Disciplined Convex Stochastic Programming: A New Framework for Stochastic Optimization

Alnur Ali
Machine Learning Dept.
Carnegie Mellon University
alnurali@cmu.edu

J. Zico Kolter
School of Computer Science
Carnegie Mellon University
zkolter@cs.cmu.edu

Steven Diamond
Dept. of Computer Science
Stanford University
stevend2@stanford.edu

Stephen Boyd
Dept. of Electrical Engineering
Stanford University
boyd@stanford.edu

Abstract

We introduce *disciplined convex stochastic programming* (DCSP), a modeling framework that can significantly lower the barrier for modelers to specify and solve convex stochastic optimization problems, by allowing modelers to naturally express a wide variety of convex *stochastic programs* in a manner that reflects their underlying mathematical representation. DCSP allows modelers to express expectations of arbitrary expressions, *partial optimizations*, and *chance constraints* across a wide variety of convex optimization problem families (*e.g.*, linear, quadratic, second order cone, and semidefinite programs). We illustrate DCSP's expressivity through a number of sample implementations of problems drawn from the operations research, finance, and machine learning literatures.

1 INTRODUCTION

We introduce *disciplined convex stochastic programming* (DCSP), a modeling framework for specifying and solving convex *stochastic programs*: convex optimization problems that include random variables. DCSP builds on principles from stochastic optimization and convex analysis to allow modelers to naturally express a wide variety of stochastic programs in a manner that reflects their underlying mathematical representation. At a high level, DCSP enables modelers to specify — in a straightforward way — and solve convex optimization problems that include (1) expectations of arbitrary expressions, (2) *partial optimizations*, optimizations over (only) a subset of the optimization variables, which additionally pave the way for the specification of *multi-stage* stochastic programs (Sec. 2.1), and (3) *chance constraints*, constraints that are required to hold with high probability — these three building blocks can be used to express a wide variety of stochastic optimization problems.

Concurrently with this paper, we also make available an open source Python implementation of DCSP, which we refer to as `cvxstoc`[1], that allows modelers to write and solve stochastic programs — we present a variety of examples of using `cvxstoc` to model stochastic optimization problems, drawn from the operations research, finance, and machine learning literatures, in Sec. 4.

Related work Although other frameworks for stochastic programming do exist ([24, 20, 11], and in Python mainly [26]), they often require significant effort from the modeler to manipulate the optimization problem into an amenable form, support a limited number of stochastic programming constructs (*e.g.*, [11] only supports chance constraints with uncertainty sets), and cannot express certain families of convex optimization problems; indeed, checking the convexity of and solving (convex) optimization problems in general is challenging. DCSP builds on (and extends) *disciplined convex programming* (DCP) [10], a recently introduced framework that makes it natural for modelers to express convex optimization problems, and additionally automates the tasks of verifying the convexity of these problems and translating them into conic form (see, *e.g.*, [8]). This means that DCSP can be used to express and solve a wide variety of stochastic convex optimization problems, including linear, quadratic, second order cone, and semidefinite programs. Probabilistic programming languages (*e.g.*, [9, 16, 13]) offer an alternative approach, but tend to focus on inference problems, and may not contain the features to capture traditional stochastic programming problem formulations; in contrast, convex modeling can be attractive because local solutions are global solutions, efficient solvers exist, and guarantees can often be obtained on the optimality of a solution obtained by a solver.

This paper is structured as follows. In Sec. 2, we review background on stochastic programming, DCP, and `cvxpy` (an open source Python implementation of DCP). In Sec. 3, we describe DCSP and `cvxstoc`'s syntax. In

[1] `cvxstoc` is available as an extension of the `cvxpy` Python package [7]: see `http://www.cvxpy.org`.

Sec. 4, we present a number of examples that illustrate our framework.

2 BACKGROUND

2.1 STOCHASTIC PROGRAMMING

A convex optimization problem has the form

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, m, \\ & h_i(x) = 0, \quad i = 1, \ldots, p, \end{array}$$

where $x \in \mathbf{R}^n$ is the optimization variable, $f_0 : \mathbf{R}^n \to \mathbf{R}$ is a convex objective function, $f_i : \mathbf{R}^n \to \mathbf{R},\ i = 1, \ldots, m$ are convex inequality constraint functions, and $h_i : \mathbf{R}^n \to \mathbf{R},\ i = 1, \ldots, p$ are affine equality constraint functions.

A convex stochastic program has the form

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \mathbf{E}\, f_0(x, \omega) \\ \text{subject to} & \mathbf{E}\, f_i(x, \omega) \leq 0, \quad i = 1, \ldots, m \\ & h_i(x) = 0, \quad i = 1, \ldots, p, \end{array} \tag{1}$$

where $f_i : \mathbf{R}^n \times \mathbf{R}^q \to \mathbf{R},\ i = 0, \ldots, m$ are convex functions in x for each value of a random variable $\omega \in \mathbf{R}^q$, and $h_i : \mathbf{R}^n \to \mathbf{R},\ i = 1, \ldots, p$ are (deterministic) affine functions; since expectations preserve convexity, the objective and inequality constraint functions in (1) are (also) convex in x, making (1) a convex optimization problem.

Two-stage stochastic programs An important special case of (1) is a so-called *two-stage* stochastic program (also referred to as an optimization problem with *recourse*) [6]:

$$\begin{array}{ll} \underset{x}{\text{minimize}} & f_0(x) + \mathbf{E}\, Q(x, \omega) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, \ldots, m, \\ & h_i(x) = 0, \quad i = 1, \ldots, p, \end{array} \tag{2}$$

$$\text{where } Q(x, \omega) = \inf_y \{\phi_0(x, y, \omega) : \phi_i(x, y, \omega) \leq 0,\ \psi_j(x, y) = 0,\ i = 1, \ldots, s,\ j = 1, \ldots, w\}$$

is the *second stage* problem, $y \in \mathbf{R}^r$ is the second stage optimization variable, $\phi_0 : \mathbf{R}^n \times \mathbf{R}^r \times \mathbf{R}^q \to \mathbf{R}$ is the second stage objective function, and is convex in (x, y) for each value of ω, $\phi_i : \mathbf{R}^n \times \mathbf{R}^r \times \mathbf{R}^q \to \mathbf{R},\ i = 1, \ldots, s$ are the second stage inequality constraint functions, also convex in (x, y) for each value of ω, and $\psi_i : \mathbf{R}^n \times \mathbf{R}^r \to \mathbf{R},\ i = 1, \ldots, w$ are the second stage equality constraint functions, and are affine in (x, y). That is, Q is itself the optimal value of another convex optimization problem, and is convex in x for each value of ω.

Two-stage stochastic programs model the uncertain consequences (in the second stage) of here-and-now decision-making (in the first stage): *e.g.*, in a finance application, we may wish to decide which assets to purchase now, while (also) factoring in how the asset prices might fluctate later.

Chance-constrained problems A *chance constraint* [5] is a constraint on the variable x of the form

$$\mathbf{Prob}\,(f(x, \omega) \leq 0) \geq \eta,$$

where f is convex in x for each value of ω, and η is typically a large probability (*e.g.*, 0.95); a *chance-constrained problem* is an optimization problem with one or more chance constraints. Chance constraints are typically non-convex, although effective convex approximations exist (see Sec. 3.3).

2.2 DISCIPLINED CONVEX PROGRAMMING

Disciplined convex programming (DCP) is a recently introduced modeling framework for specifying and solving convex optimization problems [10]. In a nutshell, DCP consists of a library of convex atomic functions, and a *convex rule-set* that prescribes how these atomic functions may be composed to express (more complex) convex optimization problems.

Convex rule-set Verifying the convexity of arbitrary expressions is challenging; DCP checks convexity using Thm. 2.1, which is equivalent to enforcing a set of rules.

Theorem 2.1 ([10]). *Suppose $f = h(g_1(x), \ldots, g_k(x))$, where $h : \mathbf{R}^k \to \mathbf{R}$ is convex and $g_i : \mathbf{R}^n \to \mathbf{R},\ i = 1, \ldots, k$, and one of the following holds for each $i = 1, \ldots, k$:*

- *g_i is convex and h is nondecreasing in argument i*
- *g_i is concave and h is nonincreasing in argument i*
- *g_i is affine.*

Then f is convex[2].

Thm. 2.1 permits a wide variety of convex expressions: for example, the maximum eigenvalue of a symmetric matrix, $\lambda_{\max}(2X - 4I)$, where $X \in \mathbf{S}^n$, is recognized as convex. (On the other hand, as an example of a limitation of the rule-set, the expression $\left(\sum_{i=1}^n x_i^2\right)^{1/2}$, where $x \in \mathbf{R}^n$, *is not* recognized as convex, although it *is* recognized as convex once reformulated as $\|x\|_2$.)

Library of atoms Atomic convex functions[3] are specified in DCP in their epigraph form: for example, the (convex) function $f(x) = \|x\|_1$ is specified as

$$\begin{array}{ll} \underset{t}{\text{minimize}} & \mathbf{1}^T t \\ \text{subject to} & -t \preceq x \preceq t, \end{array} \tag{3}$$

[2] A similar result holds for concave functions.

[3] See `http://www.cvxpy.org/en/latest/tutorial/functions/index.html` for a list of the convex atoms available in `cvxpy`.

where $x, t \in \mathbf{R}^n$. Thus, whenever a modeler writes the atom $f(x) = \|x\|_1$, DCP internally replaces it with (3), introduces the variable t, and can subsequently optimize over (x, t).

Disciplined convex programming DCP certifies a problem's convexity by constructing an abstract syntax tree for the objective and constraint functions, with atoms as internal nodes, and variables and constants as leaves, and then applying Thm. 2.1 recursively [10, 22].

2.3 cvxpy

`cvxpy` [7] is an open source Python DCP implementation; we briefly describe its syntax next.

Variables are declared simply in `cvxpy` as follows:

```
x = Variable()
x = NonNegative()
X = Semidefinite(n)
```

The first line declares `x` to be a variable in $\mathbf{R}$, the second declares `x` to be a variable in the nonnegative orthant $\mathbf{R}_+$, and the third declares `X` to be a ($n \times n$ matrix) variable in the positive semidefinite cone $\mathbf{S}_+^n$.

Convex expressions are specified by composing convex atoms; for example, the log loss $\sum_{i=1}^m \log(1 + \exp(-y_i(w^T x_i + b)))$ can be specified by using the simpler `log_sum_exp` atom as follows:

```
expr = [log_sum_exp(vstack(0, -y[i]*(w.T*x[i]+b)))
        for i in range(m)]
```

An objective is specified by instantiating a *sense* (*i.e.*, `Minimize` or `Maximize`) with an expression:

```
obj = x.T*c
Minimize(obj)
```

Constraints are specified by forming a list of expressions:

```
constrs = [x >= 0, x.T*numpy.ones((n,1)) == 1, ...]
```

A convex optimization problem, then, is specified by instantiating a `Problem` with an objective and a list of constraints:

```
prob = Problem(Minimize(obj), constrs)
prob.solve()
```

The last line solves the optimization problem.

3 DISCIPLINED CONVEX STOCHASTIC PROGRAMMING

In this section we present the chief methodological contribution of the paper: the disciplined convex stochastic programming (DCSP) framework, along with an overview of its implementation in the `cvxstoc` Python package. In a nutshell, DCSP consists of the addition of three operations to the disciplined convex programming (DCP) framework, which can be used to express a wide variety of convex stochastic programs: the ability to (1) compute (approximations to) expectations of arbitrary expressions, (2) handle *partial optimization*, and (3) compute (approximations to) chance constraints.

3.1 RANDOM VARIABLES AND EXPECTATIONS

Random variables The most fundamental operations in stochastic programs, and hence in DCSP, are the ability to specify random variables, and compute (approximations to) expectations of arbitrary expressions containing these random variables. As in Sec. 2, DCSP assumes that all expressions in a stochastic program are convex in the optimization variable(s) for each value of the random variable(s) — thus, from the point of view of DCSP, random variables do not affect the convexity of their parent expressions and can be regarded as equivalent to constants, thereby requiring no additions to the DCP convex rule-set. (Practically speaking, DCSP permits the specification of a variety of random variables; see Sec. 3.4.)

Expectations DCSP computes (approximations to) expectations of arbitrary expressions using simple Monte Carlo evaluation, *i.e.*,

$$\mathbf{E}\, f(x, \omega) \approx (1/N) \sum_{i=1}^{N} f(x, \omega_i),$$

where f is (again) assumed to be convex in the optimization variable x for each value of the random variable ω, and $\omega_i,\ i = 1, \ldots, N$ are samples of ω; this approximation is referred to as the *sample average approximation* (SAA) in the stochastic programming literature, and methods that use it are often referred to as *scenario-based* methods. By the DCP rule-set, the nonnegative weighted sum of convex functions is a convex function; thus, the expectation operator applied to an expression that is convex in x returns an expression that is (also) convex in x.

The SAA is, of course, a very simple method for approximating an expectation, and much more involved methods for solving stochastic programs exist, but the clear advantage of this method is its simplicity: any random variable can be included in a stochastic program as long as we are able to draw samples of it. If ω is a discrete random variable, then DCSP calculates its expectation exactly; otherwise, DCSP draws samples using Markov chain Monte Carlo (MCMC) methods[4] [15].

In the case of unconstrained stochastic programs, the SAA objective value is (naturally) an unbiased estimator of the

[4] We implement MCMC by leveraging the `PyMC` Python package [15].

true objective value, $\mathbf{E}\, f_0(x,\omega)$, with variance $\propto 1/N$, and an asymptotically normal distribution [21, chap. 5]. Thm. 3.1 additionally tells us that (roughly) both the optimal value and optimal set of a SAA converge almost surely to the optimal value and optimal set of the true problem.

Theorem 3.1 ([21, Thm. 5.3])**.** *Define $\hat{p}_N^*, \hat{S}_N$ and p^*, S as the optimal value and optimal set of a SAA with N samples and of the true problem, respectively, and let $K \subset \mathbf{R}^n$ be a compact set. Suppose (a) $S \subseteq K$ is nonempty, (b) $\hat{S}_N \subseteq K$ is nonempty a.s., (c) f_0 is finite and continuous on K, and (d) $(1/N)\sum_{i=1}^N f_0(x,\omega_i) \xrightarrow{\text{a.s.}} f(x)$ (uniformly) for $x \in K$. Then $\hat{p}_N^* \xrightarrow{\text{a.s.}} p^*$ and $\sup_{x\in\hat{S}_N} \inf_{y\in S} \|x-y\|_2 \xrightarrow{\text{a.s.}} 0$.*

$\hat{p}_N^*$ is also a downward biased estimator of p^*, although its bias decreases with N [21, Prop. 5.6]. In Sec. 3.4, we empirically investigate the quality of the SAA.

3.2 PARTIAL OPTIMIZATION

DCSP adds a new partial optimization atom to the DCP atom library, allowing modelers to express partial optimizations, *i.e.*, optimizations over (only) a subset of the optimization variables; this atom also forms the basis for specifying two-stage stochastic programs.

We start with the observation that partial optimization is a convex operation (see, *e.g.*, [4, page 87]): *i.e.*, if f is convex in (x,y) and C is a nonempty convex set, then

$$g(x) := \inf_y \{f(x,y) : (x,y) \in C\},$$

is convex in x.

Accordingly, DCSP specifies a new partial optimization atom that takes as input a convex optimization problem and returns (the epigraph form for) another convex atom, which complies with the DCP prescription for specifying atoms — this means that modelers can use partial optimizations in stochastic programs as they would other atoms. In particular, two-stage stochastic programs, *i.e.*, (2), can be naturally expressed using this atom; furthermore, the second stage optimization problem Q need not be in standard form (as required by other frameworks).

3.3 CHANCE CONSTRAINTS

DCSP computes conservative approximations to chance constraints, as they are typically nonconvex[5]; in particular, DCSP replaces

$$\mathbf{Prob}\,(f(x,\omega) \geq 0) \leq 1-\eta, \tag{4}$$

with a convex upper bound derived as follows [3]. Suppose $\phi : \mathbf{R} \to \mathbf{R}_+$ is a nonnegative, increasing convex function with $\phi(0) = 1$; then $\phi(z) \geq 1(z \geq 0)$, where $1(z \geq 0)$ equals 1 if $z \geq 0$ and 0 otherwise, and so $\phi(z/\alpha) \geq 1(z \geq 0)$, for some variable $\alpha \in \mathbf{R}_{++}$. Thus

$$\mathbf{E}\,\phi(f(x,\omega)/\alpha) \geq \mathbf{Prob}\,(f(x,\omega) \geq 0),$$

and so

$$\begin{aligned} &\alpha\,\mathbf{E}\,\phi(f(x,\omega)/\alpha) \leq \alpha(1-\eta) \\ &\implies \mathbf{Prob}\,(f(x,\omega) \geq 0) \leq 1-\eta, \end{aligned} \tag{5}$$

i.e., (5) is a conservative approximation to (4). Note that (5) is convex in (x,α): it is the perspective of the expectation of a convex increasing function, ϕ, of a convex function, f.[6]

In (5), α can be interpreted as modulating the "steepness" of the approximation; several choices of ϕ are possible, and are analogous, *e.g.*, to different approximations to the zero-one loss common in machine learning. DCSP uses $\phi(z) = \max\{0, z+1\}$, which roughly corresponds to a Markov-inequality type bound[7] on (4), and can also be interpreted as the *conditional value-at-risk* of $f(x,\omega)$ [19]. Prop. 3.2 also tells us that this is the tightest possible choice of ϕ.

Practically speaking, DCSP approximates (5) with its SAA (at which point all the benefits of DCP readily apply), then optimizes over (x,α) to obtain the tightest possible bound; in Sec. 4.3, we empirically investigate the quality of these approximations.

Proposition 3.2 ([14])**.** *Suppose $\phi : \mathbf{R} \to \mathbf{R}_+$ is a nonnegative, increasing convex function and $\phi(z) \geq 1(z \geq 0)$; then $\exists \alpha \in \mathbf{R}_+$ such that $\mathbf{E}\,(f(x,\omega)/\alpha + 1)_+ \leq \mathbf{E}\,\phi(f(x,\omega))$.*

3.4 `cvxstoc`

Next, we briefly detail the syntax of `cvxstoc`; `cvxstoc` builds on `cvxpy`, and thus much of the usage is similar.

Random variables are specified simply in `cvxstoc` as follows:

```
omega = RandomVariableFactory().create_normal_rv(0,1)
```

Here, `omega` is a standard normal random variable. `cvxstoc` includes a `RandomVariableFactory` object to simplify the specification of common random variables; see Sec. 4 for examples of the specification of other random variables.

Expectations are specified by applying the `expectation` atom:

[5] One notable exception is chance constraints involving affine functions of normal random variables, which can be expressed as a second order cone constraint (see, *e.g.*, [4, page 157]). However, we favor the approximate approach described in this section because it is substantially more general, and applies to any class of random variables.

[6] Alternatively, the modeler can fix α, in which case the bound is convex in (x,η) if desired.

[7] Alternatively, one could take $\phi(z) = \exp z$, which would be analogous to a Chernoff-type inequality.

```
result = expectation(exp(omega*x), m)
```

Here, `exp(omega*x)` is the expression we wish to compute the expectation of, and `m` is the number of Monte Carlo samples to use when constructing the SAA.

Partial optimizations are specified by applying the `partial_optimize` atom:

```
atom = partial_optimize(prob, [y], [x])
```

The first argument here is a `Problem`, the second is a list of variables to optimize over, and the third is a list of variables to *not* optimize over.

Two-stage stochastic programs can, in turn, be specified as follows:

```
Q = partial_optimize(prob2, [y], [x])
prob1 = Problem(Minimize(f0 + expectation(Q(x),m)),
                constrs)
```

Here, `prob2` is the second stage problem, `y` is the second stage variable, `x` is the first stage variable, and `prob1` is the first stage problem. Multi-stage stochastic programs can be specified by iterating this construction:

```
Q2 = partial_optimize(prob3, [z], [x,y])
prob2 = Problem(Minimize(phi0 + expectation(Q2(y),m)),
    ↪ constrs2)
Q1 = partial_optimize(prob2, [y], [x])
prob1 = Problem(Minimize(f0 + expectation(Q1(x),m)),
                constrs1)
```

Chance constraints are specified by instantiating the `prob` class and chaining it with an inequality:

```
prob(constr >= 0, m) <= 1-eta
```

Here, `constr` is a (stochastic) constraint, and `m` is the number of Monte Carlo samples to use.

3.4.1 Quality of the sample average approximation

Here, we investigate the quality of the SAA employed by `cvxstoc` in the special case of a (unconstrained) least squares problem

$$\underset{x}{\text{minimize}} \quad \mathbf{E}\,\|Ax-b\|_2^2, \tag{6}$$

where the entries of $A \in \mathbf{R}^{m\times n} \sim \text{Normal}(\mu_1, \sigma_1^2)$, $b \in \mathbf{R}^m \sim \text{Normal}(\mu_2, \sigma_2^2)$, and $x \in \mathbf{R}^n$, in which case the objective has the analytic form

$$x^T\,\mathbf{E}\,A^TAx - 2\,\mathbf{E}\,b^TAx + \mathbf{E}\,b^Tb,$$

assuming we know the second moments of (A, b). Fig. 1 plots the optimal value of the true problem (6) and a SAA to (6): we see that the SAA obtains reasonable accuracy after roughly 100 Monte Carlo samples.

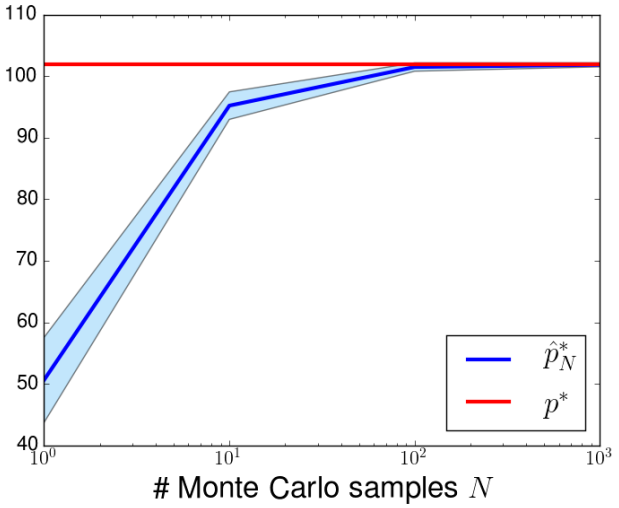

Figure 1: The optimal values of the stochastic least squares problem (6) (red) and a SAA to (6) (blue) with 95% confidence intervals (light blue) vs. the number of Monte Carlo samples; in this case, $m = 100, n = 50$, although similar results hold across a variety of problem sizes.

4 EXAMPLES

At this point, we switch gears slightly and present several examples of stochastic programs along with their corresponding `cvxstoc` implementations[8]; the majority of these applications are well established or previously known, though we also include some formulations that are novel, to the best of our knowledge (namely, the precise formulation of the stochastic optimal power flow problem in Sec. 4.4, and the budgeted learning of a classifier in a cascade problem in Sec. 4.6).

4.1 YIELD-CONSTRAINED COST MINIMIZATION

We begin with a simple example from the operations research literature (see, *e.g.*, [4, page 107]). Consider the (general) problem of choosing the parameters $x \in \mathbf{R}^n$ governing a manufacturing process so that our cost c^Tx, where $c \in \mathbf{R}^n$, is minimized, while the parameters lie in a set of allowable values S; we can model noise in the manufacturing process by expressing this constraint as $\mathbf{Prob}(x+\omega \in S) \geq \eta$, where $\omega \in \mathbf{R}^n$ is a random vector and η is a large probability (*e.g.*, 0.95), which is referred to as an *η-yield constraint*. Thus, we have the optimization problem

$$\begin{array}{ll}\underset{x}{\text{minimize}} & c^Tx \\ \text{subject to} & \mathbf{Prob}(x+\omega \in S) \geq \eta.\end{array}$$

[8] Due to space constraints, we present one of these examples in the supplementary material.

Note that if the distribution over ω is log-concave and S is a convex set, then this constraint is convex in x. We can directly express the yield-constrained cost minimization problem using `cvxstoc`; an implementation is given in Listing 1 (S is taken to be an ellipsoid).

```
# Create problem data
n = 10
c = numpy.random.randn(n)
P, q, r = numpy.eye(n), numpy.random.randn(n), numpy.
    ↪ random.randn()
mu, Sigma = numpy.zeros(n), 0.1*numpy.eye(n)
omega = RandomVariableFactory().create_normal_rv(mu,
    ↪ Sigma)
m, eta = 100, 0.95

# Create and solve optimization problem
x = Variable(n)
yield_constr = prob(quad_form(x+omega,P)
                + (x+omega).T*q + r >= 0, m) <= 1-eta
p = Problem(Minimize(x.T*c), [yield_constr])
p.solve()
```

Listing 1: A `cvxstoc` implementation of the yield-constrained cost minimization problem.

4.2 THE NEWS VENDOR PROBLEM

The news vendor problem is a classic problem in the stochastic programming literature (see, *e.g.*, [2, page 15]); in this problem, a vendor must decide how much newspaper to stock, so that profit is maximized while backorder and return fees (due to excess or insufficient demand, respectively) are minimized, in the face of uncertain demand.

Our optimization variables are the number of units of stocked newspaper $x \in \mathbf{R}_+$, the number of units purchased by customers $y_1 \in \mathbf{R}_+$, and the number of unpurchased (surplus) units that must be returned by the vendor $y_2 \in \mathbf{R}_+$. Our problem data are $b, s, r \in \mathbf{R}_+$, which denote the price to stock, sell, and return a unit of newspaper, respectively. Lastly, we let the random variable $d \sim \text{Categorical}$ model the uncertain (newspaper) demand.

We can pose the news vendor problem as the following two-stage stochastic program:

$$\begin{array}{ll} \underset{x}{\text{minimize}} & bx + \mathbf{E}\, Q(x) \\ \text{subject to} & 0 \leq x \leq u, \end{array}$$

$$\text{where } Q(x) = \begin{array}{ll} \underset{y_1, y_2}{\min} & -(sy_1 + ry_2) \\ \text{s.t.} & y_1 + y_2 \leq x \\ & 0 \leq y_1 \leq d \\ & y_2 \geq 0. \end{array}$$

A `cvxstoc` implementation of the news vendor problem is given in Listing 2; in contrast, a PySP [26] implementation (see the supplementary material) required 111 lines spanning 6 files.

```
# Create problem data
b, s, r, u = 10, 25, 5, 150
d_probs = [0.3, 0.6, 0.1]
d_vals = [55, 139, 141]
d = RandomVariableFactory().create_categorical_rv(
    ↪ d_vals, d_probs)

# Create optimization variables
x = NonNegative()
y1, y2 = NonNegative(), NonNegative()

# Create second stage problem
obj = -s*y1 - r*y2
constrs = [y1+y2<=x, y1<=d]
p2 = Problem(Minimize(obj), constrs)
Q = partial_optimize(p2, [y1, y2], [x])

# Create and solve first stage problem
p1 = Problem(Minimize(b*x + expectation(Q(x), want_de=
    ↪ True)), [x<=u])
p1.solve()
```

Listing 2: A `cvxstoc` implementation of the news vendor problem.

We can also represent a stochastic program by means of an *influence diagram*, a directed acyclic graph, where circular nodes correspond to random variables, square nodes correspond to decision variables, diamond nodes correspond to costs, and edges flow from node x to node y iff the value of node y depends in some way on the value of node x; Fig. 2 presents the influence diagram for the news vendor problem.

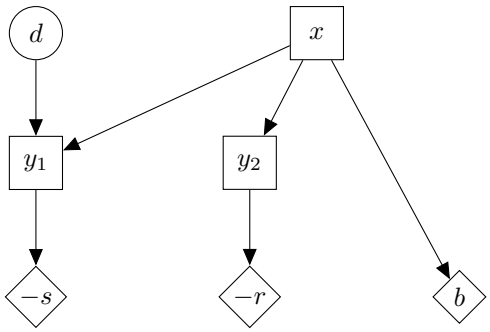

Figure 2: The influence diagram for the news vendor problem.

4.3 PORTFOLIO OPTIMIZATION

In portfolio optimization, we wish to maximize wealth while meeting certain restrictions on risk, in the face of uncertain asset prices; we can pose a standard portfolio optimization problem [12], subject to two kinds of risk constraints, as a stochastic program.

The risk constraints we consider here are the *value-at-risk* (VaR) (see, *e.g.*, [25, chap. 29]) and *conditional value-at-risk* (CVaR) (*e.g.*, [19], [23, page 286]); intuitively, VaR allows the modeler to control the probability of a loss (on asset sales) beyond a (modeler-defined) threshold, and is often nonconvex, while CVaR allows the modeler to control the expected value of such a loss, and is convex.

Our optimization variables are the allocation vector (across a set of n assets) $x \in \mathbf{R}^n$, and the CVaR $\beta \in \mathbf{R}$; the prob-

lem data is the loss threshold $u \in \mathbf{R}_+$, and the vector of returns $p \sim \text{Normal}(\bar{p}, \Sigma)$.

We can pose a CVaR-constrained portfolio optimization problem as

$$\begin{array}{ll} \underset{x,\beta}{\text{minimize}} & \mathbf{E}\, -p^T x \\ \text{subject to} & \beta + 1/(1-\eta)\, \mathbf{E}(-p^T x - \beta)_+ \leq u \\ & \mathbf{1}^T x = 1, \quad x \succeq 0, \end{array} \tag{7}$$

where $(z)_+ := \max\{0, z\}$.

A `cvxstoc` implementation of the CVaR-constrained portfolio optimization problem is given in Listing 3.

```
# Create problem data
n = 10
pbar, Sigma = numpy.random.randn(n), numpy.eye(n)
p = RandomVariableFactory().create_normal_rv(pbar,
    ↪ Sigma)
u, eta, m = numpy.random.rand(), 0.95, 100

# Create optimization variables
x, beta = NonNegative(n), Variable()

# Create and solve optimization problem
cvar = expectation(pos(-x.T*p - beta), m)
cvar = beta + 1/(1-eta)*cvar
prob = Problem(Minimize(expectation(-x.T*p,m)),
               [x.T*numpy.ones((n,1)) == 1, cvar<=u])
prob.solve()
```

Listing 3: A `cvxstoc` implementation of the CVaR-constrained portfolio optimization problem.

We can also pose a VaR-constrained portfolio optimization problem as

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \mathbf{E}\, -p^T x \\ \text{subject to} & \mathbf{Prob}(p^T x \leq 0) \leq 1 - \eta \\ & \mathbf{1}^T x = 1, \quad x \succeq 0. \end{array} \tag{8}$$

As per Sec. 3.3, DCSP replaces the chance constraint in (8) with a sample average approximation (SAA) to a (more conservative) CVaR constraint (making (8) equivalent to (7)). We investigate the quality of this approximation in the special case where $p \sim \text{Normal}(\bar{p}, \Sigma)$, in which case both the VaR and CVaR constraints have analytic forms [18]: the VaR constraint can be expressed as

$$\bar{p}^T x \geq \Phi^{-1}(\eta) \|\Sigma^{1/2} x\|_2, \tag{9}$$

where $\Phi^{-1}(\cdot)$ is the inverse standard normal cumulative distribution function, while the CVaR constraint can be expressed as

$$\bar{p}^T x \geq \exp\left(-(\Phi^{-1}(\eta))^2/2\right) / \left(\sqrt{2\pi}(1-\eta)\|\Sigma^{1/2} x\|_2\right). \tag{10}$$

Fig. 3 plots the optimal value (*i.e.*, wealth) of the VaR-constrained portfolio optimization problem (8), the CVaR-constrained portfolio optimization problem (7), and a SAA to (7): we see that the wealth obtained by constraining VaR is indeed less conservative than by constraining CVaR. The SAA also obtains reasonable accuracy after roughly 100 Monte Carlo samples. Fig. 4 plots the probability of a SAA to (7) vs. the number of Monte Carlo samples, and has a similar interpretation.

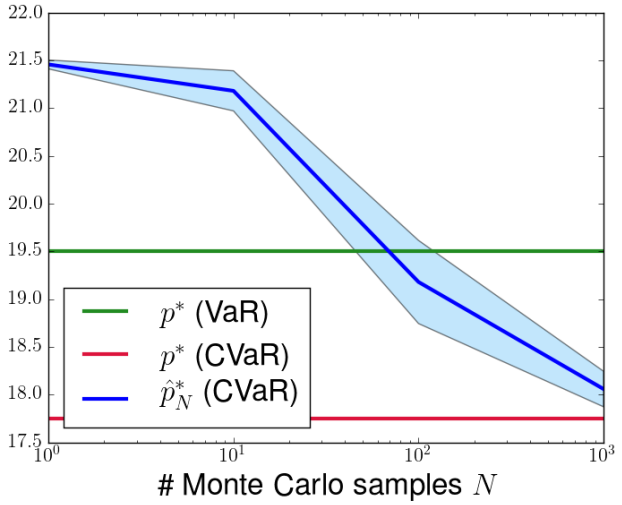

Figure 3: The optimal values (higher means more wealth) of the VaR-constrained portfolio optimization problem (8) (green), the CVaR-constrained portfolio optimization problem (7) (red), and a SAA to (7) (blue) with 95% confidence intervals (light blue) vs. the number of Monte Carlo samples; the problem size $n = 50$, although similar results hold across a variety of problem sizes.

4.4 OPTIMAL POWER FLOW

Consider a network $G = (\mathcal{V}, \mathcal{E})$, with a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$, that models an electrical grid: *i.e.*, a subset of the vertices $\mathcal{G} \subseteq \mathcal{V}$ are *generators*, which produce power, the remaining vertices $\mathcal{L} = \mathcal{V} \setminus \mathcal{G}$ are *loads*, which consume power, and an edge is drawn between a generator and a load if and only if there is a (physical) transmission line between them.

In the standard optimal power flow problem, we wish to minimize the total cost of generating power, while satisfying demand, subject to the topology of the network and per-generator capacity constraints. We often do not have complete control over all the generators in the grid, so we denote the subset of generators that we do have control over as $\mathcal{G}_1$, and also define $\mathcal{G}_2 = \mathcal{G} \setminus \mathcal{G}_1$; we also define $G = |\mathcal{G}|$, $G_1 = |\mathcal{G}_1|$, $G_2 = |\mathcal{G}_2|$, and $L = |\mathcal{L}|$. We write the per-generator costs as $c_{\mathcal{G}_1} \in \mathbf{R}^{G_1}$ and $c_{\mathcal{G}_2} \in \mathbf{R}^{G_2}$, and the per-generator lower and upper (respectively) limits as l and $u \in \mathbf{R}^G$.

The topology/demand constraints can be expressed as $Ap_{\text{lin}} = (p_{\mathcal{G}_1}, p_{\mathcal{G}_2}, p_{\mathcal{L}})$, where $A \in \mathbf{R}^{n \times E}$ is the incidence matrix for the (directed) graph G, $p_{\mathcal{G}_1} \in \mathbf{R}^{G_1}$ and $p_{\mathcal{G}_2} \in \mathbf{R}^{G_2}$ are variables denoting (nonnegative) power generation, $p_{\mathcal{L}} \in \mathbf{R}^L$ are constants denoting the (non-positive) power consumption at the loads, and $p_{\text{lin}} \in \mathbf{R}^E$ are vari-

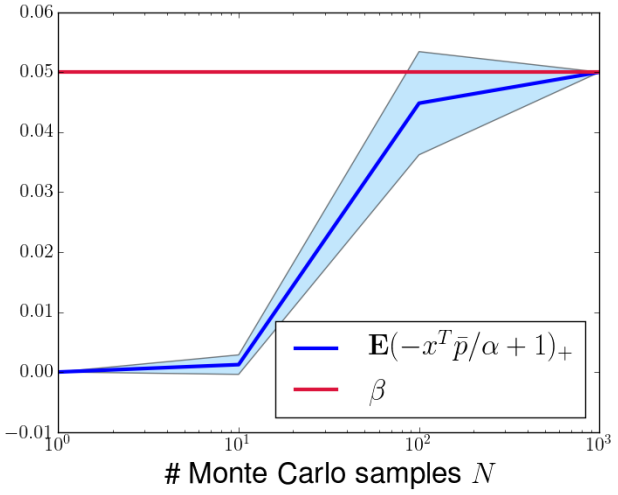

Figure 4: The probability of a SAA to (7) (blue) with 95% confidence intervals (light blue) and β (red) vs. the number of Monte Carlo samples; the problem size $n = 50$, although similar results hold across a variety of problem sizes.

```
# Create optimization variables
p_g1, p_g2 = NonNegative(), NonNegative()
z = NonNegative(num_winds)
p_lines = Variable(E)
p_w = RandomVariable(pymc.Lognormal(name="p_w", mu=1,
                                    tau=1, size=num_winds))

# Create second stage problem
p_g = vstack(p_g1, p_g2)
p = vstack(p_g1,
           p_g2,
           p[load_idxes[:-1]],
           p_w-z,
           p[load_idxes[-1]])
p2 = Problem(Minimize(p_g.T*c_g + z.T*c_w),
             [A*p_lines == p, p_g<=u_gens, z<=p_w,
              abs(p_lines)<=u_lines])
Q = partial_optimize(p2, [p_g2, z, p_lines], [p_g1])

# Create and solve first stage problem
p1 = Problem(Minimize(expectation(Q(p_g1), m)))
p1.solve()
```

Listing 4: A `cvxstoc` implementation of the optimal power flow problem.

ables denoting the power flowing through each edge.

Now, additionally consider the presence of a set of renewable generators (*e.g.*, wind farms), which we denote $\mathcal{W}$, whose (intermittent) generation an operator can either sell on the *spot market* [17], or use to power loads. We let $W = |\mathcal{W}|$, and model this situation with a random vector $p_{\mathcal{W}} \in \mathbf{R}^W$, $(p_{\mathcal{W}})_i \sim \text{LogNormal}(\mu_i, \sigma_i^2)$, $i = 1, \ldots, W$.

We can cast this as the following optimization problem:

$$\underset{p_{\mathcal{G}_1}}{\text{minimize}} \quad \mathbf{E}\, Q(p_{\mathcal{G}_1})$$

where

$$\begin{array}{lll} Q(p_{\mathcal{G}_1}) = & \underset{p_{\mathcal{G}_2}, z, p_{\text{lin}}}{\min} & \begin{bmatrix} c_{\mathcal{G}_1} \\ c_{\mathcal{G}_2} \end{bmatrix}^T \begin{bmatrix} p_{\mathcal{G}_1} \\ p_{\mathcal{G}_2} \end{bmatrix} + c_{\mathcal{W}}^T z \\ & \text{s.t.} & A p_{\text{lin}} = \begin{bmatrix} p_{\mathcal{G}_1} \\ p_{\mathcal{G}_2} \\ p_{\mathcal{L}} \\ p_{\mathcal{W}} - z \end{bmatrix} \\ & & 0 \preceq z \preceq p_{\mathcal{W}} \\ & & |p_{\text{lin}}| \preceq u_{\text{lin}} \\ & & l_{\mathcal{G}} \preceq \begin{bmatrix} p_{\mathcal{G}_1} \\ p_{\mathcal{G}_2} \end{bmatrix} \preceq u_{\mathcal{G}}, \end{array}$$

$c_{\mathcal{W}}$ is the (nonpositive) revenue obtained by selling renewable power, $z \in \mathbf{R}^W$ is the decision vector for the renewable generators, and u_{lin} are the limits on the power flowing through each edge.

A `cvxstoc` implementation of the stochastic optimal power flow problem is given in Listing 4. We solved this problem on the IEEE 14 Bus Test Case, *i.e.*, with $n = 14$, $G_1 = 1$, $G_2 = 1$, $W = 1$, and $L = 10$: Fig. 5 presents the results.

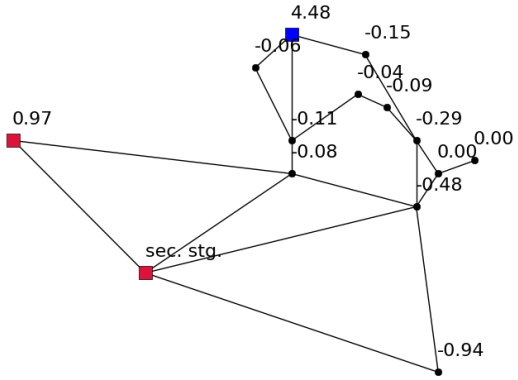

Figure 5: The electrical grid and (optimal) power generation for the optimal power flow problem on the IEEE 14 Bus Test Case. Red vertices are generators: a positive number indicates the optimal power generation, while "sec. stg." denotes an uncontrolled generator. The blue vertex is a (stochastic) renewable generator: its mean available (wind) power is shown above it. Other vertices are loads: their (nonpositive) demanded powers are shown above them.

4.5 ROBUST SUPPORT VECTOR MACHINE

Consider the problem of learning a support vector machine (SVM) from a set of m data points $\{(x_i, y_i)\}_{i=1}^m$. Suppose we would like to (additionally) model the fact that our data collection process is noisy (in order to gain robustness in our solution), by incorporating the belief that (say) $x_i \sim \text{Normal}(\mu_1, \Sigma_1)$ for all i where $y_i = 1$ and $x_i \sim \text{Normal}(\mu_2, \Sigma_2)$ for all i where $y_i = -1$ into the learning process. We can thereby pose the following chance-constrained variant of the canonical (soft-margin)

SVM optimization problem [1]

$$\begin{array}{ll}\underset{w,b,\xi_i}{\text{minimize}} & \|w\|_2^2 + C\sum_{i=1}^m \xi_i \\ \text{subject to} & \mathbf{Prob}\left(y_i(w^T x_i + b) \geq 1 - \xi_i\right) \geq \eta, \\ & \xi_i \geq 0, \quad i = 1, \ldots, m,\end{array}$$

where $w \in \mathbf{R}^n$, $b \in \mathbf{R}$, $\xi_i \in \mathbf{R}_+$ for $i = 1, \ldots, m$, C is the regularization trade-off parameter, and η is a large probability (*e.g.*, 0.95)[9].

A `cvxstoc` implementation of the robust SVM problem is given in Listing 5.

```
w, b, xi = Variable(n), Variable(), NonNegative(m)

constr = []
Sigma = 0.1*numpy.eye(n)
for i in range(m):
    mu = numpy.array(X[i])[0]
    x = RandomVariableFactory().create_normal_rv(mu,
     ↪ Sigma)
    chance = prob(-y[i]*(w.T*x+b) >= (xi[i]-1), ns)
    constr += [chance <= eta]

p = Problem(Minimize(norm(w,2) + C*sum_entries(xi)),
            constr)
p.solve()
```

Listing 5: A `cvxstoc` implementation of the robust SVM problem.

4.6 BUDGETED LEARNING OF A CLASSIFIER IN A CASCADE

Suppose we are interested in learning a (single) classifier that is part of a system (cascade) of classifiers; *i.e.*, we are interested in estimating the parameters $a \in \mathbf{R}^n$ and $b \in \mathbf{R}$ of a first stage classifier, whose output is to be (somehow) combined with the output of a second stage classifier, before presenting the combined output to a user[10].

If we knew the second stage classifier's parameters, then our learning task would be trivial. Instead, we choose to model our uncertainty as follows: we assume that we *do* know the second stage classifier's loss function, but remain uncertain of its feature representation. We can pose this as a two-stage stochastic program, where the expectation in the second stage is taken over all possible feature representations for the second stage classifier; for instance, if the cascade is being used for document classification, then we might posit that each possible feature representation in the second stage is a function of a sample of a word from a generative model (*e.g.*, latent Dirichlet allocation).

We additionally assume that there is some overall test time budget on the cascade, which we express as an upper bound $u \in \mathbf{R}_+$ on the quantity $\|a\|_1 + \|c\|_1$, where $c \in \mathbf{R}^q$ are the parameters of the second stage classifier [27].

[9]We note that this formulation is quite fine-grained, in the sense that per-data point noise models/distributions, as well as mistake probabilities, may be specified.

[10]Such scenarios are common in web search: see, *e.g.*, [27].

Concretely, we can write this optimization problem as

$$\underset{a,b}{\text{minimize}} \quad L_1\left(a, b; \{x_i, y_i\}_{i=1}^m\right) + \mathbf{E}\, Q(a, b),$$

$$\text{where } Q(a,b) = \begin{array}{ll}\underset{c,d}{\min} & L_2\left(c, d; \{z_i, w_i\}_{i=1}^p\right) \\ \text{s.t.} & \|a\|_1 + \|c\|_1 \leq u,\end{array}$$

$\{(x_i, y_i)\}_{i=1}^m$ is the (fixed) training set of m points in $\mathbf{R}^n$ for the first stage classifier, and $\{(z_i, w_i)\}_{i=1}^p$ is the (stochastic) training set of p points in $\mathbf{R}^q$ for the second stage classifier.

A `cvxstoc` implementation, where L_1 and L_2 are (both) taken to be the ℓ_2-regularized log loss, is given in Listing 6.

```
# Create optimization variables
a, b = Variable(n), Variable()
c, d = Variable(q), Variable()

# Create second stage problem
obj2 = [log_sum_exp(vstack(0, -w[i]*(c.T*z[i]+d)))
        for i in range(p)]
budget = norm1(a) + norm1(c)
p2 = Problem(Minimize(sum(obj2) + C*norm(c,2)),
             [budget<=u])
Q = partial_optimize(p2, [c,d], [a,b])

# Create and solve first stage problem
obj1 = [log_sum_exp(vstack(0, -y[i]*(x[i]*a+b)))
        for i in range(m)]
p1 = Problem(Minimize(sum(obj1) + C*norm(a,2) +
                      expectation(Q(a,b), ns)), [])
p1.solve()
```

Listing 6: A `cvxstoc` implementation of the budgeted learning of a classifier in a cascade problem.

5 CONCLUSION

We described disciplined convex stochastic programming (DCSP), a modeling framework that can significantly lower the barrier for modelers to specify and solve convex stochastic programs. We presented a number of sample implementations of stochastic programs that illustrated DCSP's expressivity; in constrast, other frameworks often require significantly more effort from the modeler to express the problem and/or manipulate it into standard form, support a limited number of stochastic programming constructs, and cannot express certain families of convex optimization problems.

Acknowledgements

We thank John Duchi and the reviewers for helpful discussions. This work was supported by a Dept. of Energy Computational Science Graduate Fellowship under grant number DE-FG02-97ER25308, and by the National Science Foundation under grant number IIS-1320402.

References

[1] A. Ben-Tal, S. Bhadra, C. Bhattacharyya, and J. Nath. Chance-constrained uncertain classification via robust optimization. *Mathematical Programming*, 127(1):145–173, 2011.

[2] J. Birge and F. Louveaux. *Introduction to Stochastic Programming*. Springer Series in Operations Research and Financial Engineering. Springer, 1997.

[3] S. Boyd. Chance-constrained optimization. `http://stanford.edu/class/ee364a/lectures/chance_constr.pdf`, January 2015.

[4] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[5] A. Charnes and W. Cooper. Chance-constrained programming. *Management Science*, 6(1):73–79, 1959.

[6] G. Dantzig. Linear programming under uncertainty. *Management Science*, 50(12 Supplement):1764–1769, December 2004.

[7] S. Diamond, E. Chu, and S. Boyd. CVXPY: A Python-embedded modeling language for convex optimization, version 0.2. `http://www.cvxpy.org`, May 2014.

[8] A. Domahidi, E. Chu, and S. Boyd. ECOS: An SOCP solver for embedded systems. In *Proceedings of the European Control Conference*, 2013.

[9] N. Goodman, V. Mansinghka, D. Roy, K. Bonawitz, and J. Tenenbaum. Church: A language for generative models with non-parametric memoization and approximate inference. In *Proceedings of the 24th Conference on Uncertainty in Artificial Intelligence*, 2008.

[10] M. Grant. *Disciplined Convex Programming*. PhD thesis, Stanford University, 2004.

[11] J. Löfberg. YALMIP: A toolbox for modeling and optimization in MATLAB. In *Proceedings of CCA/ISIC/CACSD*, September 2004.

[12] H. Markowitz. Portfolio selection. *The Journal of Finance*, 7(1):77–91, 1952.

[13] T. Minka, J. Winn, J. Guiver, S. Webster, Y. Zaykov, B. Yangel, A. Spengler, and J. Bronskill. Infer.NET 2.6, 2014. Microsoft Research Cambridge. http://research.microsoft.com/infernet.

[14] A. Nemirovski and A. Shapiro. Convex approximations of chance-constrained programs. *SIAM Journal on Optimization*, 17(4):969–996, 2006.

[15] A. Patil, D. Huard, and C. Fonnesbeck. PyMC: Bayesian stochastic modelling in Python. *Journal of Statistical Software*, 35(4):1–81, 7 2010.

[16] A. Pfeffer. Figaro: An object-oriented probabilistic programming language. *Charles River Analytics Technical Report*, page 137, 2009.

[17] D. Phan and S. Ghosh. Two-stage stochastic optimization for optimal power flow under renewable generation uncertainty. *ACM Transactions on Modeling and Computer Simulation*, 24(1):2:1–2:22, January 2014.

[18] A. Prékopa and R. Wets. *Stochastic Programming*, volume 27. North-Holland, 1986.

[19] R. Rockafellar and S. Uryasev. Conditional value-at-risk for general loss distributions. *Journal of Banking and Finance*, pages 1443–1471, 2002.

[20] Richard E Rosenthal. GAMS — A user's guide. 2004.

[21] A. Shapiro, D. Dentcheva, and A. Ruszczynski. *Lectures on Stochastic Programming: Modeling and Theory*. MOS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, 2009.

[22] M. Udell, K. Mohan, D. Zeng, J. Hong, S. Diamond, and S. Boyd. Convex optimization in Julia. In *Proceedings of the 1st First Workshop for High Performance Technical Computing in Dynamic Languages*, 2014.

[23] S. Uryasev and P. Pardalos. *Stochastic Optimization*. Applied Optimization. Springer, 2001.

[24] C. Valente, G. Mitra, M. Sadki, and R. Fourer. Extending algebraic modelling languages for stochastic programming. *INFORMS Journal on Computing*, 21(1):107–122, 2009.

[25] S. Wallace and W. Ziemba. *Applications of Stochastic Programming*. MPS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, 2005.

[26] J. Watson, D. Woodruff, and W. Hart. PySP: Modeling and solving stochastic programs in Python. *Mathematical Programming Computation*, 4(2):109–149, 2012.

[27] Z. Wu, M. Kusner, K. Weinberger, M. Chen, and O. Chapelle. Classifier cascades and trees for minimizing feature evaluation cost. *Journal of Machine Learning Research*, 15:2113–2144, 2014.