pmc

November 8, 2025

[4]:

```
import pm4py
import pandas as pd
import numpy as np
from pathlib import Path

# --- Version info (optional print) ---
print("pm4py version:", getattr(pm4py, "__version__", "unknown"))

# ----------------------------
# Discovery & conversions
# ----------------------------
from pm4py.algo.discovery.inductive import algorithm as inductive_miner
from pm4py.algo.discovery.heuristics import algorithm as heuristics_miner
from pm4py.objects.conversion.bpmn import converter as bpmn_converter
from pm4py.objects.conversion.heuristics_net import converter as hn_converter
from pm4py.objects.conversion.bpmn import converter as bpmn_to_petri_converter

def export_bpmn(bpmn_graph, path):
    path = str(path)
    try:
        pm4py.write_bpmn(bpmn_graph, path)
    except Exception:
        try:
            from pm4py.objects.bpmn.exporter import exporter as bpmn_exporter
            bpmn_exporter.apply(bpmn_graph, path)
        except Exception as e:
            raise RuntimeError(f"Could not export BPMN: {e}")

def bpmn_to_petri(bpmn_graph):
    net, im, fm = bpmn_to_petri_converter.apply(bpmn_graph)
    return net, im, fm

# ----------------------------
# Conformance (robust imports)
# ----------------------------
# Token-based replay (fitness)
try:
    # PM4Py  2.x (stable path)
    from pm4py.algo.conformance.tokenreplay import algorithm as token_replay
except Exception:
    # Older fallback (rare)
    from pm4py.algo.conformance import tokenreplay as token_replay

# Alignments (optional, slower) - only used if available
try:
    from pm4py.algo.conformance.alignments.petri_net import algorithm as␣
 ↪alignments_alg
    _HAS_ALIGNMENTS = True
except Exception:
    _HAS_ALIGNMENTS = False

# Precision (ETConformance variant)
precision_apply = None
# Newer path
try:
    from pm4py.algo.conformance.precision.variants import etconformance_token␣
 ↪as _prec_et
    precision_apply = _prec_et.apply
except Exception:
    pass
# Alternative path in some versions
if precision_apply is None:
    try:
        from pm4py.algo.conformance.precision import algorithm as _prec_algo
        # Some versions expose .apply directly on algorithm
        precision_apply = _prec_algo.apply
    except Exception:
        precision_apply = None  # we'll handle below

# Generalization (optional; not in all builds)
def _try_generalization(event_log, net, im, fm):
    try:
        from pm4py.algo.evaluation.generalization import evaluator as␣
 ↪gen_eval_new
        val = gen_eval_new.evaluate(event_log, net, im, fm)
        if isinstance(val, dict) and "generalization" in val:
            return float(val["generalization"])
        return float(val)
    except Exception:
        try:
            from pm4py.evaluation.generalization import evaluator as␣
 ↪gen_eval_old
            val = gen_eval_old.evaluate(event_log, net, im, fm)
            if isinstance(val, dict) and "generalization" in val:
```

```python
                return float(val["generalization"])
            return float(val)
        except Exception:
            return np.nan

# Built-in simplicity (optional; not in all builds)
def _try_builtin_simplicity(net):
    try:
        from pm4py.algo.evaluation.simplicity import evaluator as simp_eval_new
        val = simp_eval_new.apply(net)
        if isinstance(val, dict) and "simplicity" in val:
            return float(val["simplicity"])
        return float(val)
    except Exception:
        try:
            from pm4py.evaluation.simplicity import evaluator as simp_eval_old
            val = simp_eval_old.apply(net)
            if isinstance(val, dict) and "simplicity" in val:
                return float(val["simplicity"])
            return float(val)
        except Exception:
            return np.nan

# -----------------------------
# Metric computation (version-agnostic)
# -----------------------------
def compute_all_metrics(event_log, net, im, fm, prefer_alignments=False):
    """
    Compute quality metrics using only stable, widely available PM4Py APIs.
    - Fitness: token-based replay average trace fitness (alignments optional if available).
    - Precision: ETConformance token variant when available.
    - Generalization & built-in simplicity: best-effort (may be NaN if not present in your build).
    - Two custom simplicity metrics always computed.
    """
    # --- Fitness ---
    fitness = np.nan
    if prefer_alignments and _HAS_ALIGNMENTS:
        try:
            # Alignments return list of dicts; compute avg fitness from diagnostics if available
            al = alignments_alg.apply(event_log, net, im, fm)
            # Some versions return dicts with 'fitness' per trace; otherwise compute via costs (fallback to token)
            per_trace = []
            for x in al:
```

```
            if isinstance(x, dict) and "fitness" in x:
                per_trace.append(x["fitness"])
        if per_trace:
            fitness = float(np.mean(per_trace))
    except Exception:
        fitness = np.nan

if np.isnan(fitness):
    # Token Replay (robust & fast)
    try:
        diag = token_replay.apply(event_log, net, im, fm)
        # diag is a list per trace; each has 'trace_fitness' (0..1)
        per_trace = []
        for x in diag:
            if isinstance(x, dict) and "trace_fitness" in x:
                per_trace.append(x["trace_fitness"])
        if per_trace:
            fitness = float(np.mean(per_trace))
    except Exception:
        fitness = np.nan

# --- Precision ---
if precision_apply is not None:
    try:
        precision = float(precision_apply(event_log, net, im, fm))
    except Exception:
        precision = np.nan
else:
    precision = np.nan

# --- Generalization (best effort) ---
generalization = _try_generalization(event_log, net, im, fm)

# --- Built-in simplicity (best effort) ---
simplicity_builtin = _try_builtin_simplicity(net)

# --- Custom simplicity metrics (always available) ---
num_places = len(net.places)
num_transitions = len(net.transitions)
num_arcs = len(net.arcs)
size = num_places + num_transitions + num_arcs

# M1: Size-based simplicity
simplicity_size = 1.0 / (1.0 + np.log1p(size))

# M2: Connectivity simplicity (inverse avg degree)
nodes = (num_places + num_transitions)
```

```
    connectivity_simplicity = 1.0 / (1.0 + (2.0 * num_arcs / nodes)) if nodes >␣
↪0 else np.nan

    return {
        "fitness": fitness,
        "precision": precision,
        "generalization": generalization,
        "simplicity_builtin": simplicity_builtin,
        "simplicity_size": float(simplicity_size),
        "simplicity_connectivity": float(connectivity_simplicity),
        "places": int(num_places),
        "transitions": int(num_transitions),
        "arcs": int(num_arcs),
        "size": int(size),
    }
```

```
pm4py version: 2.7.18
```

[7]:

```
from pm4py.objects.log.util import sorting
from pm4py.objects.conversion.log import converter as log_converter

LOG_PATH = "bpi-chall.xes"

# Read XES file
elog = pm4py.read_xes(LOG_PATH)

# Convert to EventLog if needed
if isinstance(elog, pd.DataFrame):
    elog = log_converter.apply(elog, variant=log_converter.Variants.
↪TO_EVENT_LOG)

# Sort events by timestamp
elog = sorting.sort_timestamp(elog, timestamp_key="time:timestamp")

print(type(elog))
print(f"Number of cases: {len(elog)}")
```

```
parsing log, completed traces :: 100%|      | 31509/31509 [00:33<00:00,
933.69it/s]
```

```
<class 'pm4py.objects.log.obj.EventLog'>
Number of cases: 31509
```

[22]:

```
from pathlib import Path
import pm4py
import pandas as pd
import numpy as np
```

```
# Discovery
from pm4py.algo.discovery.inductive import algorithm as inductive_miner
from pm4py.algo.discovery.heuristics import algorithm as heuristics_miner
from pm4py.objects.conversion.bpmn import converter as bpmn_converter
from pm4py.objects.conversion.heuristics_net import converter as hn_converter

# --- helpers to resolve parameter keys across pm4py versions ---

def im_param_dict(noise_thr: float):
    """
    Build a parameter dict for Inductive Miner that works across pm4py versions.
    Tries the enum key first, then module-level Parameters, then plain string␣
 ↪key.
    """
    # Try enum on variant.value
    try:
        key = inductive_miner.Variants.IMf.value.Parameters.NOISE_THRESHOLD
        return {key: noise_thr}
    except Exception:
        pass
    # Try module-level Parameters
    try:
        key = inductive_miner.Parameters.NOISE_THRESHOLD
        return {key: noise_thr}
    except Exception:
        pass
    # Fallback to plain string
    return {"noise_threshold": noise_thr}

def hm_param_dict(dep_thr: float):
    """
    Build a parameter dict for Heuristics Miner across versions.
    Keys have appeared as DEPENDENCY_THRESH / DEPENDENCY_THRESHOLD /␣
 ↪dependency_threshold.
    """
    # Try enum on variant.value
    try:
        key = heuristics_miner.Variants.CLASSIC.value.Parameters.
 ↪DEPENDENCY_THRESH
        return {key: dep_thr}
    except Exception:
        pass
    try:
        key = heuristics_miner.Variants.CLASSIC.value.Parameters.
 ↪DEPENDENCY_THRESHOLD
        return {key: dep_thr}
    except Exception:
        pass
    # Try module-level Parameters
    for name in ["DEPENDENCY_THRESH", "DEPENDENCY_THRESHOLD"]:
        try:
            key = getattr(heuristics_miner.Parameters, name)
            return {key: dep_thr}
        except Exception:
            pass
    # Fallback to plain strings commonly recognized
    for s in ["dependency_threshold", "dependency_thresh"]:
        return {s: dep_thr}

def export_bpmn(bpmn_graph, path):
    path = str(path)
    try:
        pm4py.write_bpmn(bpmn_graph, path)
    except Exception:
        try:
            from pm4py.objects.bpmn.exporter import exporter as bpmn_exporter
            bpmn_exporter.apply(bpmn_graph, path)
        except Exception as e:
            raise RuntimeError(f"Could not export BPMN: {e}")

# --- your Step 2 with robust parameter handling ---

outdir = Path("models")
outdir.mkdir(exist_ok=True)

candidates = []
```

[23]:

```
# A) Inductive Miner - try multiple noise thresholds
from pm4py.objects.conversion.wf_net import converter as wf_net_converter
from pm4py.objects.process_tree.obj import ProcessTree
from pm4py.objects.conversion.process_tree import converter as pt_converter
```

[26]:

```
from pm4py.objects.process_tree.obj import ProcessTree
from pm4py.objects.conversion.process_tree import converter as pt_converter
from pm4py.objects.conversion.wf_net import converter as wf_net_converter

def to_bpmn(res):
    """
    Convert the result of a discovery algorithm to a BPMN graph, regardless of
    whether the result is a ProcessTree or (net, im, fm).
    """
    if isinstance(res, ProcessTree):
        # Convert ProcessTree -> BPMN directly
        return pt_converter.apply(res, variant=pt_converter.Variants.TO_BPMN)
    else:
        net, im, fm = res
        # Convert WF-net (Petri) -> BPMN explicitly
        return wf_net_converter.apply(
            net, im, fm,
            variant=wf_net_converter.Variants.TO_BPMN
        )

# --- A) Inductive Miner
for noise_thr in [0.0, 0.2, 0.4]:
    params = im_param_dict(noise_thr)
    res = inductive_miner.apply(
        elog,
        variant=inductive_miner.Variants.IMf,
        parameters=params
    )

    # If you still need (net, im, fm) for metrics, keep producing them too:
    if isinstance(res, ProcessTree):
        net, im, fm = pt_converter.apply(res, variant=pt_converter.Variants.
 ↪TO_PETRI_NET)
    else:
        net, im, fm = res

    bpmn_graph = to_bpmn(res)
    bpmn_path = outdir / f"candidate_IM_noise{str(noise_thr).replace('.','_')}.
 ↪bpmn"
    export_bpmn(bpmn_graph, bpmn_path)

    metrics = compute_all_metrics(elog, net, im, fm)
    metrics.update({"algo": "InductiveMiner",
                    "params": {"noise_threshold": noise_thr},
                    "bpmn_path": str(bpmn_path)})
    candidates.append(metrics)

# --- B) Heuristics Miner
for dep_thr in [0.6, 0.8]:
    params = hm_param_dict(dep_thr)
    try:
        hnet = heuristics_miner.apply_heu(elog, parameters=params)
    except Exception:
        hnet = heuristics_miner.apply(elog, variant=heuristics_miner.Variants.
 ↪CLASSIC, parameters=params)

    net, im, fm = hn_converter.apply(hnet)

    # Explicitly request BPMN here
    bpmn_graph = wf_net_converter.apply(
        net, im, fm,
        variant=wf_net_converter.Variants.TO_BPMN
    )
    bpmn_path = outdir / f"candidate_HM_dep{str(dep_thr).replace('.','_')}.bpmn"
    export_bpmn(bpmn_graph, bpmn_path)

    metrics = compute_all_metrics(elog, net, im, fm)
    metrics.update({"algo": "HeuristicsMiner",
                    "params": {"dependency_threshold": dep_thr},
                    "bpmn_path": str(bpmn_path)})
    candidates.append(metrics)
```

```
replaying log with TBR, completed traces :: 100%|      | 15930/15930
[01:07<00:00, 236.04it/s]
replaying log with TBR, completed traces :: 100%|      | 15930/15930
[01:12<00:00, 219.11it/s]
replaying log with TBR, completed traces :: 100%|      | 15930/15930
[01:04<00:00, 245.82it/s]
replaying log with TBR, completed traces :: 100%|      | 15930/15930
[01:01<00:00, 258.44it/s]
replaying log with TBR, completed traces :: 100%|      | 15930/15930
[01:00<00:00, 263.54it/s]
```

[29]:

```
results = pd.DataFrame(candidates)
```

[30]:

```
# Score: prioritize fitness, then precision, then simplicity (avg of the three␣
 ↪simplicity measures)
simplicity_cols = ["simplicity_builtin", "simplicity_size",␣
 ↪"simplicity_connectivity"]
results["simplicity_avg"] = results[simplicity_cols].apply(lambda row: np.
 ↪nanmean(row), axis=1)

# Hard constraint: fitness >= 0.80 (approx 80% of cases replayable)
feasible = results[results["fitness"] >= 0.80].copy()

if feasible.empty:
    print("No candidate reached fitness  0.80. Selecting the best fitness␣
 ↪overall and consider parameter retuning.")
    feasible = results.copy()

# Weighted score (tweak if you like)
results["score"] = (
    0.55 * results["fitness"].fillna(0)
    + 0.25 * results["precision"].fillna(0)
    + 0.20 * results["simplicity_avg"].fillna(0)
)
```

```python
feasible["score"] = (
    0.55 * feasible["fitness"].fillna(0)
    + 0.25 * feasible["precision"].fillna(0)
    + 0.20 * feasible["simplicity_avg"].fillna(0)
)

display_cols = [
    "algo", "params", "fitness", "precision", "generalization",
    "simplicity_builtin", "simplicity_size", "simplicity_connectivity",
    "size", "places", "transitions", "arcs", "score", "bpmn_path"
]
print("=== Candidates (sorted by score) ===")
display(results.sort_values("score", ascending=False)[display_cols])

final_row = feasible.sort_values("score", ascending=False).iloc[0]
final_bpmn_path = final_row["bpmn_path"]
final_row
```

```
=== Candidates (sorted by score) ===

              algo                         params   fitness  precision  \
0   InductiveMiner       {'noise_threshold': 0.0}  1.000000        NaN
1   InductiveMiner       {'noise_threshold': 0.2}  0.978382        NaN
3  HeuristicsMiner  {'dependency_threshold': 0.6}  0.948147        NaN
4  HeuristicsMiner  {'dependency_threshold': 0.8}  0.948036        NaN
2   InductiveMiner       {'noise_threshold': 0.4}  0.930738        NaN

   generalization  simplicity_builtin  simplicity_size  \
0             NaN                 NaN         0.147276
1             NaN                 NaN         0.150559
3             NaN                 NaN         0.149022
4             NaN                 NaN         0.149022
2             NaN                 NaN         0.152876

   simplicity_connectivity  size  places  transitions  arcs     score  \
0                 0.278431   326      55           87   184  0.592571
1                 0.280182   281      48           75   158  0.581184
3                 0.254167   301      42           80   179  0.561800
4                 0.254167   301      42           80   179  0.561739
2                 0.282828   254      47           65   142  0.555476

                           bpmn_path
0  models/candidate_IM_noise0_0.bpmn
1  models/candidate_IM_noise0_2.bpmn
3    models/candidate_HM_dep0_6.bpmn
4    models/candidate_HM_dep0_8.bpmn
2  models/candidate_IM_noise0_4.bpmn
```

```
[30]: fitness                                                  1.0
      precision                                                NaN
      generalization                                           NaN
      simplicity_builtin                                       NaN
      simplicity_size                                     0.147276
      simplicity_connectivity                             0.278431
      places                                                    55
      transitions                                               87
      arcs                                                     184
      size                                                     326
      algo                                          InductiveMiner
      params                              {'noise_threshold': 0.0}
      bpmn_path                  models/candidate_IM_noise0_0.bpmn
      simplicity_avg                                      0.212854
      score                                               0.592571
      Name: 0, dtype: object
```

```
[ ]:
```