

My code for parameter interpolation is an utter mess and probably unintelligible for you. So I only included the code for policy learning.

If you want to run the code, you need to replace all "\div" with ÷, as LaTeX hates the latter. Also, the code for large depends on the the struct MDP in small.

3.1 Small

```
using CSV
using DataFrames
using Printf
# basic structure for Markov Decision Problems
mutable struct MDP
    gamma::Float64
    S::UInt64
    A::UInt64
    T
    R # map (state, action) to reward
    samples # map (state, action) to a Dict mapping each state to its count
end

# loading data
# filename:
# S: an integer, representing the size of the state space
# A: an integer, representing the size of the action space
# gamma: the discount trate
# guess: a function that completes the T and R matrix based on samples
function load_from_file(filename, S, A, gamma)
    df = CSV.read(filename, DataFrame)
    return load_from_df(df, S, A, gamma)
end

function load_from_df(df, S, A, gamma)
    # initialize the sample and reward array
    samples = Dict()
    rewards = Dict()
    for s=1:S
        for a=1:A
            samples_s_a = Dict()
            df_s_a = df[(df.s.==s) .& (df.a.==a), :]
            if size(df_s_a, 1) > 0
                rewards[(s, a)] = df_s_a[1, :r]
                df_s_a_count = combine(groupby(df_s_a, :sp), nrow)
                for i=1:size(df_s_a_count,1)
                    samples_s_a[df_s_a_count[i, :sp]] = df_s_a_count[i,:nrow]
                end
                samples[(s, a)] = samples_s_a
            else
```

```
                throw(error("missing samples"*string(s)*","*string(a)))
            end
        end
    end
    return MDP(gamma, S, A, Dict(), rewards, samples)
end

# completing the Transition matrix using max likelihood
function complete_T_MLE(P::MDP)
    T = Dict()
    for s=1:P.S
        for a=1:P.A
            T_s_a = Dict()
            samples = P.samples[s, a]
            ttl = sum([cnt for (sp, cnt) in samples])
            for (sp, cnt) in samples
                T_s_a[sp] = cnt / ttl
            end
            T[(s,a)] = T_s_a
        end
    end
    P.T = T
end

# one step lookahead, based on a certain utility function
# given MDP P and utility U, output best policy and utility from state s
function lookahead(P::MDP, U, s)
    Q_s = Dict()
    for a=1:P.A
        Q_s[a] = P.R[(s,a)] + P.gamma * sum([prob * U[sp] for (sp, prob) in P
    .T[(s,a)]])
    end
    # update
    a_s, U_s = -1, -Inf
    for a=1:P.A
        if Q_s[a] > U_s
            a_s = a
            U_s = Q_s[a]
        end
    end
    return a_s, U_s
end

# solving small via value iteration
function value_iteration(P::MDP, iter::Int64)
    # initialize utility function
    # utility function
    U = Dict()
    for s=1:P.S
        U[s] = 0
```

```
    end
    # bellman update
    # looka

    for it=1:iter
        for s=1:P.S
            # compute action-value funciton
            a_s, U_s = lookahead(P,U,s)
            U[s] = U_s
        end
    end

    # output policy based on utility
    policy = Dict()
    for s=1:P.S
        a_s, U_s = lookahead(P,U,s)
        policy[s] = a_s
    end

    return U, policy
end


P_small = load_from_file("data/small.csv", 100, 4, 0.95)
complete_T_MLE(P_small)
U, policy = value_iteration(P_small, 1000)
# output policy
outfile = "policy/small.policy"
open(outfile, "w") do io
    for s=1:P_small.S
        a_s, U_s = lookahead(P_small, U, s)
        @printf(io, "%d\n", a_s)
    end
end
```

3.2 Medium

```
using CSV
using DataFrames
using Printf
using DataFrames, GLM, StatsBase, LinearAlgebra

Pi = 3.1415926

function dv_emp(p, F)
    return 0.25 * F + 2.0 * cos(2 * Pi * (p - 50) / 590)
end


# reward function
function reward(p, a)
    flag = 0
    if p >= 454
        flag = 100000
    end
    return flag - 25 * (a - 4)^2
end

# transition function interpolated from data
# the transition is deterministic
function transition(p, v, a)
    F = a - 4
    vp = v + dv_emp(p, F)
    pp = p + 0.353 * vp
    # wall collision
    if pp < 0
        pp, vp = 1, 0
    end
    return (pp, vp)
end

# generate feature for a given df
# assume df must have columns p, v
# we also normalize these p and v a bit
F_LEN = 21 # parameter, number of features
function make_feature(p, v)
    p_n = p / 500
    v_n = v / 50
    feature =  (1,
                p_n, v_n,
                p_n^2, p_n*v_n, v_n^2,
                p_n^3, p_n^2*v_n, p_n*v_n^2, v_n^3,
                v_n^4, p_n^4, p_n^3*v_n, p_n*v_n^3, p_n^2 * v_n^2,
                v_n^5, p_n^5, p_n^4*v_n, p_n^3*v_n^2, p_n^2 * v_n^3, p_n *
    v_n^4)
```

```
    @assert size(feature, 1) == F_LEN
    return feature
end

# add the features to the dataframe
function add_feature_df(df)
    for i=1:F_LEN
        col_name = 'f'*string(i)
        df[!, col_name] = [make_feature(r.p, r.v)[i] for r in eachrow(df)]
    end
    return df
end

# generate the greedy action w.r.t the reward function defined by theta
# row is a row vector, where (row.p, row.v) is the current states
# theta is the parameter
function greedy(row, theta)
    @assert size(theta, 1) == F_LEN
    best_a, best_r = 0, -Inf
    p, v = row.p, row.v
    for a=1:7
        pp, vp = transition(p, v, a)
        if reward(p, a) < 0
            r = reward(p, a) + dot(theta, make_feature(pp, vp))
        else
            # if we don't hit flag, we stop earning reward
            r = reward(p, a)
        end
        if r > best_r
            best_a, best_r = a, r
        end
    end
    return best_a, best_r
end

# update the parameters to reflect
# theta denotes the old set of parameters. The utility function is
    approximated by theta dot
# returns the new set of parameters
# df should contain the feature columns, f1 ... fn
function update(theta, df)
    # predict the reward using the
    df.U_p = [dot(theta, make_feature(r.p, r.v)) for r in eachrow(df)]
    # find the greedy utility
    df.U = [greedy(r, theta)[2] for r in eachrow(df)]
    # print how far the reward is from the actual
    loss = mean((df.U_p .- df.U) .^ 2)
    @printf("Average Square Loss: %f\n", loss)
    # Fit the new parameters, using simple linear regression
```

```
    ols = lm(@formula(U ~ f2 + f3 + f4 + f5 + f6 + f7 + f8 + f9 + f10 + f11 +
     f12 + f13 + f14 + f15 + f16 + f17 + f18 + f19 + f20 + f21), df) # f1 is
    not here since life
    return GLM.coef(ols)
end

# initialize a dataframe consisting of p, v and the feature vectors
df = DataFrame(s=1:50000)
df.p = (df.s .- 1) .% 500
df.v = (df.s .- 1) .\div 500 .- 50
add_feature_df(df)
# we start with a degree 3 model
theta = [95210.32621926059, 3167.834252108287, 681.3966194715723,
    -33975.77329677989, 3823.0941527381738, 2157.0855349683798,
    37281.21716333439, -3298.468309221864, -1936.1267282317328,
    -450.0528392257383,
    0,0,0,0,0,0,0,0,0,0]
for iter=1:1000
    theta = update(theta, df)
    if iter % 50 == 0
        @printf("Iteration: %d\n", iter)
        @printf("%s\n", theta)
        @printf("==============================")

    end
end
@printf("%s\n", theta)
# output policy
outfile = "policy/medium.policy"
open(outfile, "w") do io
    for s=1:50000
        p = (s - 1) % 500
        v = (s - 1) \div 500 - 50
        row = DataFrame(p=p, v=v)
        a_s, U_s = greedy(row[1,:], theta)
        @printf(io, "%d\n", a_s)
    end
end
```

3.3 Large

```
# let's large.csv

df_large = CSV.read("data/large.csv", DataFrame)
@printf("state space: %d\n", maximum(df_large.s))
@printf("action space: %d\n", maximum(df_large.a))
# It looks like the state space can be divided into three parts
s = df_large.s
df_large.s1 = s.\div 10000
```

```
df_large.s2 = (s.%10000).\div 100
df_large.s3 = s.%100
sp = df_large.sp
df_large.sp1 = sp.\div 10000
df_large.sp2 = (sp.%10000).\div 100
df_large.sp3 = sp.%100
# it looks like the first coord only takes 5 values
# and the second, third coord only takes 10 values
# so the state space has size 500???
S1Range = [15,23,27,29,30]
S2Range = [1,2,3,4,10,11,12,13,14,20]
S3Range = [1,2,3,4,10,11,12,13,14,20]

S1Code = Dict()
for i=1:size(S1Range,1)
    S1Code[S1Range[i]] = i - 1
end
S2Code = Dict()
for i=1:size(S2Range,1)
    S2Code[S2Range[i]] = i - 1
end
S3Code = Dict()
for i=1:size(S3Range,1)
    S3Code[S3Range[i]] = i - 1
end

function encode(s)
    s1 = s \div 10000
    s2 = (s%10000)\div 100
    s3 = s%100
    return 100 * S1Code[s1] + 10 * S2Code[s2] + S3Code[s3] + 1
end
df_large_encoded = DataFrame("s" => [encode(s) for s in df_large.s], "a" =>
    df_large.a,
                             "r" => df_large.r, "sp" => [encode(sp) for sp in
     df_large.sp])

# actions?
# alright it seems 1,2,3,4 are standard grid world actions
# where as 5,6,7,8,9 "jumps between layers" seemingly randomly with very low
    probability of succeeding
# (See below for the description of the teleporters)
# ok I guess I can just use value iteration
# this is a dedicated version that is hand-tuned to this particular map
p_dir = [(1, 0), (-1, 0), (0, -1), (0, 1)]
function load_from_df_large(df, S, A, gamma)
    # initialize the sample and reward array
    samples = Dict()
    rewards = Dict()
    for s=1:S
```

```
        for a=1:A
            samples_s_a = Dict()
            df_s_a = df[(df.s.==s) .& (df.a.==a), :]
            if size(df_s_a, 1) > 0
                df_s_a_count = combine(groupby(df_s_a, :sp), nrow)
                for i=1:size(df_s_a_count,1)
                    samples_s_a[df_s_a_count[i, :sp]] = df_s_a_count[i,:nrow]
                end
                samples[(s, a)] = samples_s_a
            end
        end
    end
    # the reward function here is a function of (sp, a) instead of (s, a)
    for sp=1:S
        for a=1:A
            samples_sp_a = Dict()
            df_sp_a = df[(df.sp.==sp) .& (df.a.==a), :]
            if size(df_sp_a, 1) > 0
                rewards[(sp, a)] = df_sp_a[1, :r]
                # check that this is indeed the correct assumption
                for row in eachrow(df_sp_a)
                    @assert row.r == df_sp_a[1, :r]
                end
            end
        end
    end
    # there might be a few missing entries
    for sp=1:S
        for a=1:A
            if !((sp, a) in keys(rewards))
                # I know that a >= 5 always give reward 0
                if a >= 5
                    rewards[(sp, a)] = 0
                else
                    # otherwise, hope that there is data for some other a
                    ap = 5 - a
                    rewards[(sp, a)] = rewards[(sp, ap)]
                end
            end
        end
    end
    return MDP(gamma, S, A, Dict(), rewards, samples)
end

# completing the Transition matrix
# fine_tuned to particular behavior of this instance
function complete_T_large(P::MDP)
    T = Dict()
    for s=1:P.S
        for a=1:P.A
```

```
        # extract coordinates
        l = (s - 1)\div100 + 1 # l ranges from 1-5
        x = (s - 1)%10 + 1 # x ranges from 1-10
        y = ((s - 1)\div 10)%10 + 1 # y also ranges from 1-10
        # non-teleporter actions
        # make the same assumption that they behave identically at each
square
        if a <= 4
            T_s_a = Dict()
            p_dir = [(1, 0), (-1, 0), (0, -1), (0, 1)] # the "principal
direction" of each action
            p_err = [0.20, 0.30, 0.20, 0.25, 0.10] # the probability that
 an action makes an error
            # this looks like a good heuristic, but not optimal still
            for dir in p_dir
                # calculate the destination if we step in dir
                xp, yp = x + dir[1], y + dir[2]
                # boundary collision
                if (xp < 1) | (xp > 10) | (yp < 1) | (yp > 10)
                    xp, yp = x, y
                end
                sp = (l - 1) * 100 + (yp - 1) * 10 + (xp - 1) + 1
                if dir == p_dir[a]
                    prob = 1 - p_err[l]
                else
                    prob = p_err[l] / 3
                end
                if !(sp in keys(T_s_a))
                    T_s_a[sp] = 0
                end
                T_s_a[sp] += prob
            end
            T[(s,a)] = T_s_a
        else
            # not a teleporter spot
            if !((s % 100) in [0,1,10,91])
                T[(s,a)] = Dict(s => 1.0)
            else
                # I dont really know what to do in this case
                # I'll go with simple empirical evidence for now
                # ok, based on further study
                # it seems that there is about 20% probability that one
is teleported to some
                # random spot not in action_layer
                # and 80% probability that one is teleported to some
random spot in action_layer
                action_layer = a - 4
                if (action_layer == l)
                    T[(s,a)] = Dict(s => 1.0)
                else
```

```
                        T[(s,a)] = Dict()
                        for lp=1:5
                            for sp=(100*lp-99):(100*lp)
                                # empirical probability of failed teleport
                                p_fail = 136 / 450
                                # there seem to be a broken teleporter at s =
    401
                                # but including this seems to be a bad idea
                                #if (s == 401)
                                #    p_fail = 0.75
                                #end
                                if (lp == action_layer)
                                    T[(s, a)][sp] = (1 - p_fail) / 100
                                # it looks like wrong teleportation
                                # always sends us to a different floor
                                elseif (lp != l)
                                    T[(s, a)][sp] = p_fail / 300
                                end
                            end
                        end
                    end
                end
            end
        end
    end
    P.T = T
end

# one step lookahead, based on a certain utility function
# given MDP P and utility U, output best policy and utility from state s
# adapted for utility function depending on (sp, a) instead of (s, a)
function lookahead_large(P::MDP, U, s)
    Q_s = Dict()
    for a=1:P.A
        Q_s[a] = sum([prob*(P.R[(sp,a)] + P.gamma * U[sp]) for (sp, prob) in
    P.T[(s,a)]])
    end
    # update
    a_s, U_s = -1, -Inf
    for a=1:P.A
        if Q_s[a] > U_s
            a_s = a
            U_s = Q_s[a]
        end
    end
    return a_s, U_s
end

# solving large via value iteration
function value_iteration_large(P::MDP, iter::Int64)
```

```
    # initialize utility function
    # utility function
    U = Dict()
    for s=1:P.S
        U[s] = 0
    end
    # bellman update
    # looka

    for it=1:iter
        for s=1:P.S
            # compute action-value funciton
            a_s, U_s = lookahead_large(P,U,s)
            U[s] = U_s
        end
    end

    # output policy based on utility
    policy = Dict()
    for s=1:P.S
        a_s, U_s = lookahead_large(P,U,s)
        policy[s] = a_s
    end

    return U, policy
end


# Now I need to change everything thanks to this wonderful reward function...

# i think to progress further, i need to understand how the actions 1,2,3,4
    behave better.
# maybe they act uniformly on all squares, who knows?
train = true
P_large = load_from_df_large(df_large_encoded, 500, 9, 0.95)
complete_T_large(P_large)
if train
    # there is a single missing entry in the reward table, that I just guess
    23333
    U, policy = value_iteration_large(P_large, 2000)
    @printf("%s\n", [U[s] for s=490:500])
end

outfile = "policy/large.policy"
open(outfile, "w") do io
    # first compute the policy from all the reachable states
    reachable = Dict()
    for encoded_s=1:500
        # compute the optimal action
        a_s = policy[encoded_s]
```

```
            # unencode the state
            s1 = S1Range[(encoded_s - 1) ÷ 100 + 1]
            s2 = S2Range[((encoded_s - 1) ÷ 10) % 10 + 1]
            s3 = S3Range[(encoded_s - 1) % 10 + 1]
            reachable[s1*10000+s2*100+s3] = a_s
        end
        for s=1:312020
            if s in keys(reachable)
                @printf(io, "%d\n", reachable[s])
            else
                @printf(io, "%d\n", 1)
            end
        end
    end
```