1 Description of strategy

There were 2 approaches used: a model-based MLE MDP (for small) and Q-learning (for medium, large). Both used an epsilon-greedy algorithm with exponential decay for exploration.

Small

For the small dataset, I implemented the maximum likelihood (ML) MDP in algorithm 16.1. On each simulation loop, the ML MDP would convert to a MDP with a set of estimated transition probabilities. It would take an action based on the epsilon-greedy policy, then from this action, update internal reward and occurrence counts.

As the last step in the simulation loop, the planner performed m Bellman backups using a randomized update strategy; i.e. `backup()` for m randomized states. Each backup was performed as equation (7.16) – the returned utility from each backup was the maximum utility across all $Q(s, a)$ that `lookahead()` calculated for a fixed s.

Medium, large

While the model-based ML MDP worked well for the small problem, the conversion to MDP was very expensive (see Memory considerations for more details). As a result, I switched to using the model-free approach of Q-learning, implemented as in algorithm 17.2, and simulated it using algorithm 15.9.

To handle transitions during simulation, the Q-learning model would randomly select a row in the given data that matched the given state and the action selected by the epsilon-greedy policy. If that state-action pair was not present, the model would return a randomized state available in data and a negative reward exceeding the maximum negative reward in the dataset. Therefore, state exploration was mostly limited to the states that were present in the dataset.

By only choosing states that were in the dataset, it was possible to end up in a loop where some state-action pairs ended up in the originating state. As a result, if a "loop" was detected where the same state was evaluated n times, the simulation would randomly select another state to start from.

Exploration

For all datasets, an ϵ-greedy was used with decay of $\alpha = 0.97$; i.e. every round of simulate saw update:

$$\epsilon \leftarrow \alpha\epsilon \text{ (15.2)}$$

By doing so, the goal was to gradually decrease exploration and hone in on an optimized policy.

Memory considerations

To save memory, I used `scipy.sparse`'s `lil_matrix` (on Python 3.7, so using older versions). `lil_matrix` is only compatible with `numpy`'s now-outdated matrix type, which requires shape (M, N); neither 3D matrices nor 1D vectors are allowed.

To flatten the 3D $T(s'|s, a)$ and $N(s, a, s')$ matrices, I turned each (s, a) pair into its own row in the ML MDP and MDP classes. Callers could choose between indexing action-wise, i.e. each chunk of $|S|$ rows corresponded to the same action,

$$N(s,a,s')_{\text{action-wise}} = \begin{bmatrix} -N(s=1, a=1, s')- \\ \vdots \\ -N(s=|S|, a=1, s')- \\ -N(s=1, a=2, s')- \\ \vdots \\ -N(s=|S|, a=|A|, s')- \end{bmatrix} \in \mathbb{Z}_+^{|S||A| \times |S|}$$

or state-wise, i.e. each chunk of $|A|$ rows corresponded to the same state. So converting the counts matrix $N(s, a, s')$ to a transition matrix could be done in chunks of actions or states (reshaping was not an option, due to aforementioned `lil_matrix` limitation).

Since $|S| >> |A|$ for all datasets, clearly the most time-efficient implementation was action-wise. However, the memory required to convert one chunk ($N(s, a = i, s')$) for medium datasets already exceeded what I had available.

2 Performance characteristics

small runtime: 29.6 seconds
medium runtime: 733 seconds
large runtime: 619 seconds

3 Code

```
explore.py

import logging
import numpy as np


class EGreedy():
    """Implements e-greedy policy."""

    def __init__(self,
                 epsilon: float = 0.3,
                 alpha: float = 0.97,
                 logger: logging.Logger = None):

        self.epsilon = epsilon
        self.alpha = alpha
        self.logger = logger

    def __call__(self, model, s: int, actions=None) -> int:
      """Returns next action to take.

      Args:
        model: Model to choose actions from.
        epsilon: probability of choosing non-greedy action.
        s: state to take action from.
        actions: allowable actions.

      Returns:
        action (in model representation) to take.
      """
      prev_epsilon = self.epsilon
      self.epsilon = prev_epsilon * self.alpha
      if np.random.random() < prev_epsilon:
          new_action = np.random.choice(model.actions())
          if self.logger:
              self.logger.info(f'Egreedy: randomly chose action {new_action}')

      max_a = 1
      q_max = -np.inf
      all_actions = actions if actions is not None else model.actions()
      for action in all_actions:
          q_sa = model.lookahead(s, action)
          if q_sa > q_max:
              q_max = q_sa
              max_a = action
      if self.logger:
          self.logger.info(f'Egreedy: chose action {max_a} with Q({s}, {max_a}) = {q_max}')
      return max_a
```

large.py (despite the name, also used for medium dataset)

```
import logging
import pandas
import numpy as np
import qlearning
import time
import utils

from explore import EGreedy
from scipy.sparse import lil_matrix
from typing import Tuple


def read_data(file_name: str,
              logger: logging.Logger) -> Tuple[pandas.DataFrame, np.ndarray, int]:
    """Opens file and reads data.

    Args:
      file_name: name of file to read.
      logger: logging instance.

    Returns:
      tuple of df, state space, action space.
    """

    df = pandas.read_csv(file_name + ".csv", dtype=np.int32)
    S = df.loc[:, 's'].unique()
    S.sort()
    A = df.loc[:, 'a'].unique()
    A.sort()

    logger.info(f'State space ({S.shape}): {S}')
    logger.info(f'Action space ({A.shape}): {A}')
    return (df, S, A.size)


def main():
    file_name = "large"  # or medium
    logger = utils.make_logger(file_name)
    start = time.time()

    # Set up MDP, planner.
    gamma = utils.get_gamma(file_name)
    df, S, A = read_data(file_name, logger)
    state_size = 312020  # 50000 for medium
    max_iter = 150000  # 100000 for medium
    Q = lil_matrix((state_size, A))
    qlearn_model = qlearning.QLearning(state_size,
                                        A,
                                        gamma,
                                        Q,
                                        0.25,
                                        df,
                                        known_states=S,
                                        logger=logger)

    # Run simulation and write output.
    try:
      qlearn_model.simulate(EGreedy(logger=logger), max_iter)
    except Exception as e:
      end = time.time()
      logger.critical("Unrecoverable failure")
      logger.critical(f"Elapsed time in seconds: {end - start}")
      qlearning.write_policy(file_name, qlearn_model, logger)
      raise(e)

    end = time.time()
    logger.critical(f"Elapsed time in seconds: {end - start}")
    qlearning.write_policy(file_name, qlearn_model, logger)


if __name__ == '__main__':
    main()
```

`mdp.py`: classes for model-based learning

```
import logging
import numpy as np

from scipy.sparse import lil_matrix
from typing import List, Tuple

class IndexAdapter():
    """Utility for converting 1-based indices to 0-based."""

    def __init__(self, S: int, A: int, order='action'):
        """Instantiates the instance.

        Args:
          S: Size of state space. States are assumed 1 - S, inclusive.
          A: Size of action space. States are assumed 1 - A, inclusive.
          order: Ordering of (state, action) pairs. If 'action', ordering
              in internal matrix is (action, state). If 'state', ordering is
              in internal matrix is (state, action). 'action' is more efficient
              as generally |S| >> |A|, but 'state' is more memory efficient.
        """
        self.S = S
        self.A = A
        self.order = order

    def action_index(self, a: int) -> int:
        """Returns the index of a in the internal map.

        Args:
          a: action.

        Returns:
          index of a in the internal representations.
        """
        return a - 1

    def state_index(self, s: int) -> int:
        """Returns the index of s in the internal map.

        Args:
          s: state.

        Returns:
          index of s in internal representations.
        """
        return s - 1

    def row_index(self, s: int, a: int) -> int:
        """Returns the index of (s, a) in the internal counts map."""

        s_index = self.state_index(s)
        a_index = self.action_index(a)
        if self.order == 'action':
            # actions 2, states 3
            # a0s0 0*3 + 0 = 0
            # a0s1 0*3 + 1 = 1
            # a0s2 0*3 + 2 = 2
            # a1s0 1*3 + 0 = 3
            return a_index*self.S + s_index
        elif self.order == 'state':
            # states 3, actions 2
            # s0a0  0*2 + 0 = 0
            # s0a1  0*2 + 1 = 1
            # s1a0  0*2 + 0 = 2
            # s1a1  1*2 + 1 = 3
            # s2a1  2*2 + 0 = 4
            return s_index*self.A + a_index
        else:
            raise ValueError(f'Unsupported ordering: {self.order}')


class MDP(IndexAdapter):
    """Defines a MDP."""

    def __init__(self,
                 S: int,
                 A: int,
                 T,
                 R,
                 logger: logging.Logger = None,
                 order = 'action'):
        """Instantiates an instance.

        Args:
          S: Size of state space. Assumes states are 1 to S, inclusive.
          A: Size of action space. Assumes states are 1 to S, inclusive.
          T: Transition matrix.
          R: Reward matrix (i = state, j = action).
        """
        super().__init__(S, A, order = order)
        self.T = T
        self.R = R
        self.logger = logger

    def transition_prob(self, s: int, a: int, next_s: int) -> float:
        i = self.row_index(s, a)
        return self.T[i, next_s]

    def TR(self, s: int, a: int) -> Tuple[int, float]:
        """Samples transition and reward.

        Args:
          s: state to take action from.
          a: action to take.

        Returns:
          Tuple of next state, reward.
        """
        s_index = self.state_index(s)
        a_index = self.action_index(a)
        i = self.row_index(s, a)
        probs = np.array(self.T[i, :].toarray()).reshape((self.S,))
        prob_sum = np.sum(probs)
        if prob_sum != 1 and self.logger is not None:
            self.logger.critical(f'Probabilities for T({s}, {a}) sum to {prob_sum}.')
        next_state = np.random.choice(np.arange(1, self.S + 1),
                                      p = probs)
        return (next_state, self.R[s_index, a_index])


class MaximumLikelihoodMDP(IndexAdapter):
    """Class defining an MLE MDP."""

    def __init__(self,
                 S: int,
                 A: int,
                 gamma: float,
                 planner,
                 logger_name: str = None,
                 order = 'action'):
        """Instantiates a new instance.

        Args:
          S: Size of state space. States are assumed 1 - S, inclusive.
          A: Size of action space. States are assumed 1 - A, inclusive.
          gamma: discount factor.
          planner: Planner.
          logger_name: name of logger
          order: order
        """

        super().__init__(S, A, order = order)
        self.gamma = gamma
        self.rho = lil_matrix((S, A))
        self.planner = planner
        # Maps action-state pairs to potential next states.
        self.N = lil_matrix((S * A, S))
        self.U = lil_matrix((S, 1))

        if logger_name:
            self.logger = logging.getLogger(logger_name)

    def __log(self, msg: str, level = logging.INFO):
        """Wrapper around logging."""
        if self.logger:
            self.logger.log(level, msg)

    def actions(self) -> List[int]:
        """Returns actions for this MDP."""

        return np.arange(1, self.A + 1)

    def get_utility(self, s: int) -> float:
        """Returns utility for state s."""

        s_index = self.state_index(s)
        return self.U[s_index, 0]

    def set_reward(self, s: int, a: int, reward: float):
        """Sets reward for state s and action a."""

        s_index = self.state_index(s)
        a_index = self.action_index(a)
        self.rho[s_index, a_index] = reward

    def set_utility(self, s: int, utility: float):
        """Sets utility for state s."""

        s_index = self.state_index(s)
        self.U[s_index, 0] = utility

    def states(self) -> List[int]:
        """Returns sorted list of states for the model."""

        return np.arange(1, self.S + 1)

    def add_count(self, s: int, a: int, next_s: int):
        """Adds 1 to count matrix.

        Args:
          s: state
          a: action
          next_s: next state that results from state-action pair.
        """
        next_s_index = self.state_index(next_s)
        i = self.row_index(s, a)
        self.N[i, next_s_index] = self.N[i, next_s_index] + 1

    def lookahead(self, s: int, a: int) -> float:
        """Returns utility of performing action a from state s.

        Uses Bellman equation.

        Args:
          s: current state
          a: current action

        Returns:
          Utility given current state s and taking action a.
        """
        s_index = self.state_index(s)
        a_index = self.action_index(a)
        i = self.row_index(s, a)
        n = np.sum(self.N[i, :])
        if n == 0:
            return 0.0

        r = self.rho[s_index, a_index] / n
        trans_prob = lambda next_state : self.N[i, self.state_index(next_state)] / n
        utilities = [trans_prob(s_next)*self.get_utility(s_next) for s_next in self.states()]
        utility = r + self.gamma * np.sum(utilities)
        return utility

    def backup(self, s: int) -> float:
        """Returns utility of optimal action from state s.

        Returns max of lookahead().

        Args:
          s: current state

        Returns:
          Utility of optimal action.
        """

        return np.max([self.lookahead(s, a) for a in self.actions()])

    def update(self, s: int, a: int, r: float, next_s: int):
        """Updates model based on given parameters.

        Args:
          s: current state
          a: action to take from s
          r: reward for taking action a in state s
          next_s: next state to go to.
        """
        s_index = self.state_index(s)
        a_index = self.action_index(a)
        next_s_index = self.state_index(next_s)

        # N(s, a, s') += 1, rho[s, a] += r
        self.add_count(s, a, next_s)
        self.rho[s_index, a_index] = self.rho[s_index, a_index] + r
        self.planner.update(self, s_index, a_index, r, next_s_index)

    def to_mdp(self) -> MDP:
        """Converts this instance to an equivalent MDP."""

        # N_sa is a matrix holding N(s,a). Each row is a state, each col is an
        # action.
        N_sa = self.N.sum(1)
        N_sa = N_sa.reshape((self.A, self.S))
        N_sa = np.array(N_sa.transpose())  # |S| x |A|

        # R(s,a) = rho[s,a] / N(s,a) if N(s, a) > 0 else 0
        R = lil_matrix(
            np.divide(self.rho.toarray(),
            N_sa,
            out=np.zeros(self.rho.shape),
            where=(N_sa != 0)))

        # T(s,a) = N(s, a, s') / N(s, a) if N(s, a) > 0 else 0
        T = lil_matrix(self.N.shape)

        if self.order == 'action':
            for a in self.actions():
                # Get counts for all (state, action) pairs with action == a.
                action_index = self.action_index(a)
                divisor = N_sa[:, action_index]  # N(s, a == a), |S| x 1 matrix

                # Get all N(s, a = a, s').
                start_index = self.row_index(1, a)
                end_index = self.row_index(self.S, a) + 1
                counts = self.N[start_index:end_index, :]

                # Perform division along the row, so each entry of divisor
                # divides a row in counts. All the transposing is empirical.
                T_sa = np.divide(
                    counts.toarray().T,
                    divisor,
                    out=np.zeros((self.S, self.S)),
                    where=(divisor != 0))
                T[start_index:end_index, :] = T_sa.T
        elif self.order == 'state':
            for a in self.actions():
                for s in self.states():
                    a_index = self.action_index(a)
                    s_index = self.state_index(s)
                    divisor = N_sa[s_index, a_index]
                    i = self.row_index(s, a)
                    if divisor == 0:
                        T[i, :] = 0
                        continue
                    T[i, :] = T[i, :] / divisor
                self.__log(f'Wrote T(s\'| a = {a}, s)')
        else:
            raise ValueError(f'Unsupported ordering: {self.order}')

        self.__log(f'Wrote T: {T}')
        return MDP(self.S, self.A, T, R, logger=self.logger, order=self.order)

    def simulate(self, policy, s: int) -> Tuple[int, int]:
        """Simulates one round using policy.

        Args:
          policy: callable policy to follow.
          s: start state.

        Returns:
          tuple of (action taken, next state)
        """
        mdp = self.to_mdp()
        a = policy(self, s)
        next_s, r = mdp.TR(s, a)
        self.__log(f"Update after simulation: (s, a, r, s'): ({s}, {a}, {r}, {next_s})")
        self.update(s, a, r, next_s)
        return (a, next_s)
```

`planner.py`: abstract class for planners

```
import abc

from mdp import MaximumLikelihoodMDP


class Planner(abc.ABC):
    """Abstract class defining a planner."""
    @abc.abstractmethod
    def update(self, model: MaximumLikelihoodMDP, s: int, a: int, r: float, next_s: int):
        """Mutates the utility of state s in model.

        Not all arguments may be used, depending on the implementation.

        Args:
          model: Model to update.
          s: The utility of this state will be updated.
          a: action to take from this state.
          r: reward given for taking action a from state s, and ending in state
            next_s.
          next_s: state ending up in after taking action a from state s.
        """
        pass
```

`qlearning.py`: class for q-learning

```
import logging
import mdp
import numpy as np
import pandas
```

```
from typing import List, Set, Tuple

class QLearning(mdp.IndexAdapter):
    def __init__(self,
                 S: int,
                 A: int,
                 gamma: float,
                 Q,
                 alpha: float,
                 df: pandas.DataFrame,
                 known_states: Set[int] = None,
                 logger: logging.Logger = None):
        self.S = S
        self.A = A
        self.gamma = gamma
        self.Q = Q
        self.logger = logger
        self.alpha = alpha
        self.df = df
        self.known_states = known_states

    def states(self) -> List[int]:
        return np.arange(1, self.S + 1)

    def actions(self) -> List[int]:
        return np.arange(1, self.A + 1)

    def __choices(self) -> List[int]:
        return list(self.known_states) if self.known_states is not None else self.states()

    def next_state(self, s: int, a: int) -> Tuple[int, float]:
        """Returns tuple of next state, reward."""

        queried_df = self.df.query('s == @s & a == @a')
        if queried_df.size == 0:
            if self.logger is not None:
                self.logger.warning(f'No results found for (s = {s}, a = {a}).')
            return (np.random.choice(self.__choices()), -15.0)  # -225 for medium, -15 for large
        total_rows = queried_df.shape[0]
        random_row = np.random.choice(total_rows)
        row = queried_df.iloc[random_row]
        next_s = row.loc['sp']
        r = row.loc['r']
        if self.logger is not None:
            self.logger.info(f'Next action: {next_s}, r: {r}')
        return (next_s, r)

    def lookahead(self, s:int, a:int):
        return self.Q[self.state_index(s), self.action_index(a)]

    def update(self, s: int, a: int, r: float, next_s: int):
        s_index = self.state_index(s)
        a_index = self.action_index(a)
        next_s_index = self.state_index(next_s)
        discounted_max = self.gamma*np.max(self.Q[next_s_index, :].toarray())
        q_sa = self.Q[s_index, a_index]
        self.Q[s_index, a_index] = q_sa + self.alpha*(r + discounted_max - q_sa)

    def simulate(
            self,
            policy,
            max_iter: int):
        choices = self.__choices()
        s = np.random.choice(choices)
        loop_count = 0
        for _ in range(max_iter):
            a = policy(self, s)
            next_s, r = self.next_state(s, a)
            self.update(s, a, r, next_s)

            # Loop detection.
            if next_s == s:
                loop_count += 1
            else:
                loop_count = 0
            if loop_count > 10:
                next_s = np.random.choice(choices)
                if self.logger is not None:
                    self.logger.warning(f'Detected loop, going to random state {next_s}')

            s = next_s


def write_policy(file_name: str, model: QLearning, logger: logging.Logger = None):
    with open(file_name + '.policy', 'w') as f:
        for state in model.states():
            # Add one for action offset.
            Q_s = model.Q[model.state_index(state), :].toarray().reshape(-1)
            best_action = np.argmax(Q_s) + 1
            if Q_s.any() != 0:
                logger.info(f"Q[{state}, :] = {Q_s}; best value is {best_action}")
            f.write("{}\n".format(best_action))
```

`randomizedupdate.py`: implements a randomized update strategy

```
import logging
import numpy as np

from mdp import MaximumLikelihoodMDP
from planner import Planner


class RandomizedUpdate(Planner):
    """Implements randomized updates."""

    def __init__(self, m: int, logger_name: str = None):
        """Initializes the instance.

        Args:
          m: number of updates.
        """
        self.m = m
        if logger_name:
            self.logger = logging.getLogger(logger_name)


    def __log(self, msg: str, level = logging.INFO):
        """Wrapper around logging."""
        if self.logger:
            self.logger.log(level, msg)


    def update(self, model: MaximumLikelihoodMDP, s: int, a: int, r: int, next_s: int):
        new_states = np.random.choice(model.states(), self.m, replace=False)
        new_states = np.concatenate(([s], new_states), axis=None)
        assert new_states[0] == s, "Needs to work this way"
        for state in new_states:
            u = model.get_utility(state)
            model.set_utility(state, model.backup(state))
            new_utility = model.get_utility(state)
            self.__log(f"Updated utility for state {state}: {u} to {new_utility}")
```

`small.py`: runs model-based RL with small dataset

```
import logging
import pandas
import numpy as np
import time
import utils

from explore import EGreedy
from mdp import MaximumLikelihoodMDP
from randomizedupdate import RandomizedUpdate
from typing import Tuple


def read_data(file_name: str,
              logger: logging.Logger) -> Tuple[pandas.DataFrame, int, int]:
    """Opens file and reads data.

    Args:
      file_name: name of file to read.
      logger: logging instance.

    Returns:
      tuple of df, state space, action space.
    """

    df = pandas.read_csv(file_name + ".csv", dtype=np.int32)
    S = df.loc[:, 's'].unique()
    S.sort()
    A = df.loc[:, 'a'].unique()
    A.sort()
    logger.info(f'State space ({S.shape}): {S}')
    logger.info(f'Action space ({A.shape}): {A}')
    return (df, S.size, A.size)


def simulate(
        model: MaximumLikelihoodMDP,
        policy,
        max_iter: int) -> np.ndarray:
    """Simulates using model, policy.

    Args:
      model: Model to simulate.
      policy: Policy to follow.
      max_iter: maximum iterations.

    Returns:
      trajectory taken in (state, action) pairs.
    """

    s = np.random.choice(model.states())
    trajectory = np.zeros((max_iter, 2))
    for i in range(max_iter):
        a, next_s = model.simulate(policy, s)
        trajectory[i][0] = s
        trajectory[i][1] = a
        s = next_s
    return trajectory


def main():
    file_name = "small"
    logger = utils.make_logger(file_name)
    start = time.time()

    # Set up MDP, planner.
    gamma = utils.get_gamma(file_name)
    df, S, A = read_data(file_name, logger)
    update_count = 8  # TODO(kathuan): tune this
    max_iter = 300  # TODO(kathuan): tune this
    planner = RandomizedUpdate(update_count, file_name)
    model = MaximumLikelihoodMDP(S, A, gamma, planner, file_name)
    utils.set_counts(model, df)

    # Run simulation and write output.
    _ = simulate(model, EGreedy(), max_iter)
    end = time.time()
    logger.critical(f"Elapsed time in seconds: {end - start}")
    utils.write_model_policy(file_name, model)


if __name__ == '__main__':
    main()
```

`utils.py`: misc. utility functions that all datasets may need

```
import logging
import pandas

from explore import EGreedy
from mdp import MaximumLikelihoodMDP


def get_gamma(file_name: str) -> float:
    """Returns discount factor for supported filenames.

    Values taken from course website.

    Args:
      file_name: supported filenames are "small", "medium", or "large".

    Returns:
      discount factor
    """
    if file_name == "small":
        return 0.95
    elif file_name == "medium":
        return 1.0
    elif file_name == "large":
        return 0.95
    else:
        raise ValueError(f"Unsupported file: {file_name}")


def set_counts(model: MaximumLikelihoodMDP,
               df: pandas.DataFrame):
    """Sets counts and rewards for the given model."""

    for _, row in df.iterrows():
        model.add_count(row['s'], row['a'], row['sp'])
        model.set_reward(row['s'], row['a'], row['r'])


def write_model_policy(file_name: str,
                 model: MaximumLikelihoodMDP,
                 logger: logging.Logger = None):
    """Writes policy to file <file_name>.policy.

    Each row i in the file corresponds to the action taken for the state i.

    Args:
      model: Model
    """
    egreedy = EGreedy(epsilon = 0)
    logger.info(f'Writing to file {file_name}.policy')
    with open(file_name + '.policy', 'w') as f:
        for state in model.states():
            best_action = egreedy(model, state)
            f.write("{}\n".format(best_action))
```

```
def make_logger(logger_name: str, level = logging.INFO) -> logging.Logger:
    logger = logging.getLogger(logger_name)
    logger.setLevel(level)
    fh = logging.FileHandler(logger_name + '.log')
    fh.setLevel(level)
    formatter = logging.Formatter('%(asctime)s - %(levelname)s: %(message)s')
    fh.setFormatter(formatter)
    logger.addHandler(fh)
```