A Case Study for Moving Targets

A Case Study for Moving Targets

Let's consider a more practical use case for MT

We will still tackle a synthetic problem, but one closer to practice

- In particular, given a classification problem
- We will require to have roughly balance class predictions

$$\left|\sum_{i=1}^{m} z_{ij} - \frac{m}{n_c}\right| \leq \beta \frac{m}{n_c}, \quad \forall j \in 1..n_c$$

- Where $z_{ij} = 1$ iff the classifier predicts class j for example i
- I.e. the result of an argmax applied to the output of a probabilistic classifier

...Basically, this the example from the previous section

The Dataset

We will use the "wine quality" dataset from UCI

```
In [11]: data = util.load_classification_dataset(fname, onehot_inputs=['quality'])
         display(data.head())
```

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality_3	quality_4	quality_5	qual
0	7.0	0.27	0.36	20.7	0.045	45.0	170.0	1.0010	3.00	0.45	8.8	False	False	False	True
1	6.3	0.30	0.34	1.6	0.049	14.0	132.0	0.9940	3.30	0.49	9.5	False	False	False	True
2	8.1	0.28	0.40	6.9	0.050	30.0	97.0	0.9951	3.26	0.44	10.1	False	False	False	True
3	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40	9.9	False	False	False	True
4	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40	9.9	False	False	False	True

- We will learn a model to predict wine quality
- There are 7 possible classes, represented via a one-hot encoding
- An ordinal encoding would be better, but our choice makes for a better example

The Dataset

We perform pre-processing as usual

In [12]:
```
dtout = [c for c in data.columns if c.startswith('quality_')]
dtin = [c for c in data.columns if c not in dtout]
trl, tsl, scalers = util.split_datasets([data], fraction=0.7, seed=42, standardize=dtin)
tr, ts, scaler = trl[0], tsl[0], scalers[0]
tr.describe()
```

Out[12]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density
count	1.469000e+03	1.469000e+03	1.469000e+03	1.469000e+03	1.469000e+03	1.469000e+03	1.469000e+03	1.469000e+03
mean	5.235960e-16	1.475259e-16	-1.934766e-16	1.644551e-16	-1.015752e-16	-1.813843e-16	1.463167e-16	-8.416231e-15
std	1.000341e+00	1.000341e+00	1.000341e+00	1.000341e+00	1.000341e+00	1.000341e+00	1.000341e+00	1.000341e+00
min	-3.513954e+00	-1.981296e+00	-2.750607e+00	-1.170902e+00	-1.436585e+00	-1.916670e+00	-2.969922e+00	-2.338980e+00
25%	-6.396233e-01	-6.660564e-01	-5.421591e-01	-9.256225e-01	-4.519909e-01	-6.550002e-01	-7.262526e-01	-7.862395e-01
50%	-4.080447e-02	-1.601951e-01	-1.331873e-01	-2.306630e-01	-1.387109e-01	-8.151391e-02	-1.309932e-01	-5.839245e-02
75%	5.580144e-01	4.468384e-01	4.393732e-01	7.197965e-01	1.745692e-01	5.493210e-01	6.474229e-01	7.387734e-01
max	8.821714e+00	4.898418e+00	5.347035e+00	4.031074e+00	1.006527e+01	1.454239e+01	6.897646e+00	3.112941e+00

Dataset Balance

We can use the (avg. of) our constraint metric to assess the dataset balance:

$$\frac{1}{n_c} \sum_{j=1}^{n_c} \left| \sum_{i=1}^{m} z_{ij} - \frac{m}{n_c} \right|, \qquad \forall j \in 1..n_c$$

- Where $\hat{z}$ are the class columns (one-hot encoding)

In [19]:
```
bal_thr = 0.3
tr_true = np.argmax(tr[dtout].values, axis=1)
ts_true = np.argmax(ts[dtout].values, axis=1)
tr_bal_src = util.avg_bal_deviation(tr_true, bal_thr, nclasses=len(dtout))
ts_bal_src = util.avg_bal_deviation(ts_true, bal_thr, nclasses=len(dtout))
print(f'Original avg deviation: {tr_bal_src*100:.2f}% (training), {ts_bal_src*100:.2f}% (tes
```

```
Original avg deviation: 101.42% (training), 98.71% (test)
```

- Our goal will be to push the balance deviation down to 30%
- I.e. we will assume $\beta = 0.3$ in the constraint

The Learner

Our "learner" will be a multilayer perceptron

The code can be found as usual in the `util` module

```
class MLPLearner(object):
    def __init__(self, hidden, epochs=20, batch_size=32,
                 epochs_fine_tuning=None, verbose=0): ...

    def fit(self, X, y): ...

    def predict_proba(self, X): ...

    def predict(self, X): ...
```

- We are using a standard scikit-learn API
- ...With the ability to use different #epochs for the first and subsequent training
- ...Which will prove useful later

The Learner

Let's start by checking how regular training fares

In [20]:
```
learner = XGBClassifier(n_estimators=20, max_depth=2, learning_rate=0.5, objective='binary:
learner.fit(tr[dtin].values, tr[dtout].values)
tr_pred_prob = learner.predict_proba(tr[dtin])
ts_pred_prob = learner.predict_proba(ts[dtin])
tr_acc, tr_bal = util.mt_balance_stats(tr[dtout].values, tr_pred_prob, bal_thr)
ts_acc, ts_bal = util.mt_balance_stats(ts[dtout].values, ts_pred_prob, bal_thr)
print(f'Accuracy: {tr_acc:.2f} (training), {ts_acc:.2f} (test)')
print(f'Classifier avg deviation: {tr_bal*100:.0f}% (training), {ts_bal*100:.0f}% (test)')
print(f'Balance violation threshold: {bal_thr*100:.0f}%')
```

```
Accuracy: 0.67 (training), 0.55 (test)
Classifier avg deviation: 119% (training), 124% (test)
Balance violation threshold: 30%
```

- The tets accuracy is slightly above 50%
- ...But the balance violation is far larger than our threshold

The Master

The master step is implemented via an Or-tools model

```python
def mt_balance_master(y_true, y_pred, bal_thr, rho=1, ...):
    # Build a model
    slv = pywraplp.Solver.CreateSolver('SAT')
    ...
    # Solve
    status = slv.Solve()
    ...
    # Return the solution and stats
    return sol, stats
```

- `y_true` corresponds to $\hat{y}$ and `y_pred` to the current prediction vector
- The balance threshold `bal_thr` is a fractional value
- A mode parameter allows one to adjust a bit the problem behavior

The Master

The master step is implemented via an Or-tools model

```python
def mt_balance_master(y_true, y_pred, bal_thr, rho=1, ...):
    ...
    # Build target variables
    z = {(i,j) : slv.IntVar(0, 1, f'z[{i},{j}]') for i in range(ns) for j in range(nc)}
    # Unique class constraints
    for i in range(ns):
        slv.Add(sum(z[i, j] for j in range(nc)) == 1)
    ...
```

- We are using integer variables for the targets
- ...Which is consistent with the semantic of our constraint

The Master

The master step is implemented via an Or-tools model

```python
def mt_balance_master(y_true, y_pred, bal_thr, rho=1, ...):
    ...
    # Add the balance constraint
    ref = ns / nc
    for j in range(nc):
        slv.Add(sum(z[i, j] for i in range(ns)) <= ref + ref * bal_thr)
        slv.Add(sum(z[i, j] for i in range(ns)) >= ref - ref * bal_thr)
    ...
```

- The balance constraint is implemented via two inequalities
- Class counts are easy to compute by relying on integer one-hot targets

The Master

The master step is implemented via an Or-tools model

```
def mt_balance_master(y_true, y_pred, bal_thr, rho=1, ...):
    ...
    # Build the loss w.r.t. the original target
    loss_t = 0
    for i in range(ns):
        for j in range(nc):
            loss_t += y_true[i, j] * (1 - z[i, j])
    ...
```

- As a loss L for the master, the categorical crossed-entropy
- ...With logarithms capped and scaled by $\log \varepsilon$
- The formula can be simplified since the original targets are 0-1

The Master

The master step is implemented via an Or-tools model

```
def mt_balance_master(y_true, y_pred, bal_thr, rho=1, ...):
    ...
    # Build the quadratic part of the objective
    loss_p = 0
    for i in range(ns):
        for j in range(nc):
            loss_p += z[i, j] * scaled_pred_cnl[i, j]
    # Define the cost function
    slv.Minimize(loss_t + 1/rho * loss_p)
    ...
```

- We use the scaled cross-entropy for the loss w.r.t. the predictions, too
- ...But no simplification is possible, since the predictions are continuous

Loss-driven Projection

A simpler approach to inject constraints in the ML model...

...Starts by directly "projecting" the ground truth $\hat{y}$ in feasible space

- This can be obtained by setting $\rho = \infty$
- The projection can be done using the loss itself as a distance:

$$\text{argmin}_z \left\{ L(z, \hat{y}) \mid z \in C \right\}$$

By doing this, we can obtain the best possible feasible target vector

In [21]:
```
%%time
zp, stats = util.mt_balance_master(tr[dtout].values, tr[dtout].values, bal_thr, time_limit=
tmp_acc, tmp_bal = util.mt_balance_stats(tr[dtout].values, zp, bal_thr)
print(f'Accuracy: {tmp_acc:.2f}, Balance deviation: {tmp_bal*100:.2f}%, Optimal solution: {
```

```
Accuracy: 0.62, Balance deviation: 25.38%, Optimal solution: True
CPU times: user 8.88 s, sys: 77.6 ms, total: 8.96 s
Wall time: 865 ms
```

Loss-driven Projection

Then, the simple approach consists training against this "ideal" vector

The method is implemented in `util` as part of the MT code:

```
In [22]: from xgboost import XGBClassifier
         learner_prj = XGBClassifier(n_estimators=50, max_depth=2, learning_rate=0.5, objective='bina
         util.mt_balance(tr[dtin].values, tr[dtout].values, learner_prj, bal_thr, rho=np.inf, master_
         tr_pred_prob = learner_prj.predict_proba(tr[dtin])
         ts_pred_prob = learner_prj.predict_proba(ts[dtin])
         tr_acc, tr_bal = util.mt_balance_stats(tr[dtout].values, tr_pred_prob, bal_thr)
         ts_acc, ts_bal = util.mt_balance_stats(ts[dtout].values, ts_pred_prob, bal_thr)
         print(f'Accuracy: {tr_acc:.2f} (training), {ts_acc:.2f} (test)')
         print(f'Classifier balance violation: {tr_bal*100:.0f}% (training), {ts_bal*100:.0f}% (test
```

```
Accuracy: 0.57 (training), 0.40 (test)
Classifier balance violation: 58% (training), 66% (test)
```

- The constraint violation is still too high!

Moving Targets

Let's now test the actual MT method

We use fewer training epochs after the first `fit` call to speed up the process

```
In [23]: from xgboost import XGBClassifier
         rho, max_iter = 0.1, 10
         learner_mt = XGBClassifier(n_estimators=50, max_depth=2, learning_rate=0.5, objective='bina
         util.mt_balance(tr[dtin].values, tr[dtout].values, learner_mt, bal_thr, rho=rho, max_iter=ma

(#1) l-acc: 0.52, l-bal: 0.37, l-time: 0.28s, m-acc: 0.59, l-m dist: 2.08, m-time: 0.49s
(#2) l-acc: 0.52, l-bal: 0.32, l-time: 0.28s, m-acc: 0.58, l-m dist: 0.62, m-time: 0.46s
(#3) l-acc: 0.53, l-bal: 0.29, l-time: 0.27s, m-acc: 0.57, l-m dist: 0.54, m-time: 0.43s
(#4) l-acc: 0.52, l-bal: 0.29, l-time: 0.28s, m-acc: 0.57, l-m dist: 0.50, m-time: 0.43s
(#5) l-acc: 0.53, l-bal: 0.29, l-time: 0.27s, m-acc: 0.57, l-m dist: 0.47, m-time: 0.46s
(#6) l-acc: 0.52, l-bal: 0.29, l-time: 0.27s, m-acc: 0.56, l-m dist: 0.45, m-time: 0.49s
(#7) l-acc: 0.52, l-bal: 0.30, l-time: 0.27s, m-acc: 0.56, l-m dist: 0.43, m-time: 0.45s
(#8) l-acc: 0.52, l-bal: 0.29, l-time: 0.27s, m-acc: 0.56, l-m dist: 0.42, m-time: 0.44s
(#9) l-acc: 0.53, l-bal: 0.30, l-time: 0.27s, m-acc: 0.56, l-m dist: 0.41, m-time: 0.44s
(#10) l-acc: 0.53, l-bal: 0.29, l-time: 0.27s, m-acc: 0.56, l-m dist: 0.41, m-time: 0.45s
```

- Constraint satisfaction improves across iterations
- The learner/master accuracy tends to decrease/increase

Moving Targets

Let's check generalization over the test set

```
In [24]: tr_pred_prob = learner_mt.predict_proba(tr[dtin])
         ts_pred_prob = learner_mt.predict_proba(ts[dtin])
         tr_acc, tr_bal = util.mt_balance_stats(tr[dtout].values, tr_pred_prob, bal_thr)
         ts_acc, ts_bal = util.mt_balance_stats(ts[dtout].values, ts_pred_prob, bal_thr)
         print(f'Accuracy: {tr_acc:.2f} (training), {ts_acc:.2f} (test)')
         print(f'Classifier balance deviation: {tr_bal*100:.0f}% (training), {ts_bal*100:.0f}% (test
```

```
Accuracy: 0.53 (training), 0.34 (test)
Classifier balance deviation: 29% (training), 28% (test)
```

We do have some overfitting in terms of accuracy

- This is due to the fact that we have very restrictive constraints
- ...And they directly oppose information in the data

Constraint satisfaction generalizes without issues

- ...And this is a big deal!