Prof. Cristiano Arbex Valle

Instruções para uso do *wrapper* que fiz para resolver problemas de otimização quadrática, linear e inteira

Existem diversos *solvers* comerciais e *open-source* para a resolução de problemas de otimização em geral. Atualmente os (reconhecidamente) mais rápidos são o CPLEX e o Gurobi, ambos proprietários, mas ambos com licenças gratuitas para pessoas do mundo acadêmico.

O GLPK é possivelmente o *solver* open-source mais conhecido, sendo também um dos mais rápidos dentre as alternativas gratuitas. Porém, é ainda consideravelmente mais lento que tanto o CPLEX quanto o Gurobi. Para efeitos da matéria, eu acredito que quebre o galho bem. Eu particularmente gosto também do SCIP, um *solver* alemão que resolve não apenas problemas de otimização linear e inteira mas também resolve problemas não-lineares em geral.

O CPLEX é também capaz de resolver problemas quadráticos como o do Markowitz (assim como o SCIP). O GPLK não resolve. Em `R` existem pacotes que fazem interface com GLPK (`Rglpk`), lpSolve (`lpSolveAPI`), o `quadprog` (que implementa algoritmo próprio para programação quadrática), entre outros.

Você pode escolher qualquer *solver* que quiser para resolver problemas de otimização, mas se não tem nem ideia de como começar, pode utilizar o *wrapper* que escrevi para simplificar a criação de modelos. Este wrapper utiliza o pacote `quadprog` quando o modelo a ser resolvido é quadrático, e utiliza o `Rglpk` quando o modelo é linear ou inteiro.

O *wrapper* está todo localizado no arquivo `model.r`, sendo autocontido. Através de 3 exemplos, explico como utiliza-lo:

Capital budgeting

O problema de capital budgeting é simples. Neste exemplo, você é o dono de uma empresa e existem 4 projetos nos quais pode investir. Os 4 projetos possuem lucro garantido de $(8, 11, 6, 3)$ milhões de dólares respectivamente. Para investir em cada projeto, a empresa deve fazer um aporte inicial de $(5, 7, 4, 3)$ milhões de reais respectivamente. O orçamento disponível para estes aportes é 14 milhões de reais. Em quais projeto a empresa deve investir de forma a maximizar seu lucro?

O modelo de otimização correspondente utiliza variáveis binárias de decisão (x_1, x_2, x_3, x_4). Uma variável binária só pode possuir valores 0 ou 1. Se $x_i = 1$, a empresa decide por investir no projeto i, se $x_i = 0$, ela não investe. Problemas de otimização onde o domínio de valores das variáveis só contém números inteiros (como é o caso aqui) são chamados de problemas de Programação Inteira. A formulação para este problema é dada por:

$$\begin{aligned} \max \quad & 8x_1 + 11x_2 + 6x_3 + 3x_4 \\ \text{sujeito a} \quad & 5x_1 + 7x_2 + 4x_3 + 3x_4 \leq 14 \\ & x_i \in \mathbb{B} \quad i = 1, \dots, 4 \end{aligned}$$

O arquivo de exemplo que implementa e resolve este problema usando o *wrapper* é o arquivo `example_capitalBudgeting.r`. Cada comando está bem comentado e o arquivo em si já contém instruções detalhadas, mas repito alguns desses detalhes aqui. Para começar, assumo que os arquivos do modelo e do wrapper (`model.r`) estão no mesmo diretório. Através dos comandos:

```
script.dir <- dirname(sys.frame(1)$ofile);
setwd(script.dir);
source("model.r");
```

alteramos o diretório corrente no RStudio para o diretório onde o arquivo `.r` do modelo está, e carregamos o *wrapper*. Em seguida temos os comandos:

```
model = initialiseModel();
model$setModelFilename("capitalBudgeting.lp");
model$setSolverDebug(0);
model$setTimeLimit(60);
```

Primeiro inicializamos o modelo. O comando em seguida, `setModelFilename`, exportará o modelo de forma legível para arquivo `.lp` de nome acima. Este arquivo utiliza um formato padrão e pode ser utilizado como ferramenta de debug para verificar se o modelo está sendo criado corretamente. Sugiro abri-lo para verificar após rodar o arquivo do modelo (rodar o arquivo significa carregar o arquivo `example_capitalBudgeting.r` no RStudio e clicar no botão `Source`).

Os comandos seguintes são:

- `setSolverDebug`: os *solvers* imprimem saídas padrão do status de seus algoritmos. Se setar pra zero, o *solver* não imprime nada. A função é opcional e por default o valor é 0.
- `setTimeLimit`: Alguns modelos podem demorar bastante para serem resolvidos na otimalidade. Este comando seta o tempo máximo permitido para o solver tentar resolver o modelo, no exemplo acima este tempo foi de 60 segundos. Se, ao terminar este tempo, o solver ainda não tiver terminado, ele será parado e retornará a melhor solução encontrada até o momento, que pode não ser a ótima. Esta função também é opcional, por default o valor é 120 segundos.

Agora partimos apara a criação do modelo em si. Começamos com:

```
model$setDirection(1);
model$addBinaryVariable("x1", 8);
model$addBinaryVariable("x2", 11);
model$addBinaryVariable("x3", 6);
model$addBinaryVariable("x4", 3);
```

Os comandos acima são:

- `setDirection`: define se o problema é de maximização (1) ou minimização (0). No nosso caso é maximização. Por default, o valor é minimização.
- `addBinaryVariable`: adiciona uma variável binária, os parâmetros são o nome da variável e o valor do coeficiente da mesma na função objetivo. Note que não é necessário passar os limites inferiores e superiores nos valores que a variável pode ter. Como ela é binária, só pode possuir os valores 0 ou 1.

Para adicionar a única restrição do problema, fazemos:

```
elements = c("x1", "x2", "x3", "x4");
values  = c(5, 7, 4, 3);
model$addConstraint("<=", 14, elements, values, "restricao1");
```

A função `addConstraint` recebe 5 parâmetros:

- O sinal da restrição, pode ser "<=", ">=" ou "=".
- O lado direito da restrição, no caso, 14 (o limite de capital disponível para investimento).
- Um vetor contendo os nomes das variáveis cujos coeficientes na restrição são diferentes de zero (todas neste caso). No exemplo, usei o nome `elements` para este vetor.
- Um vetor contendo os coeficientes das variáveis. Usei o nome `values`, e este vetor deve ter o mesmo tamanho de `elements`.
- (Opcional) O nome da restrição, se for vazio ela recebe um nome padrão.

Se quiser passar as variáveis com coeficiente zero, e os valores zero correspondente no vetor `values`, não tem problema, mas é desnecessário.

Resolvemos o modelo com:

```
model$solve();
```

Após a resolução, o arquivo `capitalBudgeting.lp`, especificado acima, será salvo no mesmo diretório. Caso não queira exportar o arquivo, sete `modelFilename = ""` (ou não faça nada, por default é vazio). Podemos então verificar o status e valores das soluções com os comandos:

```
model$solverTime
model$solutionExists
model$status
model$objValue
model$solution
```

Estes comandos trazem as seguintes informações:

- `solverTime`: tempo (em segundos) que o *solver* gastou para tentar resolver o problema.
- `solutionExists`: Variável 1 ou 0 se uma solução foi encontrada ou não. Pode ser que o *solver* termine sem encontrar uma solução. Isto pode acontecer se o modelo for inviável (não possuir solução válida, pode acontecer por exemplo se o modelo for mal construído) ou se o *solver* não conseguir encontrar nenhuma solução no tempo limite estabelecido.
- `status`: Status do solver ao final da execução, os valores possíveis são:
  1. `Optimal solution`: A solução ótima foi encontrada.
  2. `Non-optimal solution`: Existe uma solução, mas não é necessariamente ótima, talvez porque o tempo limite foi atingido.
  3. `No solution exists, the model is infeasible`: O modelo não possui nenhuma solução viável, e o *solver* conseguiu provar isto.
  4. `The time limit was reached before a solution could be found`: Auto-explicativo.
  5. `No solution exists, the model is undefined`: Qualquer outra razão pela qual o *solver* terminou sem achar solução, espero que nunca aconteça.
- `objValue`: Valor da função objetivo na solução encontrada. Só é válido se `solutionExists = 1`.
- `solution`: Vetor com os valores das variáveis na solução ótima. Os valores estão na ordem que as variáveis foram criadas. Só é válido se `solutionExists = 1`.

Outros comandos que podem ser úteis:

```
model$numVariables
model$variables
model$getSolutionValue(model$variables[1]);
```

- `numVariables`: retorna o número de variáveis criados no problema.
- `variables`: vetor com os nomes das variáveis criadas.
- `getSolutionValue`: função que, dado um nome, obtém o valor da variável correspondente na solução. O comando acima seria equivalente a `getSolutionValue("x1")`.

Portfolio que maximiza o pior retorno possível

Agora vamos resolver um modelo pequeno de otimização de portfolios baseado em cenários (distribuição conjunta discreta de retornos, geralmente usamos dados históricos). Vamos resolver o modelo que maximiza a **pior realização**, isto é, queremos escolher o portfolio cujo pior retorno é o máximo possível. Este modelo está criado e detalhado no arquivo `example_linearPortfolio.r`.

O modelo utiliza variáveis de decisão w_i com o peso de cada um dos N ativos considerados. Também precisaremos de uma variável auxiliar M. Vamos considerar também a impossibilidade de *shorting*, ou seja, todo $w_i \geq 0$. Observe, porém, que a variável M é livre, podendo tomar qualquer valor negativo ou positivo. Assumimos também que possuímos T possíveis cenários futuros conhecidos, e que R_{it} é o retorno do ativo i no cenário t. Temos também μ_i como o retorno médio de i e $\overline{\mu}$ como o retorno mínimo desejado. Para mais detalhes sobre cenários e este modelo, veja os slides da matéria a respeito do tema **Cenários e modelos lineares**.

O modelo é dado por:

$$
\begin{aligned}
\max \quad & M \\
\text{sujeito a} \quad & M \leq \sum_{i=1}^{N} R_{it} w_i \quad \forall t = 1, \ldots, T \\
& \sum_{i=1}^{N} w_i \mu_i \geq \overline{\mu} \\
& \sum_{i=1}^{N} w_i = 1 \\
& w_i \geq 0 \quad \forall i = 1, \ldots, N
\end{aligned}
$$

No arquivo `.r`, criei cenários para 5 ativos com 30 retornos aleatórios cada, seguindo distribuições normais com médias e desvios padrão diferentes - temos então que $N = 5$ e $T = 30$.

```
model = initialiseModel();
model$setModelFilename("piorRealizacao.lp");
```

O comando acima indica que um arquivo `.lp` será salvo com o modelo criado, para debug se necessário. Em seguida, fazemos:

```
model$setDirection(1);
model$addVariable("M", 1);
for (i in 1 : N) model$addVariable(paste0("w", i), 0, 0, 1);
```

Com os comandos acima, setamos o problema para maximização (`setDiretion`). As variáveis M e w_i não precisam ser inteiras, podem ter valores contínuos - por isso a função `addVariable`. Esta função recebe 4 parâmetros, dois opcionais:

- Nome da variável.
- Valor do coeficiente correspondente na função objetivo.
- (Opcional) Limite inferior do domínio de valores que a variável pode ter. O valor default é $-\infty$.
- (Opcional) Limite superior do domínio de valores que a variável pode ter. O valor default é $+\infty$.

Observe que a variável M é totalmente livre, e por isso omiti os limites. Para as variáveis w, temos $0 \leq w_i \leq 1$, por isso os últimos parâmetros são 0 e 1. Note que não é necessário criar uma restrição dizendo $w_i \geq 0$ pois isto já é especificado na criação da variável. Note também que como a função objetivo é somente $\max M$, os coeficientes de todos w_i são iguais a 0 (segundo parâmetro na chamada de `addVariable`).

As restrições são criadas com os seguintes comandos:

```
elements = paste0("w", seq(1, N));
values   = rep(1, 5);
model$addConstraint("=", 1, elements, values, "somaDosPesos");

# Restricao de retorno esperado minimo
elements = paste0("w", seq(1, N));
values   = colMeans(scenarios);
```

```
model$addConstraint(">=", minExpectedReturn, elements, values, "retornoEsperadoMinimo");

# Restricoes para limitar o valor de M ao pior cenario possivel
for (t in 1 : T) {
    elements = c("M", paste0("w", seq(1, N)));
    values  = c(1, -scenarios[t, ]);
    model$addConstraint("<=", 0, elements, values, paste0("piorRealizacao", t));
}
```

Observe que na restrição que limita o valor de M, é necessário passar todas as variáveis para o lado esquerdo. Efetivamente ela é escrita assim:

$$M - \sum_{i=1}^{N} R_{it} w_i \leq 0 \qquad \forall t = 1, \ldots, T$$

Ao final resolvemos o problema de forma semelhante ao exemplo acima.

Modelo quadrático de Markowitz

O exemplo que resolve o modelo de Markowitz com desigualdades está localizado no arquivo `example_quadratic.r`. Nele utilizo os dados disponibilizados para os exercícios da matéria. Utilizaremos apenas 5 ativos, os 5 primeiros ativos componentes do índice iBOV de acordo com a ordem que aparecem no arquivo de preços.

No modelo de Markowitz utilizaremos apenas variáveis w_i contendo o peso no ativo i. Seguindo notação utilizada nos slides de **Teoria Moderna de Portfolios**, resolveremos o seguinte modelo:

$$\begin{aligned}
\max \quad & \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} w_i w_j \sigma_{ij} \\
\text{sujeito a} \quad & \sum_{i=1}^{N} w_i \mu_i \geq \overline{\mu} \\
& \sum_{i=1}^{N} w_i = 1 \\
& w_1 \leq 0.3 \\
& w_i \geq 0 \qquad \forall i = 1, \ldots, N
\end{aligned}$$

Observe que não permitimos *shorting* e que o retorno esperado mínimo é $\overline{\mu}$. Além disso, adicionamos uma restrição adicional que diz que o peso no ativo 1, w_1, não pode ser maior que 30%.

Começamos com:

```
model = initialiseModel();
model$setModelFilename("markowitz.lp");
```

O modelo será exportado para o arquivo `markowitz.lp`. Note que, por limitação do pacote `quadprog`, a função objetivo não está sendo no momento exportada (ela estará vazia). Eventualmente implementarei uma função manual para exportar a função objetivo neste caso. No momento, uso uma função de exportação do pacote `lpSolveAPI`. Outra limitação do `quadprog` é que não é possível resolver problemas quadráticos com variáveis inteiras ou binárias.

Em seguida, adicionamos as variáveis:

```
for (i in 1 : N) model$addVariable(paste0("w", i), 0, 0, 1);
model$addQuadraticMatrix(Sigma);
```

Após criar as variáveis w_i, adicionamos a matriz do problema de programação quadrática com a função `addQuadraticMatrix`, que no caso é a matriz de covariâncias Σ. No momento não é possível adicionar variáveis ao problema após adicionar a matriz quadrática.

Para adicionar as restrições:

```
elements = paste0("w", seq(1, N));
model$addConstraint(">=", minExpectedReturn, elements, mu, "retornoEsperadoMinimo");

values  = rep(1, N);
model$addConstraint("=", 1, elements, values, "somaDosPesos");

model$addConstraint("<=", 0.3, "w1", 1, "limitW1");
```

Ao final resolvemos o problema de forma semelhante aos exemplos acima.

Funções adicionais não incluídas nos testes acima

Os seguintes campos e funções adicionais podem ser utilizados caso necessário:

```
model$addIntegerVariable(name, coefficient, low, up);
model$numConstraints
model$useGLPK
```

Abaixo uma descrição:

- `addIntegerVariable`: adiciona uma variável que necessariamente precisa ter valores inteiros, e que não seja necessariamente binária (0 ou 1). Os parâmetros `low` e `up` indicam os valores mínimos e máximos que a variável pode ter. São opcionais e por default são $-\infty$ e ∞.
- `numConstraints`: Número de restrições que o problema possui.
- `useGLPK`: Para problemas lineares ou inteiros, usamos por baixo dos panos o solver GLPK através do pacote `Rglpk`. Opcionalmente você pode utilizar outro solver, lpSolve, através do pacote `lpSolveAPI`. Experiências passadas indicam que no geral a performance do lpSolve é pior que a do GLPK, por isso a opção default é `useGLPK = 1`. Para usar o lpSolve, sete para `useGLPK = 0`.

Futuramente planejo incluir o CPLEX nesta interface, mas não estou conseguindo instalar de jeito nenhum, no Ubuntu, algum dos pacotes `Rcplex` ou `cplexAPI`. Para instala-los, é necessário ter o CPLEX instalado. Eu até tenho, mas tentei de tudo e não consegui fazer a instalação do pacote funcionar. Existe também um tal pacote `ROI` em `R` que teoricamente faz o que eu faço aqui, e faz melhor e mais completo. Eu nunca testei.