Prize recipient documentation

Power Laws: Optimizing Demand-side Strategies

Guillermo Barbadillo Villanueva
guillermobarbadillo@gmail.com

II. Code submission and result reproducibility

On this challenge providing the code is very easy because we already have to provide the code of the battery controller previously.

In addition to that I’m also providing a jupyter notebook that was used to compute the coefficients used by the battery controller script.

A txt with the requirements is also provided, the list is really short.

III. Model documentation and write-up

1. Who are you (mini-bio) and what do you do professionally?

I finished a master in Electronic Engineering 5 years ago. Since June 2014 I have been studying and practicing Artificial Intelligence on my own. First on my free time and 3 years ago I started working also on AI. My expertise is on applying deep learning to computer vision, but I have also worked with structured data and text.

2. High level summary of your approach: what did you do and why?

I have prepared a nice presentation that is attached on the mail. There you would find all the insights of my approach alongside many visualizations.
My solution is based on:

- Simplification of the period
- Dynamic programming
- Intelligent pruning of the actions

3. **Copy and paste the 3 most impactful parts of your code and explain what each does and how it helped your model.**

```
@lru_cache(maxsize=CACHE_MAX_SIZE)
    def _get_target_battery_range(self, current_charge, block_idx):
        balance_match = self.balance_matches[block_idx]
        target_battery_range = [current_charge + balance_match]

        max_battery_charge, max_battery_discharge =
self.max_charge_variations[block_idx]
        if balance_match > 0: # The system is giving energy
            pass
        else: # The system is demanding energy
            # # The following line only has sense if higher prices
are expected on following epochs
            if self.is_wait_condition[block_idx]:
                # target_battery_range.append(current_charge)
                if block_idx < len(self.period_summary) - 1:
                    next_balance_match =
self.balance_matches[block_idx+1]
                    target_battery = - next_balance_match
                    # Need to check that it is possible to achieve
that
                    battery_change = target_battery - current_charge
                    if target_battery - current_charge > 0:
                        battery_change = np.minimum(battery_change,
max_battery_charge)
                    else:
                        battery_change = np.maximum(battery_change,
max_battery_discharge)
                    target_battery_range.append(current_charge +
battery_change)

        if self.is_low_price[block_idx]:
            target_battery_range.append(current_charge +
max_battery_charge)

        optimize_end = self.allow_end_optimization and
len(self.period_summary) - block_idx < 4
        if optimize_end:
            target_battery_range.append(current_charge +
max_battery_discharge)
            target_battery_range.append(current_charge)

        target_battery_range = np.clip(target_battery_range, 0,
1).round(5)
        target_battery_range = np.unique(target_battery_range)
        return target_battery_range
```

This part of the code is relevant because it is describing the available actions for the battery. Depending on the context more actions are allowed. Having the smallest possible set of actions increases the speed of the algorithm. Having more available actions could improve slightly the score but at the cost of a much slower algorithm. Notice the decorator at the top of the function that allows to do value caching. The set of available actions are described on the presentation attached.

```
@lru_cache(maxsize=CACHE_MAX_SIZE)
    def _find_best_cost_and_policy(self, initial_charge,
initial_epoch, end_epoch):
        if initial_epoch == end_epoch:
            return 0, []

        target_charge_range =
self._get_target_battery_range(initial_charge, initial_epoch)
        costs = []
        paths = []
        for target_charge in target_charge_range:
            cost = self._block_cost(initial_charge, target_charge,
initial_epoch)
            ret = self._find_best_cost_and_policy(target_charge,
initial_epoch+1, end_epoch)
            costs.append(cost + ret[0])
            paths.append([target_charge] + ret[1])

        min_index = np.argmin(costs)
        return costs[min_index], paths[min_index]
```

This code snippet is relevant because it implements the dynamic programming algorithm that it is used for optimizing the cost. Notice the decorator at the top of the function that allows to do value caching, necessary for dynamic programming.

```python
def get_fine_grain_policy_for_timestep(timestep_idx, policy,
balances,
                                       timestep_balance, battery,
block_length,
                                       current_charge):
    """
    It only works for the initial block

    Returns None when is not able to compute the policy
    """
    block_idx = 0
    initial_charge = policy[block_idx]
    final_charge = policy[block_idx+1]
    battery_change = final_charge - initial_charge
    balance = balances[block_idx]

    remaining_steps = block_length - timestep_idx
    if remaining_steps < 1:
        # Then the block has ended
        return None

    if timestep_balance != 0 and timestep_balance * balance < 0:
        # Then the balance of the system has changed
        return None

    if battery_change == 0:
        return current_charge
    elif balance*battery_change < 0:
        # Then the changes are opposite
        return _sincronize_battery_and_system_for_timestep(
            initial_charge=initial_charge, final_charge=final_charge,
balance=balance,
            current_charge=current_charge,
            battery=battery, timestep_balance=timestep_balance)
    else:
        return (final_charge - current_charge)/remaining_steps +
current_charge
```

This final snippet shows how the high level policy is transformed into actions for each timestep. We can see that three situations are considered:

- No change is done to the battery
- The change of the battery is opposite to the system energy balance

- The change of the battery has the same sign as the system energy balance

4. What are some other things you tried that didn't necessarily make it into the final workflow (quick overview)?

At the start of the challenge I tried using genetic algorithms to optimize the battery policy. However they were very slow and did not capture correctly the essence of the problem.

I also tried using neural networks to improve the accuracy of the forecasts. However there was a severe overfit and simple linear regression worked better. I could only improve the accuracy by a 2%.

5. Did you use any tools for data preparation or exploratory data analysis that aren't listed in your code submission?

I used the typical tools for visualization in python such as matplotlib and seaborn.

6. How did you evaluate performance of the model other than the provided metric, if at all?

I only used the metric provided on the challenge. However execution time of the simulation was also a very important factor for taking decisions.

7. Anything we should watch out for or be aware of in using your model (e.g. code quirks, memory requirements, numerical stability issues, etc.)?

The battery_controller is quite simple in terms of requirements. However the notebook that I used for computing the coefficients for improving the accuracy of the forecast has very unstable results. Every time I run it gives different coefficients. I think the solution would be cleaner without those coefficients, and the gain in score is very small (-0.1922 vs -0.1920). However I included them because in the competition that could make the difference between winning or losing.

8. Do you have any useful charts, graphs, or visualizations from the process?

Yes, in the presentation attached there are many useful visualizations.

9. If you were to continue working on this problem for the next year, what methods or techniques might you try in order to build on your work so far? Are there other fields or features you felt would have been very helpful to have?

I think that the most obvious way of improving my solution is modifying how it handles the uncertainty in the forecast. The algorithm was designed using perfect information, and it turned to work quite well also under forecast with uncertainty.
However I believe that a design that from the beginning makes the assumption that the forecast are not accurate could achieve better scores.