IN-STK5000 project 2: Medical project

Espen H. Kristensen (espenhk)

November 30, 2018

Exercise 1 (Measuring utility)

Estimating the utility using the historical policy, can be found simply as

$$\hat{E}(U) = \sum_t r(a_t, y_t)/T$$

The `estimate_utility()` method can be found in `HistoricalRecommender.py`, and is as follows. Note that the `self.hist_data` etc come from the training of the model using `fit_treatment_outcome()`, so this function must have been called before calling `estimate_utility()` without a policy.

```
    ## Estimate the utility of a specific policy from historical
    data.
    def estimate_utility(self, data, actions, outcomes, policy=None
    ):
        if policy == None: # use historical data
            T = len(self.hist_data)
            estimate = 0
            for i in range(T):
                r_t = self.reward(self.hist_actions[i],
                                  self.hist_outcomes[i])
                estimate += r_t
            estimate /= T
            return estimate
        else:
            return policy.estimate_utility(data, actions, outcomes)
```

I've made some additions to the test bench `TestRecommender.py`, where the first part the estimated utility. Running it gives:

```
---- Estimating utility for historical recommender on historical
    data ----
k neighbors: 2
Estimated utility: 0.1191
```

Exercise 2 (Improved policies)

The functions relevant to this exercise can be found in `ImprovedRecommender.py`. To build a model, `fit_treatment_outcome()` is used to fit $P(y|a, x)$, that is the probability of an outcome given historical data of patient information and actions (treatments given). This is done using a K-Nearest Neighbors-classifier.

I've created a bootstrap to pick the best k, but this always suggests $k = 1$, despite the fact that trial and error suggests $k = 2$ is better for the historical recommender, and $k = 25$ for this improved recommender. I'm not sure if this is because my bootstrap isn't designed correctly, or because just running the test bench with different k values will give some form of overfitting, but as I can't find a problem in the bootstrap I'm leaning towards the latter.

Note the flag `use_bootstrap`, which can be changed to either use the bootstrap (True) or use a manually set value for k (False). Also note that the bootstrap takes a few minutes to run, which is why it's disabled by default. So, the model is fit and saved as `self.model` by

```
    ## Fit a model from patient data, actions and their effects
    ## Here we assume that the outcome is a direct function of data and actions
    ## - sets model to self.model
    ## We use a K-Nearest Neighbors classifier to fit P(y|a,x)
    def fit_treatment_outcome(self, data, actions, outcome):
        # save as historical data
        self.hist_data = data
        self.hist_actions = actions
        self.hist_outcomes = outcomes

        n_samples = 5
        X = np.concatenate((data, actions), axis=1)
        outcome = outcome.flat

        K = 25
        k_accuracy = np.zeros(K)
        # This is a bootstrap to choose k. It consistently recommends
        # k = 1 for both Historical and ImprovedRecommender, but
        # trial and error suggests k = 2 and k = 25, respectively
        use_bootstrap = False
        if use_bootstrap:
            print("Bootstrap for k begin, K = %d" % K)
            for k in range(K):
                print("testing k = %d" % k)
                for i in range(n_samples):
                    train_set, test_set = train_test_split(data, test_size = 0.2)
                    # pick len(train_set) indexes in (0 , len(train_set)-1)
                    train_sample_index = np.random.choice(len(train_set),
                                                          len(train_set))
                    test_sample_index = np.random.choice(len(test_set),
                                                         len(test_set))
                    # use picked indexes to pick data points with
                    # replacement for bootstrap
                    k_model = KNeighborsClassifier(n_neighbors=k+1).fit(data[train_sample_index], actions[train_sample_index])
                    k_accuracy[k]+= accuracy_score( actions[test_sample_index],
                                k_model.predict(data[test_sample_index]) )
                print(k_accuracy)
                k_accuracy[k] /= n_samples
```

```
                print(k_accuracy)
            k = np.argmax(k_accuracy[1:]) + 1

            print("Bootstrap END, k = %d" % k)
            # Bootstrap end
        else: # don't use bootstrap for k
            # hard set k to avoid running bootstrap all the time
            k = 25
        print("k neighbors: %d" % k)

        self.model = KNeighborsClassifier(n_neighbors = k).fit(X,
    outcome)
```

and estimated expected utility found by

```
    ## Estimate the utility of a specific policy from historical
    data.
    ## If the policy argument is None, use the improved policy
    def estimate_utility(self, data, actions, outcome, policy=None)
    :
        if policy is None:
            if data.ndim == 1: # only one user, action and outcome
                proba = self.predict_proba(data, actions)[outcome]
                rewa = self.reward(actions, outcome)
                result = proba * rewa
                return result
            elif data.ndim == 2: # full dataset
                estimate = 0
                T = len(data)
                for i in range(T):
                    res = self.estimate_utility(data[i], actions[i
    ], outcome[i])
                    estimate += res
                estimate /= T
                return estimate

        else:
            return policy.estimate_utility(data, actions, outcome)
```

The second part of my testbench calculates this, and gives:

```
---- Estimating utility for improved recommender on historical data
     ----
k neighbors: 25
Estimated utility: 0.0324
```

Exercise 3 (Online policy testing)

1. HistoricalRecommender

The historical policy π_0 we seek is $P(a|x)$, that is the probability of seeing an action given data about a patient. Similarly to the improved recommender, I use a K-NN classifier and fit a model to this using `fit_treatment_outcome()`. Essentially, the only difference between the historical and the improved versions of this function is the arguments passed to `fit()` at the end of the function. Note also that a different k is used by default, since $k = 2$ gave better results for this classifier. The function is thus

```
    ## Fit a model from patient data, actions and their effects
    ## Here we assume that the outcome is a direct function of data
     and actions
    ## - sets model to self.model
    ## We use a K-Nearest Nea,ighbors classifier to fit P(a|x)
    def fit_treatment_outcome(self, data, actions, outcomes):

        # save as historical data
        self.hist_data = data
        self.hist_actions = actions
        self.hist_outcomes = outcomes

        actions = actions.flat

        K = 15
        k_accuracy = np.zeros(K)
        # Change this flag to use the bootstrap.
        use_bootstrap = False
        # This is a bootstrap to find k.
        # See notes on issues with this in my report.
        if use_bootstrap:
            n_samples = 5
            print("Bootstrap for k begin, K = %d" % K)
            for k in range(K):
                print("testing k = %d" % k)
                for i in range(n_samples):
                    train_set, test_set = \
                        train_test_split(data, test_size = 0.2)
                    # pick len(train_set) indexes
                    train_sample_index = \
                        np.random.choice(len(train_set),
                                         len(train_set))
                    test_sample_index = \
                        np.random.choice(len(test_set),
                                         len(test_set))
                    # use picked indexes to pick data points
                    # with replacement for bootstrap
                    k_model = KNeighborsClassifier(n_neighbors=k+1)
    .fit(data[train_sample_index], actions[train_sample_index])
                    k_accuracy[k] += accuracy_score( actions[
    test_sample_index], k_model.predict(data[test_sample_index]) )
                k_accuracy[k] /= n_samples
            k = np.argmax(k_accuracy[1:]) + 1

            print("Bootstrap END, k = %d" % k)
        else: # don't use bootstrap for k
            # hard set k to avoid running bootstrap all the time
            k = 2
        print("k neighbors: %d" % k)

        self.model = KNeighborsClassifier(n_neighbors = k).fit(data
    , actions)
```

To compare the result from the testbench with the estimates previously found, I've changed the default reward function to reflect the one in the exercise text, and also divide the total reward by number of steps T. All test bench runs are with 1000 samples, unless otherwise noted. Running the testbench with `HistoricalRecommender` chosen, gives output

```
==== Start of original test bench, using HistoricalRecommender ====
---- Testing with only two treatments -----
Setting up simulator
Setting up policy
Fitting historical data to the policy
k neighbors: 2
Running an online test
Testing for  1000 steps
Total reward: 0.2091
Final analysis of results
time taken to test: 47.652 seconds
--- Testing with an additional experimental treatment and 126 gene
    silencing treatments ---
Setting up simulator
Setting up policy
Fitting historical data to the policy
k neighbors: 2
Running an online test
Testing for  1000 steps
Total reward: 0.2006
Final analysis of results
time taken to test: 47.031 seconds
```

2. ImprovedRecommender

This recommender is already described through exercise 2. Running the main portion of the test bench with it chosen, gives

```
==== Start of original test bench, using ImprovedRecommender ====
---- Testing with only two treatments -----
Setting up simulator
Setting up policy
Fitting historical data to the policy
k neighbors: 25
Running an online test
Testing for  1000 steps
Total reward: 0.3647
Final analysis of results
time taken to test: 16.020 seconds
--- Testing with an additional experimental treatment and 126 gene
    silencing treatments ---
Setting up simulator
Setting up policy
Fitting historical data to the policy
k neighbors: 25
Running an online test
Testing for  1000 steps
Total reward: -0.3236
Final analysis of results
time taken to test: 1048.353 seconds
```

3. Comment differences

Clearly, both recommenders perform better during online testing than when tested on the historical data. This seems quite an odd result, as we would expect the model to perform better on the data it was trained on, rather than these unseen data. We fortunately see that the improved recommender does better than the historical one on the first testbench case.

For the second case, it seems the exercise text does not expect this case to be used until exercise 4. It can still be useful to see what they gave. We see that neither recommenders are built for this and so they "fail" in one way each: The `HistoricalRecommender`'s `recommend()`-function only considers the probability of the model choosing action 0, choosing 0 if it's above 0.5 and 1 if not. This means it's less affected by there suddenly being a third option, as it will pool choices 1 and 2 together.. The ImprovedRecommender however is thrown terribly off, which is understandable as the data has a different nature to what it was trained on. This can be a useful baseline to have before exercise 4.

Exercise 4 (Adaptive experiments)

The adaptive recommender has been implemented in `AdaptiveRecommender.py`. It is at its core the same as `ImprovedRecommender`, except for the `observe()` method which has been implemented here. It is implemented simply by appending the new data to that previously seen, and refitting the model. It refits in batches, batch size is set by the `refit_every` variable – a value of 1 means refit with every datapoint, 100 means refit every one hundredth etc.

A more efficient way of handling it would be to do some form of incremental learning, but `scikit-learn` does not support this for a KNN classifier. They do however have a `partial_fit` API which some classifiers implement, so if one were to change to one of these you could do `self.model.partial_fit(X, y)` to only fit the new data in with the historical. This should be computationally quite a bit more efficient, but I have not had time to change my implementation to one of these.

Running the `AdaptiveRecommender` through our test bench, we get

```
==== Start of original test bench, using AdaptiveRecommender ====
---- Testing with only two treatments -----
Setting up simulator
Setting up policy
Fitting historical data to the policy
Running an online test
Testing for  1000 steps
Total reward: 0.3759
Final analysis of results
time taken to test: 179.616 seconds
--- Testing with an additional experimental treatment and 126 gene
    silencing treatments ---
Setting up simulator
Setting up policy
Fitting historical data to the policy
Running an online test
```

```
Testing for  1000 steps
Total reward: -0.0078
Final analysis of results
time taken to test: 365.210 seconds
```

This table summarizes the estimated utilities found:

	hist_data	TB case 1	TB case 2
Historical	0.1191	0.2091	0.2006
Improved	0.0324	0.3647	-0.3236
Adaptive (1k, 1)	–	0.3759	-0.0078
Adaptive (10k, 100)	–	0.4007	0.2653

Adaptive (1k, 1) is the adaptive recommender tested with 1000 samples and refitting for every new data point, whereas Adaptive (10k, 100) has 10 000 samples and refits every 100 data points. First we'll consider only the first of these, as it's the most comparable with the others.

We see that in the first testbench case (two treatments available), the adaptive recommender does marginally better than the improved, which outperforms the historical. In the second testbench case (additional treatment available), both improved and adaptive struggle, but we see that adaptive does quite a lot better than improved here. Note that there is no point in testing the adaptive recommender on historical data, as it would perform equally to improved. Thus, I would say the adaptive recommender shows considerable promise, and might catch up to the historical if given enough data.

And indeed, when we increase to 10 000 samples, refitting every 100, we see another improvement in the first test case. But more importantly we see a great improvement in the new treatment case. We can speculate that starting without previous data would give a better result, again at least as the amount of data increases, but I've not had the time to test what magnitude of data would be necessary for this. The issue with starting from scratch would be the initial batches, where the distribution in the data would be highly random.

This is how far I got with my analysis. Thanks for providing an interesting and challenging course, that I wish I had more time to work on!

Attached:

- Recommender classes: `HistoricalRecommender.py`, `ImprovedRecommender.py`, `AdaptiveRecommender.py`
- Modified test bench: `TestRecommender.py`
- Generating matrices files (unmodified): `big_generating_matrices.mat`, `generating_matrices.mat`
- Data generator (unmodified): `data_generation.py`

In addition to this, the data files containing historical data need to be in the same position relative to the project files as it is in the GitHub-repository.