Report 16/12/2020

Alessandro Franca

Modifiche al codice di Mattia *PodAggregator:*

- *pod.py:* Inserito attributo *self.to_optimize* utile per sapere se il Pod necessita di una prima ottimizzazione locale (Come concordato è *True* se il pod contiene più di un profilo)
- *api.py*: integrazione di APIs, chiamate dai vari endpoint della Webapp, in particolare:
    - `get_baselines_from_config:` prende in ingresso un json contenente la configurazione di pod selezionata dall'utente, e ritorna i profili completi di baselines (per ora i dati sono presi da un *dict* di prova).
      Dal momento che lo script di cui mi aveva parlato prende solo un *pod_id* + *data* alla volta, questa api è da modificare. Volevo infatti ragionare insieme a lei se conviene parallelizzare le richieste da inoltrare a Libra (rischiando un sovraccarico), o se conviene aspettare che implementino loro una API che consenta di recuperare le varie baseline a partire da più *pod_id,* con una singola chiamata)
    - `optimize_and_aggregate:` prende in ingresso un json contenente la configurazione completa di baseline. Crea i Pod, effettua l'ottimizzazione locale, e l'aggregazione.
    - `local_optimization:` prende in ingresso un json contenente la configurazione completa di baseline. Crea i Pod, effettua l'ottimizzazione locale, e salva i risultati in un oggetto `LocalOptimizationResult` (vedi seguito)
    - `aggregate:` prende in ingresso la rappresentazione json di un oggetto `LocalOptimizationResult` ed effettua l'aggregazione a partire da questo.

      Tutte le Responses vengono fornite come json.

- *local_optimization_result.py:*
    - Nuovo classe `LocalOptimizationResult` inserita.
      Necessaria per salvare i dati relativi a tutte le ottimizzazioni locali ai vari pod (se non da ottimizzare, vengono presi in considerazione la flessibilità minima e massima del pod in questione) in modo da poter poi ripartire da questo per l'aggregazione.
      Le ottimizzazioni vengono salvate in un array sotto forma di *dict*, nel formato indicato nella figura sottostante.
      Non viene salvato il Pod in sè, ma soltanto la sua flessibilità e i costi.

```
def populate(self, pods: [Pod]):
    for pod in pods:
        if not pod.needs_local_optimization():
            element = {
                'minimized': {
                    'flexibility': pod.get_flexibility('min'),
                    'cost': [0] * 96
                },
                'maximized': {
                    'flexibility': pod.get_flexibility('max'),
                    'cost': [0] * 96
                },
            }
            self.optimizations.append(element)
        else:
            element = {
                'minimized': {
                    'flexibility': pod.solver.results['minimized']['grid'],
                    'cost': pod.solver.results['minimized']['cost']
                },
                'maximized': {
                    'flexibility': pod.solver.results['maximized']['grid'],
                    'cost': pod.solver.results['maximized']['cost']
                },
            }
            self.optimizations.append(element)
```

- *aggregator.py:*
  - Inserito attributo *local_optimization_result*: inizializzato a *None*, è necessario qualora si volesse effettuare un aggregazione a partire da un'istanza di `LocalOptimizationResult`.
    Questo è stato necessario dal momento che, per la webapp, è stato richiesta una separazione tra l'ottimizzazione locale e l'aggregazione.
  - Inserito metodo `set_local_optimization_result(self, opt: LocalOptimizationResult)`, per inizializzare *local_optimization_result* a partire dall'istanza passata come argomento.
  - Inserito metodo *`aggregate_from_local_opt_result():`* effettua l'aggregazione a partire da un'istanza di `LocalOptimizationResult`, precedentemente inizializzata.

```
def aggregate_from_local_opt_result(self):
    try:
        for el in self.local_optimization_result.optimizations:
            self.result['minimized']['flexibility'] = [sum(x) for x in zip(self.result['minimized']['flexibility'],
                                                                           el['minimized']['flexibility'])]
            self.result['maximized']['flexibility'] = [sum(x) for x in zip(self.result['maximized']['flexibility'],
                                                                           el['maximized']['flexibility'])]
            self.result['minimized']['cost'] = [sum(x) for x in zip(self.result['minimized']['cost'],
                                                                    el['minimized']['cost'])]
            self.result['maximized']['cost'] = [sum(x) for x in zip(self.result['maximized']['cost'],
                                                                    el['maximized']['cost'])]

            return self.result
    except:
        raise Exception('LocalOptimizationResult not set.')
```

  - Inserito metodo *`resolve_pods_multiprocessing():`* Come richiesto, invocando questo metodo i pod inseriti nell'aggregatore vengono ottimizzato localmente in parallelo.

E’ stata utilizzata la libreria *multiprocess:* una volta deciso quanti core utilizzare, viene invocato il metodo *Pool.map(worker, self.pods)*.
La parallelizzazione è automatica: su ogni elemento dell’array di pods, viene eseguito ciò che è stato definito nella funzione *worker.* In questo caso si tratta di `Pod.resolve()`.

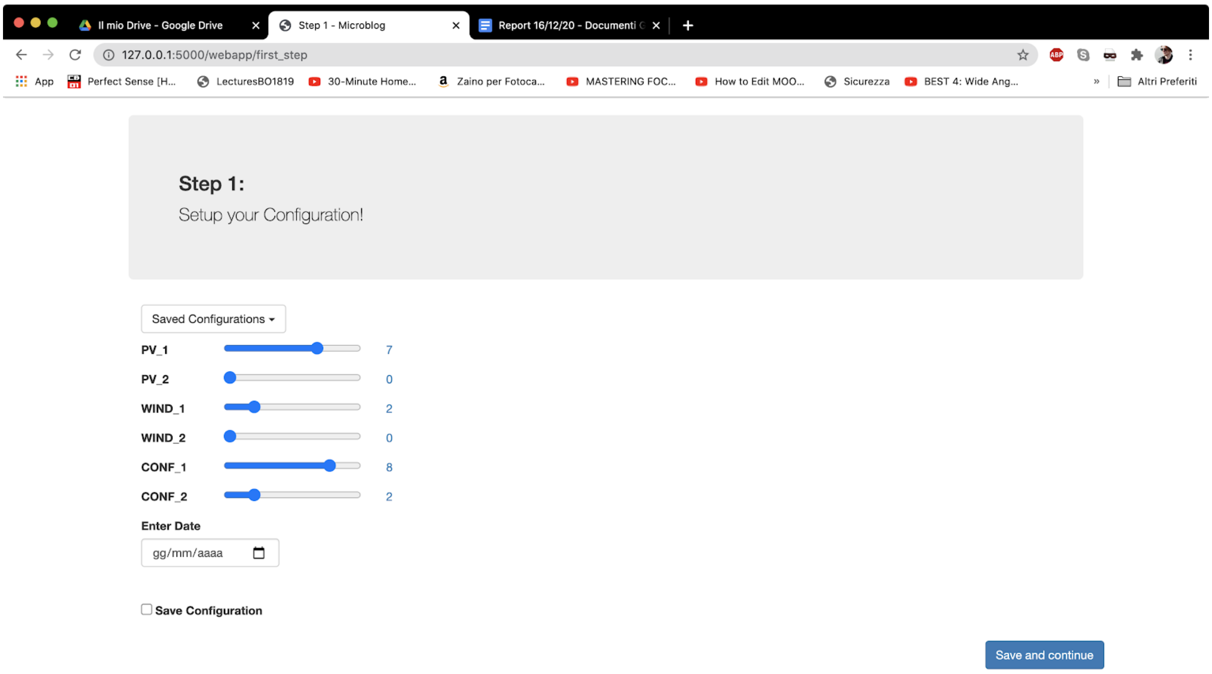
```
def worker(arg):
    obj = arg
    return obj.resolve()
```

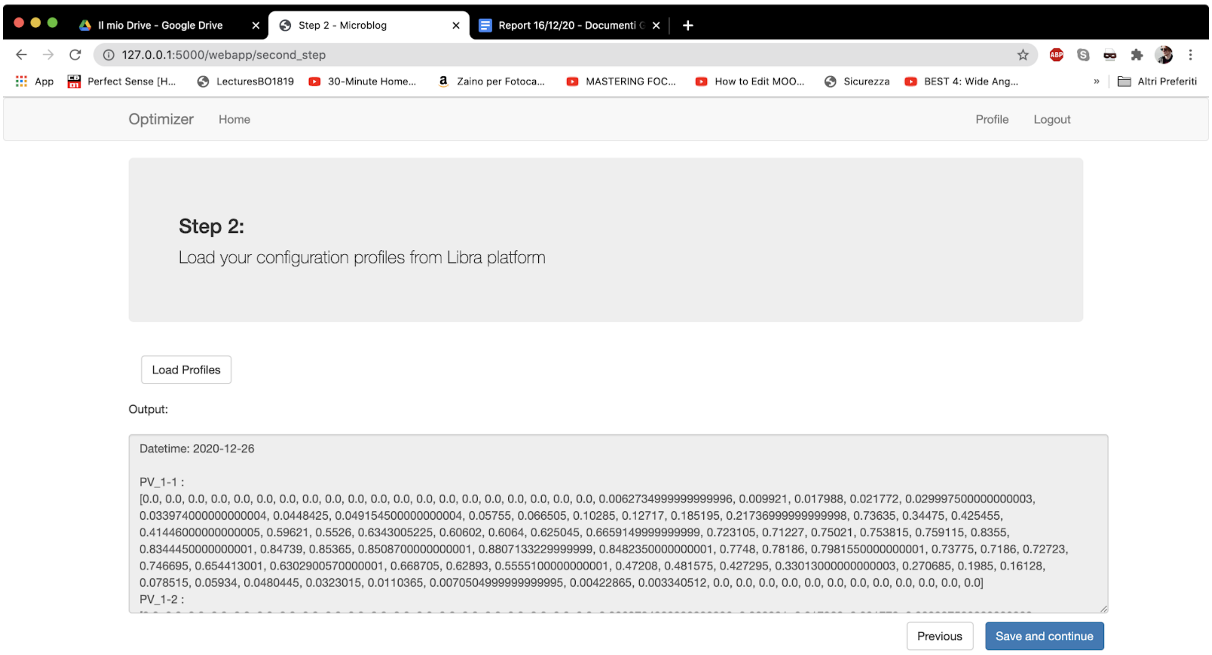
```
    def resolve_pods_multiprocessing(self):
        n_core = mp.cpu_count()  # set to the number of cores you want to use
        try:
            with mp.Pool(n_core) as pool:
                self.pods = pool.map(worker, self.pods)
        except TimeoutError:
            "We lacked patience and got a multiprocessing.TimeoutError"
        return self.pods
```

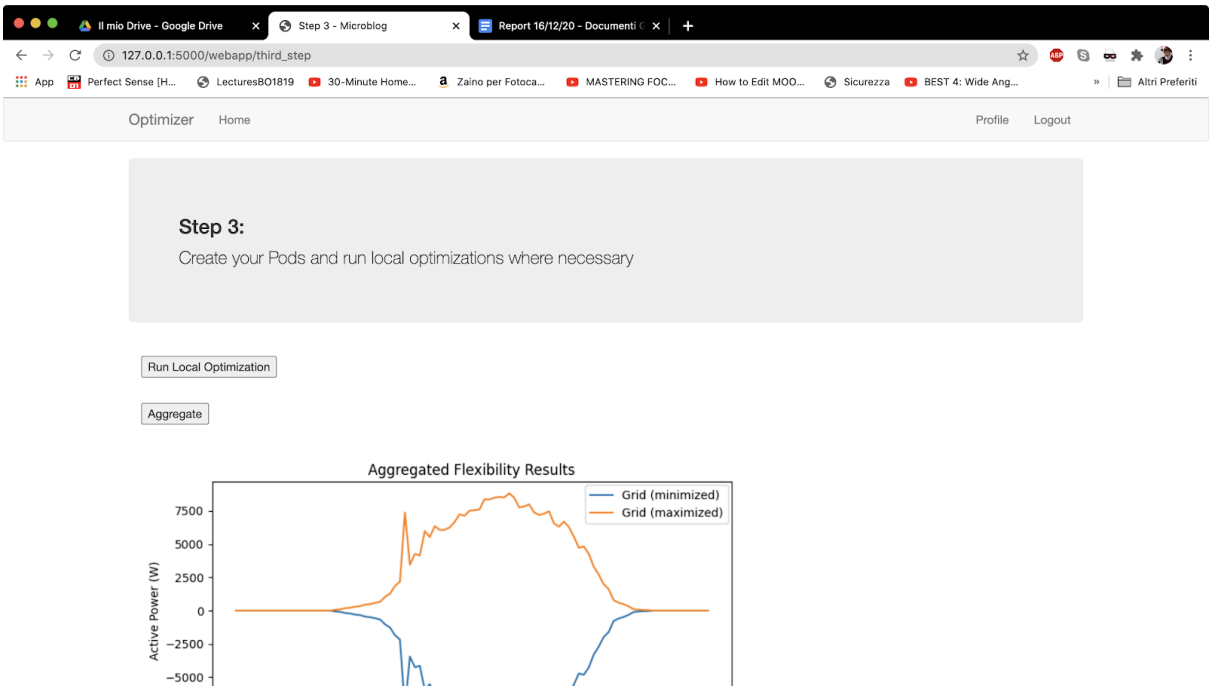
Modifiche alla Webapp:

- Suddivisione della View in tre Steps:
    - Step1: Configurazione parametri di input, quali tipo e numero di impianti da utilizzare nell'ottimizzazione, e data. Possibilità di salvare i parametri di configurazione su db utente (SQLAlchemy)
    - Step2: Vengono caricati i profili degli impianti richiesti da Libra (per ora vengono presi da un *dict* di prova) e mostrati nell’interfaccia.
    - Step3: Possibilità di eseguire l’ottimizzazione locale di ogni pod e, successivamente l’aggregazione (Il bottone per l’aggregazione viene abilitato soltanto una volta effettuate le varie ottimizzazioni locali).
      La rappresentazione json dell’oggetto *LocalOptimizationResponse* ritornato dall’ API `local_optimization` viene salvata in sessione, in modo da andare a recuperare il dato prima di effettuare la chiamata all’api `aggregate`.
      Poichè il limite della sessione client-side è 4Kb, e con un numero elevato di pod questo limite verrebbe superato, si è scelto di optare per una Flask server-side session utilizzando *Redis.*

      (Di seguito gli screenshots degli Steps)


Il mio Drive - Google Drive
Step 1 - Microblog
Report 16/12/20 - Documenti
127.0.0.1:5000/webapp/first_step
App
Perfect Sense [H...
LecturesBO1819
30-Minute Home...
Zaino per Fotoca...
MASTERING FOC...
How to Edit MOO...
Sicurezza
BEST 4: Wide Ang...
Altri Preferiti
Step 1:
Setup your Configuration!
Saved Configurations
PV_1 7
PV_2 0
WIND_1 2
WIND_2 0
CONF_1 8
CONF_2 2
Enter Date
gg/mm/aaaa
Save Configuration
Save and continue


Il mio Drive - Google Drive
Step 2 - Microblog
Report 16/12/20 - Documenti
127.0.0.1:5000/webapp/second_step
App
Perfect Sense [H...
LecturesBO1819
30-Minute Home...
Zaino per Fotoca...
MASTERING FOC...
How to Edit MOO...
Sicurezza
BEST 4: Wide Ang...
Altri Preferiti
Optimizer
Home
Profile
Logout
Step 2:
Load your configuration profiles from Libra platform
Load Profiles
Output:
Datetime: 2020-12-26
PV_1-1 :
[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.006273499999999996, 0.009921, 0.017988, 0.021772, 0.029997500000000003,
0.033974000000000004, 0.0448425, 0.049154500000000004, 0.05755, 0.066505, 0.10285, 0.12717, 0.185195, 0.21736999999999998, 0.73635, 0.34475, 0.425455,
0.41446000000000005, 0.59621, 0.5526, 0.6343005225, 0.60602, 0.6064, 0.625045, 0.6659149999999999, 0.723105, 0.71227, 0.75021, 0.753815, 0.759115, 0.8355,
0.8344450000000001, 0.84739, 0.85365, 0.8508700000000001, 0.8807133229999999, 0.8482350000000001, 0.7748, 0.78186, 0.7981550000000001, 0.73775, 0.7186, 0.72723,
0.746695, 0.654413001, 0.6302900570000001, 0.668705, 0.62893, 0.5555100000000001, 0.47208, 0.481575, 0.427295, 0.33013000000000003, 0.270685, 0.1985, 0.16128,
0.078515, 0.05934, 0.0480445, 0.0323015, 0.0110365, 0.007050499999999995, 0.00422865, 0.003340512, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
PV_1-2 :
Previous
Save and continue



Riassunto operazioni concluse nelle ultime settimane:

- Parallelizzazione delle ottimizzazioni locali
- Separazione della fase di Ottimizzazione Locale, da quella di Aggregazione.
- Refactoring della View della Webapp in modo da rendere il tutto più funzionale

Da completare:

- Rivedere la parte di lettura dei profili da Libra
- Modellare la fase di aggregazione come una seconda ottimizzazione
- Aggiungere alcune piccolezze javascript client-side