Predict-then-Optimize: a tour of the state-of-the-art using PyEPo

Elias B. Khalil
Department of Mechanical & Industrial Engineering
SCALE AI Research Chair in Data-Driven Algorithms for Modern Supply Chains

For complete references to related work, please check our arXiv manuscript linked in the GitHub.
Optimization with a linear objective

\[
\min \left\{ c^T w : w \in \mathcal{S} \right\}
\]

- **Decision variables**
- **Linear cost function**
- **Feasible set**

**Solution:** use appropriate algorithm depending on type of feasible set (MILP, MIQCP, CP, custom algorithms...)
The $n$ cost vectors could be completely unrelated.
\[
\min \{ c_1^T w : w \in \mathcal{S} \} \\
\min \{ c_2^T w : w \in \mathcal{S} \} \quad \text{Same Decision variables}
\]
\[
\min \{ c_n^T w : w \in \mathcal{S} \} \quad \text{Same Feasible set}
\]

Instance feature vector (Observed)

\[
x_1 \in \mathbb{R}^p
\]

\[
x_2
\]

\[
x_n
\]
Function $g$ is parametrized by a vector $\theta$. 

$$c_i = g(x_i; \theta) \in \mathbb{R}^d$$

$$\min \{ c_1^T w : w \in \mathcal{S} \}$$

$$\min \{ c_2^T w : w \in \mathcal{S} \}$$

$$\min \{ c_n^T w : w \in \mathcal{S} \}$$

Instance feature vector (Observed)

$$x_1 \in \mathbb{R}^p$$

$$x_2$$

$$x_n$$
If you know $\theta$ then you can evaluate $g(x_i; \theta)$ to obtain $c_i$ and optimize with your favorite method.

$$c_i = g(x_i; \theta) \in \mathbb{R}^d$$

$$\begin{align*}
\min_{w} \{ c_1^T w : w \in \mathcal{S} \} \\
\min_{w} \{ c_2^T w : w \in \mathcal{S} \} \\
\min_{w} \{ c_n^T w : w \in \mathcal{S} \}
\end{align*}$$

Instance feature vector (Observed)

$$x_1 \in \mathbb{R}^p$$

$$x_2$$

$$x_n$$
If you know $\theta$ then you can evaluate $g(x_i; \theta)$ to obtain $c_i$ and optimize with your favorite method.

$$\hat{c}_i = g(x_i; \theta) \in \mathbb{R}^d$$

$\min \{ \hat{c}_1^Tw : w \in \mathcal{S} \}$

$\min \{ \hat{c}_2^Tw : w \in \mathcal{S} \}$

$\min \{ \hat{c}_n^Tw : w \in \mathcal{S} \}$

What if you don’t know $\theta$ and you only observe $x_i$?

Instance feature vector (Observed)

$$x_1 \in \mathbb{R}^p$$

$$x_2$$

$$x_3$$

$$x_n$$

If you know $\theta$ then you can evaluate $g(x_i; \theta)$ to obtain $c_i$ and optimize with your favorite method.

$\hat{c}_i = g(x_i; \theta) \in \mathbb{R}^d$

$\min \{ \hat{c}_1^Tw : w \in \mathcal{S} \}$

$\min \{ \hat{c}_2^Tw : w \in \mathcal{S} \}$

$\min \{ \hat{c}_n^Tw : w \in \mathcal{S} \}$

What if you don’t know $\theta$ and you only observe $x_i$?

Instance feature vector (Observed)

$$x_1 \in \mathbb{R}^p$$

$$x_2$$

$$x_3$$

$$x_n$$
Assume these $n$ instances are from the past, i.e., “training” instances

$$\hat{c}_i = g(x_i; \theta) \in \mathbb{R}^d$$

$$\min_{\mathcal{S}} \{ \hat{c}_1^\top w : w \in \mathcal{S} \}$$

$$\min_{\mathcal{S}} \{ \hat{c}_2^\top w : w \in \mathcal{S} \}$$

$$\min_{\mathcal{S}} \{ \hat{c}_n^\top w : w \in \mathcal{S} \}$$

True cost vector (Observed for training instances only)

$$c_1 \in \mathbb{R}^d$$

$$c_2$$

$$c_n$$

Instance feature vector (Observed)

$$x_1 \in \mathbb{R}^p$$

$$x_2$$

$$x_n$$
Assume these \( n \) instances are from the past, i.e., “training” instances

\[
\hat{c}_i = g(x_i; \theta) \in \mathbb{R}^d
\]

\[
\min \left\{ \hat{c}_1^T w : w \in \mathcal{S} \right\}
\]

\[
\min \left\{ \hat{c}_2^T w : w \in \mathcal{S} \right\}
\]

\[
\min \left\{ \hat{c}_n^T w : w \in \mathcal{S} \right\}
\]

Goal: given training set \( \mathcal{D} = \{ (x_i, c_i) \}_{i=1}^n \), learn \( \theta \) that approximates groundtruth \( c_i \) vectors “well”

True cost vector (Observed for training instances only)

\[
c_1 \in \mathbb{R}^d
\]

\[
c_2
\]

\[
c_n
\]

Instance feature vector (Observed)

\[
x_1 \in \mathbb{R}^p
\]

\[
x_2
\]

\[
x_n
\]
Predict-then-Optimize
Training

Dataset $\mathcal{D}$ with data points $(x, c)$

Prediction model $g(x; \theta)$ with parameters $\theta$

Loss function $l(\cdot)$ to measure prediction error

$x \rightarrow \hat{c} = g(x; \theta) \rightarrow \text{Loss}$
Predict-then-Optimize
Training and Test-time inference

Dataset $\mathcal{D}$
with data points $(x, c)$

Prediction model $g(x; \theta)$
with parameters $\theta$

Loss function $l(\cdot)$
to measure prediction error

$x \xrightarrow{} \hat{c} = g(x; \theta)$

Test Time

Optimization solver
$w_\ast^c = \arg\min_{w \in \mathcal{S}} \hat{c}^T w$

Subject to $Aw = b$
Predict-then-Optimize
s-t shortest path: training data

$\mathbf{x}_1$ 8 am
$\mathbf{x}_2$ 10 am
$\mathbf{x}_3$ 12 pm
$\mathbf{x}_4$ 4 pm

Google Maps, Montréal, Québec, Canada
Predict-then-Optimize

s-t shortest path: full pipeline

Dataset $\mathcal{D}$
with data points $(x, c)$

Prediction model $g(x; \theta)$
with parameters $\theta$

Loss function $l(\cdot)$
to measure prediction error

Travel time prediction model
(regression) for each road segment

Predicted travel times $\Leftarrow$ s-t shortest path cost vector
Predict-then-Optimize

$s$-$t$ shortest path: full pipeline

Dataset $\mathcal{D}$ with data points $(x, c)$

Prediction model $g(x; \theta)$ with parameters $\theta$

Loss function $l(\cdot)$ to measure prediction error

Travel time prediction model (regression) for each road segment

Predicted travel times $\leftrightarrow$ $s$-$t$ shortest path cost vector

$\mathcal{c}_1, \mathcal{c}_2, \mathcal{c}_3, \mathcal{c}_4$
Predict-then-Optimize

$s$-$t$ shortest path: full pipeline

Dataset $\mathcal{D}$ with data points $(x, c)$

Prediction model $g(x; \theta)$ with parameters $\theta$

Loss function $l(\cdot)$ to measure prediction error

Travel time prediction model (regression) for each road segment

Predicted travel times $\leftrightarrow$ s-t shortest path cost vector

Squared error between $c_i$ and $\hat{c}_i$
Why regression on cost coefficients may fail
Vertical axis: edge cost. Horizontal axis: feature x

Why regression on cost coefficients may fail

Vertical axis: edge cost. Horizontal axis: feature x

Least-squares regression on dataset of (x, c) pairs

Why regression on cost coefficients may fail

Vertical axis: edge cost. Horizontal axis: feature x

Least-squares regression on dataset of (x, c) pairs

Why regression on cost coefficients may fail
Vertical axis: edge cost. Horizontal axis: feature $x$

Least-squares regression on dataset of $(x, c)$ pairs

“Smart Predict then Optimize”

To measure the error in decision-making, the notion of regret (also called SPO Loss) has been proposed and is defined as the difference between the optimal solution if you optimize with \( \hat{c} \) and the true costs \( c \):

\[
\text{Decision error} \quad l_{\text{Regret}}(\hat{c}, c) = c^T \mathbf{w}^*(\hat{c}) - z^*(c)
\]

**Optimal solution if you optimize with \( \hat{c} \)**

**Optimal value with true costs \( c \)**
Algorithm 1 End-to-end Gradient Descent

**Require:** coefficient matrix $A$, right-hand side $b$, data $D$

1: Initialize predictor parameters $\theta$ for predictor $g(x; \theta)$
2: for epochs do
3:   for each batch of training data $(x, c)$ do
4:     Sample batch of the cost vectors $c$ with the corresponding features $x$
5:     Predict cost using predictor $\hat{c} := g(x; \theta)$
6:     Forward pass to compute optimal solution $w^*(\hat{c}) := \arg\min_{w \in S} \hat{c}^T w$
7:     Forward pass to compute decision loss $l(\hat{c}, \cdot)$
8:     Backward pass from loss $l(\hat{c}, \cdot)$ to update parameters $\theta$ with gradient
9:   end for
10: end for
it is not necessary to consider it in practice. Therefore, although unambiguous regret is more theoretically rigorous, regret and the unambiguous regret are almost the same in all training procedures. However, as Figure 3 shows, the cost vector.

To measure the error in decision-making, the notion of regret (also called SPO Loss) has been proposed and is defined as the difference in objective value between an optimal solution (using the true but unknown cost vector) and one allowing customized loss functions.

3.3.1 Decision Loss

Therefore, Elmachtoub and Grigas [16] devised the "unambiguous" regret (also called unambiguous SPO Loss):

\[
\text{URegret} = \min_{w} c^T w \quad \text{s.t.} \quad Aw \leq b
\]

\[
\hat{w}_c^* = \arg\min_{w \in S} \hat{c}^T w
\]

\[
\hat{c} = g(x, \theta)
\]

\[
l(\cdot)\to \text{measure decision error}
\]

Algorithm 1 End-to-end Gradient Descent

Require: coefficient matrix $A$, right-hand side $b$, data $D$

1: Initialize predictor parameters $\theta$ for predictor $g(x; \theta)$
2: for epochs do
3: for each batch of training data $(x, c)$ do
4: Sample batch of the cost vectors $c$ with the corresponding features $x$
5: Predict cost using predictor $\hat{c} := g(x; \theta)$
6: Forward pass to compute optimal solution $\hat{w}^*(\hat{c}) := \arg\min_{w \in S} \hat{c}^T w$
7: Forward pass to compute decision loss $l(\hat{c}, \cdot)$
8: Backward pass from loss $l(\hat{c}, \cdot)$ to update parameters $\theta$ with gradient
9: end for
10: end for
\[
\frac{\partial l(\hat{c}, \cdot)}{\partial \theta} = \frac{\partial l(\hat{c}, \cdot)}{\partial \hat{c}} \frac{\partial \hat{c}}{\partial \theta}
\]

**Algorithm 1** End-to-end Gradient Descent

**Require:** coefficient matrix \( A \), right-hand side \( b \), data \( D \)

1: Initialize predictor parameters \( \theta \) for predictor \( g(x; \theta) \)
2: **for** epochs **do**
3: \hspace{1em} **for** each batch of training data \((x, c)\) **do**
4: \hspace{2em} Sample batch of the cost vectors \( c \) with the corresponding features \( x \)
5: \hspace{2em} Predict cost using predictor \( \hat{c} := g(x; \theta) \)
6: \hspace{2em} Forward pass to compute optimal solution \( w^*(\hat{c}) := \arg\min_{w \in S} \hat{c}^T w \)
7: \hspace{2em} Forward pass to compute decision loss \( l(\hat{c}, \cdot) \)
8: \hspace{2em} Backward pass from loss \( l(\hat{c}, \cdot) \) to update parameters \( \theta \) with gradient
9: \hspace{1em} **end for**
10: **end for**
It is not necessary to consider it in practice.

Therefore, although unambiguous regret is more theoretically rigorous, Elmachtoub and Grigas [16] devised the “unambiguous” regret (also called SPO Loss):

\[
\frac{\partial l(\hat{c}, \cdot)}{\partial \theta} = \frac{\partial l(\hat{c}, \cdot)}{\partial \hat{c}} \frac{\partial \hat{c}}{\partial \theta}
\]

Algorithm 1 End-to-end Gradient Descent

**Require:** coefficient matrix \(A\), right-hand side \(b\), data \(D\)

1: Initialize predictor parameters \(\theta\) for predictor \(g(x; \theta)\)
2: for epochs do
3:   for each batch of training data \((x, c)\) do
4:     Sample batch of the cost vectors \(c\) with the corresponding features \(x\)
5:     Predict cost using predictor \(\hat{c} := g(x; \theta)\)
6:     Forward pass to compute optimal solution \(w^*(\hat{c}) := \arg\min_{w \in S} \hat{c}^T w\)
7:     Forward pass to compute decision loss \(l(\hat{c}, \cdot)\)
8:     Backward pass from loss \(l(\hat{c}, \cdot)\) to update parameters \(\theta\) with gradient
9:   end for
10: end for
Trickier: how does the decision loss vary with the predicted costs?

\[
\frac{\partial l(\hat{c}, \cdot)}{\partial \theta} = \frac{\partial l(\hat{c}, \cdot)}{\partial \hat{c}} \frac{\partial \hat{c}}{\partial \theta}
\]

**Algorithm 1** End-to-end Gradient Descent

**Require:** coefficient matrix \( A \), right-hand side \( b \), data \( D \)

1: Initialize predictor parameters \( \theta \) for predictor \( g(x; \theta) \)
2: for epochs do
3: for each batch of training data \((x, c)\) do
4: Sample batch of the cost vectors \( c \) with the corresponding features \( x \)
5: Predict cost using predictor \( \hat{c} := g(x; \theta) \)
6: Forward pass to compute optimal solution \( \hat{w}^* := \text{argmin}_{w \in S} \hat{c}^T w \)
7: Forward pass to compute decision loss \( l(\hat{c}, \cdot) \)
8: Backward pass from loss \( l(\hat{c}, \cdot) \) to update parameters \( \theta \) with gradient
9: end for
10: end for
Trickier: how does the decision loss vary with the predicted costs?

\[
\frac{\partial l(\hat{c}, \cdot)}{\partial \theta} = \frac{\partial l(\hat{c}, \cdot)}{\partial \hat{c}} \frac{\partial \hat{c}}{\partial \theta}
\]

\[
l_{\text{Regret}}(\hat{c}, c) = \mathbf{c}^\top \mathbf{w}^*(\hat{c}) - z^*(c)
\]

**Algorithm 1** End-to-end Gradient Descent

**Require:** coefficient matrix \( A \), right-hand side \( b \), data \( \mathcal{D} \)

1: Initialize predictor parameters \( \theta \) for predictor \( g(x; \theta) \)
2: for epochs do
3:   for each batch of training data \((x, c)\) do
4:     Sample batch of the cost vectors \( c \) with the corresponding features \( x \)
5:     Predict cost using predictor \( \hat{c} := g(x; \theta) \)
6:     Forward pass to compute optimal solution \( \mathbf{w}^*(\hat{c}) := \min_{\mathbf{w} \in S} \hat{c}^\top \mathbf{w} \)
7:     Forward pass to compute decision loss \( l(\hat{c}, \cdot) \)
8:     Backward pass from loss \( l(\hat{c}, \cdot) \) to update parameters \( \theta \) with gradient
9:   end for
10: end for
**Trickier:** how does the decision loss vary with the predicted costs?

\[
\frac{\partial l(\hat{c}, \cdot)}{\partial \theta} = \frac{\partial l(\hat{c}, \cdot)}{\partial \hat{c}} \frac{\partial \hat{c}}{\partial \theta}
\]

**Crux:** how does the optimum change with the predicted costs?

\[
l_{\text{Regret}}(\hat{c}, c) = c^T w^*(\hat{c}) - z^*(c)
\]

**Algorithm 1** End-to-end Gradient Descent

**Require:** coefficient matrix \(A\), right-hand side \(b\), data \(D\)

1. Initialize predictor parameters \(\theta\) for predictor \(g(x; \theta)\)
2. **for** epochs **do**
3. **for** each batch of training data \((x, c)\) **do**
4. Sample batch of the cost vectors \(c\) with the corresponding features \(x\)
5. Predict cost using predictor \(\hat{c} := g(x; \theta)\)
6. Forward pass to compute optimal solution \(w^*(\hat{c}) := \arg\min_{w \in S} \hat{c}^T w\)
7. Forward pass to compute decision loss \(l(\hat{c}, \cdot)\)
8. Backward pass from loss \(l(\hat{c}, \cdot)\) to update parameters \(\theta\) with gradient
9. **end for**
10. **end for**
SPO+: a principled and effective method


\[ l_{\text{Regret}}(\hat{c}, c) = c^T w^*(\hat{c}) - z^*(c) \]

is not differentiable w.r.t. \( \hat{c} \)

---

**Algorithm 1 End-to-end Gradient Descent**

**Require:** coefficient matrix \( A \), right-hand side \( b \), data \( D \)

1. Initialize predictor parameters \( \theta \) for predictor \( g(x; \theta) \)
2. for epochs do
   3. for each batch of training data \( (x, c) \) do
   4. Sample batch of the cost vectors \( c \) with the corresponding features \( x \)
   5. Predict cost using predictor \( \hat{c} := g(x; \theta) \)
   6. Forward pass to compute optimal solution \( w^*(\hat{c}) := \arg\min_{w \in S} \hat{c}^T w \)
   7. Forward pass to compute decision loss \( l(\hat{c}, \cdot) \)
   8. Backward pass from loss \( l(\hat{c}, \cdot) \) to update parameters \( \theta \) with gradient
3. end for
4. end for

---
SPO+: a principled and effective method


\[ l_{\text{Regret}}(\hat{c}, c) = c^T w^*(\hat{c}) - z^*(c) \]

is not differentiable w.r.t. \( \hat{c} \)

\[ l_{SPo+}(\hat{c}, c) = -\min_{w \in S} \{(2\hat{c} - c)^T w\} + 2\hat{c}^T w^*(c) - z^*(c) \]

is a convex upper bound on regret and has subgradients

**Algorithm 1 End-to-end Gradient Descent**

**Require:** coefficient matrix \( A \), right-hand side \( b \), data \( D \)
1. Initialize predictor parameters \( \theta \) for predictor \( g(x; \theta) \)
2. for epochs do
3. for each batch of training data \((x, c)\) do
4. Sample batch of the cost vectors \( c \) with the corresponding features \( x \)
5. Predict cost using predictor \( \hat{c} := g(x; \theta) \)
6. Forward pass to compute optimal solution \( w^*(\hat{c}) := \arg\min_{w \in S} \hat{c}^T w \)
7. Forward pass to compute decision loss \( l(\hat{c}, \cdot) \)
8. Backward pass from loss \( l(\hat{c}, \cdot) \) to update parameters \( \theta \) with gradient
9. end for
10. end for
SPO+: a principled and effective method


\[
l_{\text{Regret}}(\hat{c}, c) = c^T w^* (\hat{c}) - z^* (c)
\]

is not differentiable w.r.t. \( \hat{c} \)

\[
l_{\text{SPO+}}(\hat{c}, c) = -\min_{w \in S} \left\{ (2\hat{c} - c)^T w \right\} + 2\hat{c}^T w^* (c) - z^* (c)
\]

is a convex upper bound on regret and has subgradients

Algorithm 1 End-to-end Gradient Descent

Require: coefficient matrix \( A \), right-hand side \( b \), data \( D \)
1: Initialize predictor parameters \( \theta \) for predictor \( g(x; \theta) \)
2: for epochs do
3:   for each batch of training data \( (x, c) \) do
4:     Sample batch of the cost vectors \( c \) with the corresponding features \( x \)
5:     Predict cost using predictor \( \hat{c} := g(x; \theta) \)
6:     Forward pass to compute optimal solution \( w^*(\hat{c}) := \arg\min_w_{\in S} \hat{c}^T w \)
7:     Forward pass to compute decision solution \( l(\hat{c}, \cdot) \)
8:     Backward pass from loss \( l(\hat{c}, \cdot) \) to update parameters \( \theta \) with gradient
9:   end for
10: end for

Computational overhead: To compute SPO+ loss, we need to solve an optimization problem in the forward pass; this is shared with other methods.
Why regression on cost coefficients may fail
Vertical axis: edge cost. Horizontal axis: feature x

Elmachtoub and Grigas: Smart "Predict, then Optimize"

Figure 3 Illustrative Example.

Note. The circles correspond to edge 1 costs and the squares correspond to edge 2 costs. Red lines and points correspond to the least squares fit and predictions, while green lines and points correspond to the SPO fit and predictions. The vertical dotted lines correspond to the decision boundaries under the true and prediction models.

The SPO+ decision boundary in this stylized example coincides with the SPO decision boundary.

Finding the prediction model that minimizes the empirical risk using the SPO+ loss, this prediction model will also approximately minimize (4), the empirical risk using the SPO loss.

To begin the derivation of the SPO+ loss, we first observe that for any $\alpha \in \mathbb{R}$, the SPO loss can be written as

$$\ell_{SPO}(\hat{c}, c) = \max_{w \in W^*}(\hat{c}) \{ c^T w - \alpha \hat{c}^T w \} + \alpha z^*(\hat{c}) - z^*(c)$$

(5)

Predict-then-Optimize
s-t shortest path: training data

\[ x_1 \]
\[ x_2 \]
\[ x_3 \]
\[ x_4 \]

Google Maps, Montréal, Quebéc, Canada
What if the historical cost vectors were unavailable, but the optimal solutions were?
\[ \mathcal{D} = \{(x_i, w^*_i)\}_{i=1}^n \]
PFYL: Perturbed Fenchel-Young Loss


Recall the required gradient:

\[
\frac{\partial l(\hat{c}, \cdot)}{\partial \theta} = \frac{\partial l(\hat{c}, \cdot)}{\partial \hat{c}} \frac{\partial \hat{c}}{\partial \theta}
\]

A random perturbation \((\sigma \xi_k)\) of predicted costs \(\hat{c}\)

\[
\frac{\partial l_{FY}(\hat{c}, w^*(c))}{\partial \hat{c}} \approx w^*(c) - \frac{1}{K} \sum_{\kappa}^{K} \text{argmin}_{w \in S} \{(\hat{c} + \sigma \xi_k)\^T w\}
\]

Average optimal solution over \(K\) randomly perturbed costs
PFYL: Perturbed Fenchel-Young Loss

Recall the required gradient:
\[
\frac{\partial l(\hat{c}, \cdot)}{\partial \theta} = \frac{\partial l(\hat{c}, \cdot)}{\partial \hat{c}} \frac{\partial \hat{c}}{\partial \theta}
\]

A random perturbation \((\sigma \xi_k)\) of predicted costs \(\hat{c}\)

\[
\frac{\partial l_{FY}(\hat{c}, w^*(c))}{\partial \hat{c}} \approx w^*(c) - \frac{1}{K} \sum_{\kappa}^{K} \text{argmin}_{w \in S} \{(\hat{c} + \sigma \xi_{\kappa})^T w\}
\]

Notice the true costs \(c\) do not appear here.

Average optimal solution over \(K\) randomly perturbed costs
**PFYL: Perturbed Fenchel-Young Loss**


Recall the required gradient:

\[
\frac{\partial l(\hat{c}, \cdot)}{\partial \theta} = \frac{\partial l(\hat{c}, \cdot)}{\partial \hat{c}} \frac{\partial \hat{c}}{\partial \theta}
\]

**A random perturbation** \((\sigma \xi_k)\) **of predicted costs** \(\hat{c}\)

\[
\frac{\partial l_{FY}(\hat{c}, w^*(c))}{\partial \hat{c}} \approx w^*(c) - \frac{1}{K} \sum_{\kappa} \arg\min_{w \in S} \{(\hat{c} + \sigma \xi_{\kappa})^T w\}
\]

Notice the true costs \(c\) do not appear here.

**Computational overhead:** To compute the PFYL gradient, we need to solve \(K\) optimization problems.

**Average optimal solution** over \(K\) randomly perturbed costs
### Avoiding solver calls by Learning to Rank


Recall that both SPO+ and PFYL made one or more calls to the solver in each forward pass!

<table>
<thead>
<tr>
<th></th>
<th>w1</th>
<th>w2</th>
<th>w3</th>
<th>Objective using prediction A</th>
<th>Objective using prediction B</th>
<th>True Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sol 1</strong></td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td><strong>Sol 2</strong></td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td><strong>Sol 3</strong></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
Avoiding solver calls by Learning to Rank


<table>
<thead>
<tr>
<th>Sol 1</th>
<th>w1</th>
<th>w2</th>
<th>w3</th>
<th>Objective using prediction A</th>
<th>Objective using prediction B</th>
<th>True Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
### Avoiding solver calls by Learning to Rank


<table>
<thead>
<tr>
<th></th>
<th>w1</th>
<th>w2</th>
<th>w3</th>
<th>Using Prediction</th>
<th>Using Prediction</th>
<th>True Objective</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sol 1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Sol 2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Sol 3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

**Key Idea:** Learn to predict coefficients that lead to good rankings of a set of collected solutions!

- **Somewhat better...**
- **Terrible solution ranking!**
Predict-then-Optimize: a tour of the state-of-the-art using PyEPO

Manuscript describing PyEPO, the literature, and extensive experiments: https://arxiv.org/abs/2206.14234
Welcome to PyEPO's documentation!

This is the documentation of PyEPO (PyTorch-based End-to-End Predict-then-Optimize Library Tool), which aims to provide end-to-end methods for predict-then-optimize tasks.

Sample Code

```python
import random
import gurobipy as gp
from gurobipy import GRB
```
Benchmarks

Optimization modeling

ML modeling

Training algorithms

Shortest path

Knapsack

TSP

Gurobi

Any custom solver

SPO+

SPO+ with relaxations

PFYL/DPO

DBB

(Pogancic et al. [2019])
Benchmark generation
Based on Elmachtoub & Grigas

\[ c_{ij} = \alpha \left( (Bx_i)_j + 3 \right)^{deg} \cdot \epsilon_{i,j} \]

- \( c_{ij} \): Benchmark value
- \( x_i \): i-th instance's feature vector
- \( \alpha \): Constant
- \( B \): Integer in \([1,6] \) regulating how non-linear the mapping is
- \( (Bx_i)_j \): Element of the transformed feature vector
- \( \epsilon_{i,j} \): Uniform random noise
- \( deg \): Integer in \([0,6] \)

\( x_i \in \mathbb{R}^p \)

- \( x_i \): i-th instance's feature vector
- \( \mathbb{R}^p \): p-dimensional real space

\( B \in \{0,1\}^{d \times n} \)

- \( B \): Boolean matrix
- \( d \times n \): Dimension of matrix

\( B_{p,q} \sim \text{Bernoulli}(0.5) \)

- \( B_{p,q} \): Element of the Boolean matrix
- \( \text{Bernoulli}(0.5) \): Bernoulli distribution with probability 0.5

Uniform random noise
We compare the performance between two-stage methods.

### 6.1 Performance Comparison between Different Methods

Specifically, we used loss for knapsack and TSP. For the sake of consistency, we only use regret (4) as the sensible for the shortest path problem but leads to much worse decisions in experiments, compared to the regret, et al. [6] employed the squared difference between solutions. However, in our distance between the true optimum and the predicted solution, while Berthet differentiation of blackbox combinatorial solvers.

As Table 3 shows, the methods we compare include the two-stage approach normalized regret (as defined in Section 4.5) on a test set of size $n = 1000$. The experiments aimed to investigate the training time and polynomials de-
Similarly, there is no need to assign a value to the attribute.

On the other hand, we provide Pyomo, where it is more lightweight but less flexible for users. Let us use the following optimization model (7) as an example:

\[ \text{max} \sum_{i=0}^{4} c_i x_i \]
\[ \text{s.t.} \quad 3x_0 + 4x_1 + 3x_2 + 6x_3 + 4x_4 \leq 12 \]
\[ 4x_0 + 5x_1 + 2x_2 + 3x_3 + 5x_4 \leq 10 \]
\[ 5x_0 + 4x_1 + 6x_2 + 2x_3 + 3x_4 \leq 15 \]
\[ \forall x_i \in \{0,1\} \]

Inheriting from EPO allows modeling mathematical programs with multiple solvers, instantiating optOmoModel requires an explicit object to create an optimization model.

In contrast to optGrbModel, where the only implementation required is to override getModel. Since optOmoModel supports multiple solvers, instantiating optGrbModel allows modeling mathematical programs with EPO.

An example for Model (7) is as follows:

```python
import gurobipy as gp
from gurobipy import GRB
from pyepo.model.grb import optGrbModel

class myModel(optGrbModel):
    def _getModel(self):
        # create a model
        m = gp.Model()
        # variables
        x = m.addVars(5, name="x", vtype=GRB.BINARY)
        # sense (must be minimize)
        m.modelSense = GRB.MAXIMIZE
        # constraints
        return m, x

optmodel = myModel()```
Creating a dataset based on features and true costs

```python
import pyepo
from torch.utils.data import DataLoader

# build dataset
dataset = pyepo.data.dataset.optDataset(optmodel, feats, costs)

# get data loader
dataloader = DataLoader(dataset, batch_size=32, shuffle=True)

for x, c, w, z in dataloader:
    # a batch of features
    print(x)
    # a batch of true costs
    print(c)
    # a batch of true optimal solutions
    print(w)
    # a batch of true optimal objective values
    print(z)
```
Creating an ML model with PyTorch

```python
from torch import nn

def __init__(self):
    super(LinearRegression, self).__init__()
    self.linear = nn.Linear(num_feat, len_cost)

def forward(self, x):
    out = self.linear(x)
    return out

predmodel = LinearRegression()
```
4.4 End-to-End Training

The core capability of PyEPO is to build an optimization model, and then embed the optimization model into a PyTorch neural network for the end-to-end training. Here, we build a simple linear regression model in PyTorch as an example:

```python
import torch

# set SGD optimizer
optimizer = torch.optim.SGD(predmodel.parameters(), lr=1e-3)

# training
for epoch in range(num_epochs):
    # iterate features, costs, solutions, and objective values
    for x, c, w, z in dataloader:
        # forward pass
        cp = predmodel(x)  # predict costs
        loss = spop(cp, c, w, z).mean()  # calculate SPO+ loss
        # backward pass
        optimizer.zero_grad()  # reset gradients to 0
        loss.backward()  # compute gradients
        optimizer.step()  # update model parameters
```

4.5 Metrics

PyEPO provides evaluation functions to measure model performance, in particular the two metrics mentioned in Section 3.3.1: regret and unambiguous regret. We further define the normalized (unambiguous) regret by

$$\text{Regret}(\hat{c}_i, c_i) \quad P_{test} = \frac{1}{l} \sum_{i=1}^{l} |z^*(c_i)|.$$
Experiments with TSP20

Vertical axis: average regret w.r.t. true OPT on unseen test instances

2-stage methods: regress on true costs (no end-to-end training)

Fig. 8: Normalized regret for the TSP problem on the test set: There are 20 nodes to visit. The methods in the experiment include two-stage approaches with linear regression, random forest and Auto-Sklearn and end-to-end learning such as 2-stage LR, 2-stage RF, SPO+, PFYL, DBB. The normalized regret is visualized under different sample sizes, noise half-width, and polynomial degrees. For normalized regret, lower is better.

6.2 Two-stage Method with Automated Hyperparameter Tuning

This method leverages the sophisticated Auto-Sklearn tool that uses bayesian optimization methods for automated hyperparameter tuning of Scikit-Learn regression models. The metric of "2-stage Auto" is the mean squared error of the predicted costs, which does not reduce decision error directly. Because of the limitation of multioutput regression in Auto-Sklearn v0.14.6, the choices of the predictor in 2-stage Auto only include five models: k-nearest neighbor (KNN), decision tree, random forest, extra-trees, and Gaussian process. Even with these limitations, Auto-Sklearn can achieve a low regret. Although the training of 2-stage Auto is time-consuming, it is still a competitive method.

2-stage methods: regress on true costs (no end-to-end training)

Harder learning task

Lower is better
Experiments with TSP20

Vertical axis: average regret w.r.t. true OPT on unseen test instances

2-stage methods: regress on true costs (no end-to-end training)

Training of 2-stage Auto is time-consuming, it is still a competitive method. Even
with these limitations, (KNN), decision tree, random forest, extra-trees, and Gaussian process. Even
of the predictor in 2-stage Auto only include five models: k-nearest neighbor
the limitation of multioutput regression in
the predicted costs, which does not reduce decision error directly. Because of
regression models. The metric of “2-stage Auto” is the mean squared error of
optimization methods for automated hyperparameter tuning of

This method leverages the sophisticated

6.2 Two-stage Method with Automated Hyperparameter Tuning

SPO+ and PFYL, both using a linear model perform the best.
With more data, 2-stage Random Forest is competitive!
Regret-accuracy tradeoff

Vertical axis: average regret w.r.t. true OPT on unseen test instances

Horizontal axis: Mean-Squared Error on true costs

\[ l_{\text{MSE}}(\hat{c}, c) = \frac{1}{2n} \sum_{i}^{n} \| \hat{c}_i - c_i \|_2^2 \]
We compare the performance between two-stage methods, specifically the two-stage approach with varying training data size using the Hamming distance. Notably, the normalized regret (as defined in Section 4.5) on a test set of size $n = 1000$. The experiments aimed to investigate the training time and validation accuracy.

In this section, we present experimental results for the benchmark datasets. We use the knapsack and TSP problems. For the sake of consistency, we only use regret (4) as the sensible for the shortest path problem but leads to much worse decisions in experiments, compared to the regret. Pogančić et al. [35] employed the squared distance between the true optimum and the predicted solution, while Berthet [6] used the Hamming distance. Based on Pogančić, Marin Vlastelica, et al. "Differentiation of blackbox combinatorial solvers." ICLR 2020.

### Warcraft Shortest-Path Benchmark


**Image inputs (RGB features)**

**True costs**

**Optimal solution (NW -> SE)**

![Image](image.png)

<table>
<thead>
<tr>
<th>0.8</th>
<th>0.8</th>
<th>0.8</th>
<th>0.8</th>
<th>0.8</th>
<th>0.8</th>
<th>0.8</th>
<th>1.2</th>
<th>1.2</th>
<th>1.2</th>
<th>0.8</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>9.2</td>
<td>1.2</td>
</tr>
<tr>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>1.2</td>
<td>0.8</td>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>7.7</td>
<td>7.7</td>
<td>1.2</td>
</tr>
<tr>
<td>1.2</td>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>0.8</td>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>7.7</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>1.2</td>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>1.2</td>
<td>7.7</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>1.2</td>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>1.2</td>
<td>7.7</td>
<td>1.2</td>
</tr>
<tr>
<td>1.2</td>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>1.2</td>
<td>7.7</td>
</tr>
<tr>
<td>1.2</td>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>0.8</td>
<td>1.2</td>
</tr>
<tr>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>7.7</td>
<td>7.7</td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>7.7</td>
<td>7.7</td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>7.7</td>
<td>7.7</td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>7.7</td>
<td>7.7</td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>7.7</td>
<td>7.7</td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>7.7</td>
<td>7.7</td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>7.7</td>
<td>7.7</td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
<td>7.7</td>
<td>7.7</td>
<td></td>
</tr>
</tbody>
</table>

Optimal solution (NW -> SE): 7.7 7.7 7.7 1.2 1.2 1.2 1.2 1.2 1.2 1.2
Warcraft Shortest-Path Benchmark

Image inputs

ResNet-18

Cost predictions
**Warcraft Shortest-Path Benchmark**


---

2-Stage, SPO+, and PFYL, all using a truncated ResNet-18, perform the best. DBB, originally benchmarked on this dataset, is far worse.

---

SPO+ and PFYL match true optimal paths better than other methods, including 2-Stage.
Predict-then-Optimize is a highly practical paradigm.

SPO+ and PFYL are very effective end-to-end learning methods.

The “naive” 2-stage approach is sufficient training set is large.

Open questions:

- Predictions in the constraints; see Hu, Xinyi, Jasper CH Lee, and Jimmy HM Lee. "Branch & Learn with Post-hoc Correction for Predict+ Optimize with Unknown Parameters in Constraints." CPAIOR 2023.

- Reducing training time; see work by Tias Guns and collaborators.

- More applications; see work by B. Dilkina, M. Tambe, B. Wilder, H. Bastani
## Machine Learning for Integer Programming

<table>
<thead>
<tr>
<th>SL + GNN</th>
<th>Mixed-Integer Linear Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[TMLR-22]</td>
</tr>
<tr>
<td></td>
<td>[AAAI-22]</td>
</tr>
<tr>
<td></td>
<td>[NeurIPS-20]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SL + Simple models</th>
<th>[IJCAI-17]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[AAAI-16]</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Custom ML</th>
<th>[AAAI-22]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>[NeurIPS-21]</td>
</tr>
</tbody>
</table>

- **SL**: Supervised Learning
- **RL**: Reinforcement Learning
- **GNN**: Graph Neural Networks

- Branch
- Schedule heuristics
- Select nodes
- Warmstart solver
- Detect backdoors
<table>
<thead>
<tr>
<th>Method</th>
<th>MILP</th>
<th>Column Generation</th>
<th>Stochastic Programming</th>
<th>Graph Optimization</th>
<th>Multiobjective optimization</th>
</tr>
</thead>
<tbody>
<tr>
<td>RL + GNN</td>
<td>[NeurIPS-22]</td>
<td></td>
<td></td>
<td>[TMLR-22]</td>
<td>[NeurIPS-17]</td>
</tr>
<tr>
<td>SL + GNN</td>
<td>[TMLR-22]</td>
<td>[AAAI-22]</td>
<td></td>
<td></td>
<td>[NeurIPS-20]</td>
</tr>
<tr>
<td>Custom ML</td>
<td>[AAAI-22]</td>
<td>[NeurIPS-21]</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Lab Colab:

https://tinyurl.com/ACP23-PredictAndOptimize

PyEPO Github/Docs:

https://github.com/khalil-research/PyEPO