

Relatório Científico Final – Iniciação Científica

Processo FAPESP 2016/25959-7

Projeto de algoritmos baseados em florestas de posets
para o problema de otimização U-curve

Beneficiário: Gustavo Estrela de Matos

Responsável: Marcelo da Silva Reis

Relatório referente aos trabalhos desenvolvidos entre 1 de maio e 31
de dezembro de 2017

Laboratório Especial de Toxinologia Aplicada, Instituto Butantan

São Paulo, 9 de Janeiro de 2018

Conteúdo

1	Resumo do Projeto Proposto	2
2	Atividades Realizadas	2
2.1	Estudo de algoritmos baseados em florestas	3
2.2	Modificações do PFS na escolha de raízes	4
2.3	Mudificação do PFS no armazenamento de raízes	6
2.4	Paralelização do PFS	8
2.5	Elaboração do UBB-PFS	8
2.6	Elaboração do Parallel-U-Curve-Search	12
2.7	Testes com instâncias reais do problema de seleção de características	17
3	Avaliação e disseminação de resultados	18
4	Conclusão	18
	Referências	19

1 Resumo do Projeto Proposto

O problema U-curve é uma formulação de um problema de otimização que pode ser utilizado na etapa de seleção de características em Aprendizado de Máquina, com aplicações em desenho de modelos computacionais de sistemas biológicos. Não obstante, soluções propostas até o presente momento para atacar esse problema têm limitações do ponto de vista de consumo de tempo computacional e/ou de memória, o que implica na necessidade do desenvolvimento de novos algoritmos. Nesse sentido, em 2012 foi proposto o algoritmo **Poset-
-Forest-Search** (PFS) [1], que organiza o espaço de busca em florestas de posets. Esse algoritmo foi implementado e testado, com resultados promissores; todavia, novos melhoramentos são necessários para que o PFS se torne uma alternativa competitiva para resolver o problema U-curve. Neste projeto propomos modificações ao PFS na escolha de caminhos de percorrimento da floresta de busca, e na estrutura de dados utilizada para armazenar este grafo, com o uso de diagramas de decisão binária ordenados (OBDDs) [3]; também propomos a criação de uma versão paralela e escalável do algoritmo PFS. Além disso, propomos a criação de um algoritmo baseado no PFS que tenha características de um algoritmo de aproximação, no qual o critério de aproximação da solução ótima se baseie no teorema da navalha de Ockham. Os algoritmos desenvolvidos serão implementados no arcabouço *featsel* [2] e testados com instâncias artificiais e também reais, com conjuntos de dados de aprendizado de máquina retirados do University of California Irvine (UCI) Machine Learning Repository.

2 Atividades Realizadas

Uma implementação do PFS, feita por Reis, está disponível no arcabouço *featsel*. Usamos esta implementação como base para estudar as modificações feitas ao PFS.

2.1 Estudo de algoritmos baseados em florestas

O algoritmo **Poset-Forest-Search** (PFS) é um algoritmo ótimo para resolver o problema de otimização U-Curve e serviu de base para a criação da maioria dos algoritmos elaborados neste trabalho. O PFS é uma generalização de um outro algoritmo mais simples, o **U-curve-Branch-and-Bound** (UBB), que é um algoritmo *branch-and-bound* ótimo que decompõe o espaço de busca em uma árvore, e acha o mínimo global do problema fazendo ramificações e podas nesta árvore.

A árvore de busca do UBB permite que a procura pelo mínimo ocorra de maneira parecida com uma busca em profundidade, que percorre cadeias do reticulado Booleano de subconjuntos menores para maiores. Sempre que o custo de um subconjunto X_i aumenta em comparação ao anterior X_j no percorrimento, a hipótese de que a função de custo é decomponível em curvas em U garante que a subárvore que começa em X_i pode ser removida do espaço de busca; chamamos este procedimento de poda. O algoritmo UBB tem, entretanto, uma limitação, pois quando a função de custo do problema é monótona não-crescente, a condição de poda nunca é verdadeira e o espaço de busca inteiro é visitado, o que compromete a escalabilidade do algoritmo.

O PFS enfrenta esta limitação ao fazer percorrimentos de cadeias do espaço de busca em duas direções, de conjuntos menores para maiores (como faz o UBB) e também o contrário. Para fazer isso, este algoritmo decompõe o espaço de busca em duas árvores, uma para cada direção de percorrimento. Com a criação de duas estruturas para representar o mesmo espaço de busca, torna-se necessário a atualização de uma estrutura sempre que a outra sofrer mudanças, e isto implica na utilização de florestas ao invés de árvores para representar o espaço de busca no PFS. Resumidamente, uma iteração do deste algoritmo deve escolher uma direção de percorrimento; fazer o percorrimento com poda (de maneira similar ao UBB); e, finalmente, atualizar a floresta dual a que foi percorrida.

2.2 Modificações do PFS na escolha de raízes

Para se fazer um percorrimento no espaço de busca, o algoritmo PFS deve escolher uma direção de percorrimento, isto é, uma das duas florestas que representam o espaço de busca, e também uma raiz da floresta escolhida. Esta escolha é feita de maneira arbitrária, e por conta disso, investigamos como diferentes escolhas podem afetar o desempenho do algoritmo.

Na implementação de Reis, a estrutura de dados utilizada para armazenar raízes é a *map* do C++. Escolhe-se nesta implementação o primeiro ou último elemento da estrutura, o que coincide com a primeira ou última raiz quando estas estão ordenadas lexicograficamente por seus vetores característicos. Em nosso trabalho, experimentamos duas modificações para esta escolha:

- de maneira aleatória e uniformemente provável;
- e de maneira determinística, com a raiz de maior sub-árvore completa. Neste caso, o tamanho da sub-árvore completa é o tamanho deste grafo quando nenhuma poda foi feita.

A justificativa para se fazer uma escolha uniforme entre as raízes é ter um algoritmo que não possui viés na escolha de percorrimentos, assim podemos investigar se o viés da escolha arbitrária feita na implementação de Reis compromete a execução do algoritmo. Para fazer a implementação da escolha aleatória e uniforme fizemos uma pequena modificação ao código de Reis. Dado uma floresta $\mathcal{F} = \{r_1, r_2, \dots, r_l\}$ sorteia-se um número pseudo-aleatório a entre 1 e l e escolhe-se a raiz r_a para o percorrimento.

A escolha de raiz com maior sub-árvore foi feita com a intuição de que percorrimentos em árvores maiores implicariam em maiores podas e consequentemente menos nós visitados no espaço de busca. A decomposição do espaço de busca em árvore feito por Reis faz com que ordenar as raízes por tamanho decrescente de sub-árvores completas coincida com uma ordenação lexicográfica da direita para a esquerda dos vetores característicos das raízes. Para fazer isto, basta modificar a ordenação feita pela estrutura *map* do C++.

Tabela 1: Comparação entre os algoritmos *PFS* e *PFS_RANDOM*. O tempo de execução do segundo é maior do que o primeiro enquanto a quantidade de chamadas da função custo é parecida em ambos.

Instância		Tempo de execução médio (s)		Número médio de cálculos de custo	
$ S $	$2^{ S }$	PFS	PFS_RANDOM	PFS	PFS_RANDOM
10	1024	0.013 ± 0.003	0.014 ± 0.003	590.8 ± 198.5	599.5 ± 177.5
11	2048	0.019 ± 0.004	0.022 ± 0.007	1114.8 ± 331.3	1090.1 ± 350.3
12	4096	0.029 ± 0.008	0.036 ± 0.013	1848.6 ± 600.8	1835.7 ± 683.0
13	8192	0.060 ± 0.018	0.090 ± 0.039	4314.4 ± 1496.4	4201.1 ± 1580.7
14	16384	0.100 ± 0.041	0.191 ± 0.110	7323.4 ± 3318.9	7333.8 ± 3161.0
15	32768	0.180 ± 0.076	0.453 ± 0.311	12958.1 ± 5654.0	12807.5 ± 5753.7
16	65536	0.406 ± 0.185	1.715 ± 1.400	27573.8 ± 12459.5	27036.9 ± 12687.5
17	131072	0.717 ± 0.397	5.416 ± 5.266	48176.2 ± 26938.3	47852.1 ± 26427.6
18	262144	1.325 ± 0.754	15.890 ± 17.726	84417.9 ± 48587.7	84025.0 ± 48882.4
19	524288	2.771 ± 1.603	69.600 ± 82.342	167659.1 ± 99686.7	164612.1 ± 102018.3

Chamamos o algoritmo que faz a escolha uniforme das raízes de *PFS_RANDOM*, e os resultados de tempo de execução e número de chamadas da função de custo são apresentados na tabela 1. Podemos observar que esta modificação ao *PFS* não foi benéfica, pois comparando com a implementação de Reis o tempo de execução médio aumentou, e o número médio de chamadas da função de custo continua parecido. O fato do número de chamadas da função de custo não ter diferenças significativas implica que esta escolha de raiz não trouxe mudanças a dinâmica do algoritmo original, logo o tempo a mais de execução veio do código que faz a escolha da raiz. Isto é feito com o percorrimento da estrutura de *map*, o que adiciona tempo linear (sobre a quantidade de raízes) a cada escolha de raiz.

Chamamos de *PFS_LEFTMOST* o algoritmo que faz a escolha da raiz com maior sub-árvore completa, e a tabela 2 mostra resultados de tempo de execução e número de chamadas da função de custo desta variante comparado a implementação de Reis. Podemos observar que esta modificação também não foi benéfica pois, além do maior tempo de execução, o número de chamadas da função de custo aumentou. Isto significa que o comportamento do algoritmo foi o contrário a intuição que motivou a modificação, pois mais nós do espaço de busca foram visitados.

Tabela 2: Comparação entre os algoritmos PFS e PFS_LEFTMOST. O tempo de execução e também o número de chamadas da função custo é maior para o PFS_LEFTMOST.

Instância		Tempo de execução médio (s)		Número médio de cálculos de custo	
$ S $	$2^{ S }$	PFS	PFS_LEFTMOST	PFS	PFS_LEFTMOST
10	1024	0.013 ± 0.002	0.023 ± 0.004	606.1 ± 133.5	665.0 ± 165.8
11	2048	0.020 ± 0.004	0.042 ± 0.010	1122.1 ± 351.2	1316.6 ± 382.2
12	4096	0.032 ± 0.008	0.078 ± 0.024	2183.7 ± 733.2	2515.8 ± 871.3
13	8192	0.054 ± 0.017	0.160 ± 0.061	3887.7 ± 1389.9	4716.8 ± 1777.8
14	16384	0.107 ± 0.034	0.345 ± 0.133	7851.2 ± 2793.0	9506.8 ± 3673.9
15	32768	0.196 ± 0.085	0.672 ± 0.274	13780.3 ± 6049.9	17071.6 ± 7005.1
16	65536	0.348 ± 0.189	1.271 ± 0.661	24106.5 ± 13159.9	30055.6 ± 15363.6
17	131072	0.785 ± 0.361	3.137 ± 1.476	52369.0 ± 24751.2	67585.6 ± 30978.4
18	262144	1.445 ± 0.657	6.146 ± 3.032	92095.9 ± 42566.6	120635.7 ± 58039.0
19	524288	3.298 ± 1.883	13.881 ± 7.595	199151.0 ± 112167.8	256078.6 ± 135958.4

2.3 Mudificação do PFS no armazenamento de raízes

A quantidade de raízes durante a execução do PFS pode ser grande, e por isso a estrutura utilizada para o seu armazenamento deve ser eficiente para operações como inserções, remoções e consultas. Motivados por resultados de um trabalho de iniciação científica anterior (FAPESP processo #2014/23564-0), experimentamos o uso de diagramas de decisão binária ordenados (OBDDs) para o armazenamento de raízes.

A estrutura de OBDD é um diagrama capaz de representar funções Booleanas $f : \{0, 1\}^n \rightarrow \{0, 1\}$. Para utilizarmos esta estrutura no armazenamento de raízes, criamos uma OBDD que representa a função que mapeia para valores 1 os subconjuntos que são raízes da floresta e para 0 os demais. Para de fato armazenar as raízes tivemos que modificar a estrutura usual de OBDD, então ao invés de usar folhas com valores 0 ou 1, usamos folhas que armazenam nós do espaço de busca ou um ponteiro nulo. A figura 1 mostra um exemplo de floresta do espaço de busca e a correspondente representação em nossa estrutura de OBDD.

Como a estrutura de dados que armazena as raízes foi modificada, é necessário definir o abordagem para escolha de raiz na etapa de percorrimto. Utilizamos então a abordagem de escolha aleatória e uniforme, pois esta mostrou uma dinâmica similar a implementação de Reis.

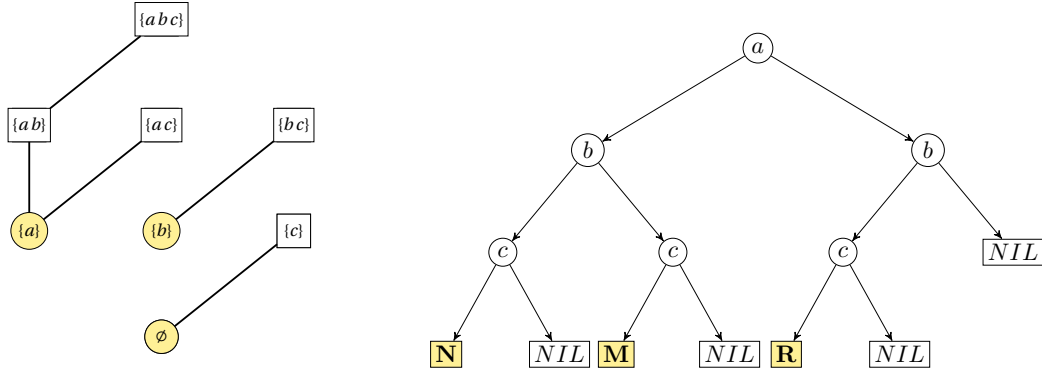


Figura 1: Exemplo de OBDD que representa uma floresta do PFS. Esta OBDD contém os nós **N**, **M** e **R**, que representam respectivamente os subconjuntos 000, 010 e 100. As folhas **NIL** indicam que os subconjuntos de tal caminho na OBDD não são raízes na floresta, como por exemplo os subconjuntos 11X

Tabela 3: Comparação entre os algoritmos PFS e OPFS. O número de chamadas médio da função custo é parecido enquanto o tempo de execução é maior para o OBDD.

Instância		Tempo de execução médio (s)		Número médio de cálculos de custo	
$ S $	$2^{ S }$	PFS	OPFS	PFS	OPFS
10	1024	0.013 ± 0.003	0.018 ± 0.003	598.0 ± 192.8	635.5 ± 171.9
11	2048	0.020 ± 0.004	0.029 ± 0.007	1152.1 ± 314.7	1117.9 ± 336.4
12	4096	0.031 ± 0.010	0.049 ± 0.013	2024.1 ± 751.6	2048.2 ± 700.9
13	8192	0.057 ± 0.017	0.097 ± 0.033	3996.3 ± 1431.6	3973.4 ± 1462.6
14	16384	0.094 ± 0.038	0.171 ± 0.063	6634.8 ± 2944.0	6906.5 ± 2786.5
15	32768	0.182 ± 0.079	0.323 ± 0.156	13140.1 ± 6020.6	12711.2 ± 6319.7
16	65536	0.370 ± 0.169	0.660 ± 0.314	25658.2 ± 11606.7	25303.4 ± 12169.5
17	131072	0.819 ± 0.370	1.480 ± 0.665	53344.9 ± 24350.4	53217.2 ± 24154.5
18	262144	1.515 ± 0.905	2.736 ± 1.626	94677.6 ± 54496.3	94079.4 ± 55435.6
19	524288	2.612 ± 1.869	4.818 ± 3.355	156150.5 ± 107369.8	156021.8 ± 107516.8
20	1048576	6.085 ± 3.900	11.550 ± 7.661	344144.1 ± 212627.1	343229.2 ± 212624.4

Chamamos o algoritmo que usa OBDDs para armazenamento de raízes de OPFS. A tabela 3 mostra uma comparação de tempo de execução e número de cálculos da função de custo entre o algoritmo original e esta modificação. Observamos que o número de chamadas da função custo é similar enquanto o tempo de execução é maior para o algoritmo OPFS, portanto esta modificação não trouxe melhoras ao algoritmo PFS.

2.4 Paralelização do PFS

A decomposição do espaço de busca em uma floresta feita pelo algoritmo PFS separa tal espaço em partes disjuntas que são árvores. Estruturas disjuntas como estas podem ser percorridas de maneira paralela sem ou com poucas interferências entre linhas de processamento, portanto a paralelização do algoritmo PFS pode trazer ganhos em tempo de execução. Desta maneira, paralelizamos este algoritmo, e para isso utilizamos a biblioteca *OpenMP*.

Entretanto, apesar da etapa de percorrimento das árvores ser quase independente entre linhas de processamento paralelas, a etapa de atualização da floresta dual a que foi percorrida é dependente. Além de possuir seções críticas, em que apenas uma linha de processamento pode avançar de cada vez, a atualização da floresta dual pode ter condições de corrida, isto é, o resultado da atualização pode depender da ordem em que as linhas de processamento avançam no código. Condições de corrida não tratadas podem inclusive levar a inconsistências na representação do espaço de busca.

Este algoritmo paralelo foi chamado de PPFS. Testamos o desempenho deste algoritmo comparado ao PFS e os resultados são apresentados na tabela 4. Podemos notar que o tempo de execução do algoritmo paralelo é pior do que a versão original, portanto o trabalho adicional de controle de linhas de processamento não é recompensado pelo ganho de tempo de processamento paralelo. Isto aconteceu porque a etapa de ramificação, que pode ser feita em paralelo sem grandes dificuldades, é curta em relação a etapa de atualização de floresta dual, que possui diversas seções críticas e também um código adicional para controle de condições de corrida.

2.5 Elaboração do UBB-PFS

Visto que a paralelização do PFS não teve bons resultados devido o entrelace de linhas de processamento na atualização da floresta dual, criamos um novo algoritmo paralelo em que as linhas de execução interferem minimamente uma nas outras. Este algoritmo particiona o espaço de busca e resolve cada parte de maneira paralela e independente como um sub-

Tabela 4: Comparação entre os algoritmos PFS e PPFS. O algoritmo PPFS apresenta um número similar de média de chamadas da função custo ao PFS, mas possui tempo de execução médio maior.

Instância		Tempo de execução médio (s)		Número médio de cálculos de custo	
$ S $	$2^{ S }$	PFS	PPFS	PFS	PPFS
10	1024	0.011 ± 0.001	0.146 ± 0.027	643.6 ± 133.0	626.5 ± 151.1
11	2048	0.017 ± 0.004	0.227 ± 0.062	1151.0 ± 359.7	1135.6 ± 381.1
12	4096	0.029 ± 0.007	0.385 ± 0.113	2173.1 ± 652.5	2139.2 ± 710.3
13	8192	0.049 ± 0.015	0.640 ± 0.237	3839.8 ± 1376.6	3743.5 ± 1532.9
14	16384	0.104 ± 0.037	1.337 ± 0.513	8175.9 ± 3037.4	8026.4 ± 3303.7
15	32768	0.163 ± 0.078	2.010 ± 1.096	12459.6 ± 6164.5	12062.2 ± 6897.2
16	65536	0.360 ± 0.163	4.483 ± 2.030	27027.3 ± 12397.0	26835.7 ± 12446.9
17	131072	0.664 ± 0.362	8.072 ± 4.370	48001.9 ± 26149.2	48093.4 ± 26233.2
18	262144	1.250 ± 0.690	14.341 ± 8.062	85880.9 ± 47950.8	86050.8 ± 49186.8
19	524288	2.936 ± 1.629	34.639 ± 19.074	198503.0 ± 108116.1	197832.5 ± 110659.6
20	1048576	5.024 ± 3.097	61.038 ± 38.250	321495.8 ± 198004.3	318507.0 ± 199354.3

problema do problema original. Chamamos este algoritmo de UBB-PFS porque ele possui duas etapas em que procedimentos similares ao UBB e o PFS ocorrem.

Na primeira etapa, percorre-se o reticulado de maneira idêntica a que se faz no algoritmo UBB, ou seja, com uma busca em profundidade. A cada iteração deste percorrimto, a pilha de busca em profundidade contém nós que são raízes de sub-árvores completas do espaço de busca. Portanto, a cada iteração da primeira etapa, o espaço de busca está particionado nas seguintes partes:

- Subconjuntos visitados na primeira etapa: estes subconjuntos entraram na pilha de busca de profundidade, foram visitados e foram removidos da pilha. O custo de todos estes subconjuntos foram calculados.
- Subconjuntos não visitados na primeira etapa: estes elementos estão contidos em uma das sub-árvores com raiz na pilha de busca em profundidade. Note este conjunto é particionado por estas sub-árvores.

Desta maneira, a cada iteração da primeira etapa, podemos encontrar a solução do problema ao encontrar a melhor solução entre o mínimo dos elementos visitados e o mínimo de cada uma das partes que são sub-árvores. Determinar o mínimo dos elementos visitados é fácil, pois o custo destes elementos já foi calculado no percorrimto, já para determinar o

mínimo dos elementos não visitados é necessário resolver sub-problemas do problema original, o que é feito na segunda etapa do UBB-PFS.

A decomposição do espaço de busca em uma árvore feita pelo algoritmo UBB permite que cada sub-árvore do espaço de busca seja representada também por um reticulado Booleano completo. Desta maneira, é possível transformar o problema de encontrar o mínimo local de uma sub-árvore em um problema de seleção de características. A segunda etapa do UBB-PFS é então responsável por mapear cada uma das sub-árvores não percorridas do espaço de busca em um problema auxiliar de seleção de características e resolvê-los em paralelo com chamadas do algoritmo PFS. Como estes sub-problemas são independentes, as chamadas em paralelo devem ter pouco ou nenhum entrelace, o que deve trazer a melhora de desempenho que não foi possível com a simples paralelização do PFS.

Entretanto, ainda é necessário definir em que momento o algoritmo deve passar da primeira para a segunda etapa. A princípio, esta transição pode ocorrer em qualquer fim de iteração da busca em profundidade, entretanto fazer esta transição prematuramente pode implicar em um particionamento grosso do espaço de busca, com poucas raízes na pilha de busca em profundidade; por outro lado, quanto mais tarde fizermos esta transição, menor será o trabalho realizado em paralelo, já que a primeira etapa é feita sempre de maneira serial. Definimos em nossa implementação que a transição deve ocorrer sempre que o número de sub-árvores for maior do que o número de núcleos de processamento do computador ou quando o número de iterações da primeira etapa for maior do que duas vezes a quantidade de características do problema.

A tabela 5 mostra a comparação de tempo de execução entre os algoritmos UBB, PFS e UBB-PFS. Apesar do UBB possuir limitações quando a função é monotônica não-crescente, este é o algoritmo mais rápido no caso médio, seguido pelo UBB-PFS, e o PFS é o mais lento. A tabela 6 mostra a comparação de número de chamadas da função custo, e podemos observar que o PFS tem o menor número, seguido pelo UBB-PFS e o UBB, que possui maior número de chamadas.

Tabela 5: Comparação de tempo médio de execução entre os algoritmos UBB, PFS e UBB-PFS. Podemos observar que o PFS foi o mais lento enquanto o UBB foi o mais rápido e o UBB-PFS teve desempenho intermediário para estas instâncias.

Instância		Tempo de execução médio (s)		
$ S $	$2^{ S }$	UBB	PFS	UBB-PFS
10	1024	0.006 ± 0.001	0.011 ± 0.002	0.023 ± 0.004
11	2048	0.007 ± 0.001	0.017 ± 0.004	0.026 ± 0.004
12	4096	0.010 ± 0.003	0.029 ± 0.009	0.034 ± 0.006
13	8192	0.013 ± 0.006	0.047 ± 0.016	0.044 ± 0.011
14	16384	0.024 ± 0.013	0.094 ± 0.034	0.068 ± 0.023
15	32768	0.043 ± 0.026	0.186 ± 0.074	0.113 ± 0.042
16	65536	0.083 ± 0.060	0.339 ± 0.168	0.187 ± 0.082
17	131072	0.161 ± 0.122	0.650 ± 0.347	0.326 ± 0.175
18	262144	0.321 ± 0.233	1.482 ± 0.768	0.703 ± 0.380
19	524288	0.620 ± 0.447	2.711 ± 1.562	1.309 ± 0.729
20	1048576	1.312 ± 0.970	5.007 ± 3.302	2.478 ± 1.547
21	2097152	2.494 ± 1.893	11.125 ± 6.749	5.458 ± 3.294
22	4194304	4.589 ± 4.122	19.085 ± 15.147	8.832 ± 6.846
23	8388608	12.228 ± 7.922	40.323 ± 29.649	18.891 ± 12.786
24	16777216	24.273 ± 16.277	113.332 ± 76.688	67.178 ± 46.516

Tabela 6: Comparação sobre o número de chamadas de função custo entre os algoritmos UBB, PFS e UBB-PFS. Vemos nesta tabela que o número de nós computados pelo UBB é o maior enquanto o do PFS é o menor; o UBB-PFS tem desempenho intermediário, porém próximo ao do PFS.

Instância		Número médio de cálculos de custo		
$ S $	$2^{ S }$	UBB	PFS	UBB-PFS
10	1024	699.4 ± 361.3	611.3 ± 178.8	634.7 ± 209.8
11	2048	1217.2 ± 747.1	1145.4 ± 358.9	1178.8 ± 484.1
12	4096	2898.0 ± 1380.4	2103.1 ± 793.6	2363.0 ± 760.4
13	8192	4422.6 ± 3293.8	3650.9 ± 1371.9	3934.4 ± 1487.7
14	16384	10089.4 ± 6452.1	7536.3 ± 2926.1	8012.2 ± 3387.4
15	32768	19097.5 ± 12793.8	14546.5 ± 6081.0	15299.2 ± 6598.4
16	65536	37663.1 ± 28321.2	25744.0 ± 12795.4	27028.4 ± 13031.9
17	131072	73373.3 ± 55994.3	46808.9 ± 24533.5	49348.6 ± 24556.7
18	262144	150035.2 ± 108299.3	103166.6 ± 52464.7	105306.4 ± 53472.0
19	524288	292561.2 ± 210771.2	183125.7 ± 104965.4	189545.7 ± 102145.9
20	1048576	617049.5 ± 450468.2	323097.4 ± 213634.3	340694.2 ± 202389.6
21	2097152	1172641.6 ± 879148.5	691991.3 ± 413262.9	704790.2 ± 407143.8
22	4194304	2099973.2 ± 1863285.8	1133395.1 ± 874492.0	1156564.2 ± 862152.0
23	8388608	5435778.8 ± 3468245.3	2276694.5 ± 1621342.2	2345648.2 ± 1558258.5
24	16777216	10146842.9 ± 6673018.3	5527504.2 ± 3413432.3	5609052.7 ± 3337059.1

Estes resultados mostram que o desempenho do UBB-PFS é intermediário entre o UBB e PFS para dois parâmetros diferentes de qualidade. Portanto, podemos considerar o UBB-PFS uma opção competitiva para problemas de seleção de características.

2.6 Elaboração do Parallel-U-Curve-Search

Apesar do algoritmo UBB-PFS ser uma opção competitiva para o problema de seleção de características, este algoritmo possui pontos que poderiam ser explorados para se obter uma melhor paralelização. Entre estes pontos, podemos citar a divisão de trabalho do algoritmo, que é arbitrária e depende da busca em profundidade, o que não é ideal, pois uma boa paralelização deve tentar dividir o trabalho de maneira igual entre os núcleos dos processadores; além disso, a escolha do momento de finalização da primeira etapa para a segunda também é arbitrária; por último, a criação de sub-problemas de seleção de características indica que este algoritmo poderia ser recursivo.

O cronograma da proposta deste trabalho inclui a elaboração de um algoritmo de aproximação para o problema U-curve, porém optamos em continuar estudando algoritmos paralelos para o problema U-curve, o que nos levou a construção do algoritmo **Parallel-U-curve-Search** (PUCS). Ao contrário do UBB-PFS que é uma adaptação paralela de algoritmos já existentes, o PUCS é um algoritmo que foi criado com o intuito de ser paralelo, usando um esquema de divisão e conquista.

Assim como o UBB-PFS, o PUCS particiona o espaço de busca. Para este particionamento acontecer, escolhe-se arbitrariamente um subconjunto S' do conjunto de características S do problema. Chamamos as características de S' de variáveis fixas; de maneira complementar, definimos $\overline{S'} = S \setminus S'$ o conjunto de variáveis livres. Com a relação de equivalência

$$X \sim Y \iff (X \cap S') = (Y \cap S')$$

definimos como o particionamento do espaço de busca as suas classes de equivalência. Em

outras palavras, todos os elementos de uma parte devem conter os mesmos elementos do conjunto de variáveis fixas. Por exemplo, se o conjunto S possui quatro características e as duas primeiras são fixas, então os subconjuntos **0000**, **0001**, **0010** e **0011** formam uma classe de equivalência e uma parte do espaço de busca.

O conjunto de características S' induz o reticulado Booleano $(\mathcal{P}(S'), \subseteq)$, que chamamos de reticulado externo. Cada nó deste reticulado pode representar uma classe de equivalência do particionamento feito. O conjunto $\overline{S'}$ por sua vez define o reticulado Booleano $(\mathcal{P}(\overline{S'}), \subseteq)$ que chamamos de reticulado interno. Podemos representar cada elemento de uma parte por um nó do reticulado interno. Desta maneira, conseguimos representar o espaço de busca inteiro com um reticulado externo em que cada nó está associado a um reticulado interno, assim um nó $Q \in \mathcal{P}(\overline{S'})$ do reticulado interno de uma parte $P \in \mathcal{P}(S')$ corresponde ao subconjunto $Q \cup P$ do espaço de busca original. A figura 2 mostra um exemplo desse particionamento.

Dado o particionamento do espaço de busca e construção do reticulado externo, o algoritmo PUCS descobre o mínimo global fazendo um passeio pelo reticulado externo que deve armazenar partes candidatas a conterem o mínimo global. O conjunto de partes candidatas a conter o mínimo são resolvidas como problemas de seleção de características auxiliares. Seja uma instância $\langle S, c \rangle$ do problema de seleção de características, dado uma parte $P \in \mathcal{P}(S')$, o mínimo desta parte é o mínimo global do problema auxiliar $\langle \overline{S'}, c_P \rangle$, onde $c_P(X) = c(X \cup P)$.

Consideramos partes candidatas a conter o mínimo como aquelas que não são podadas durante o passeio no reticulado externo. Antes de apresentar as condições de poda, definimos como ponta superior de um reticulado $\mathcal{P}(A)$ como o conjunto A , e a ponta inferior como o conjunto \emptyset . Sejam S um conjunto de características e S' um conjunto de variáveis fixas no particionamento definido pelo algoritmo PUCS; dados $P, Q \in \mathcal{P}(S')$ dois elementos do reticulado externo com $Q \subseteq P$, então as condições de poda são:

- se a ponta inferior do reticulado interno de P tem custo maior do que a ponta inferior do reticulado interno de Q , então todas as partes do intervalo $[P, S']$ podem ser podadas, pois possuem reticulados internos cujos elementos terão custo maior do que o custo da

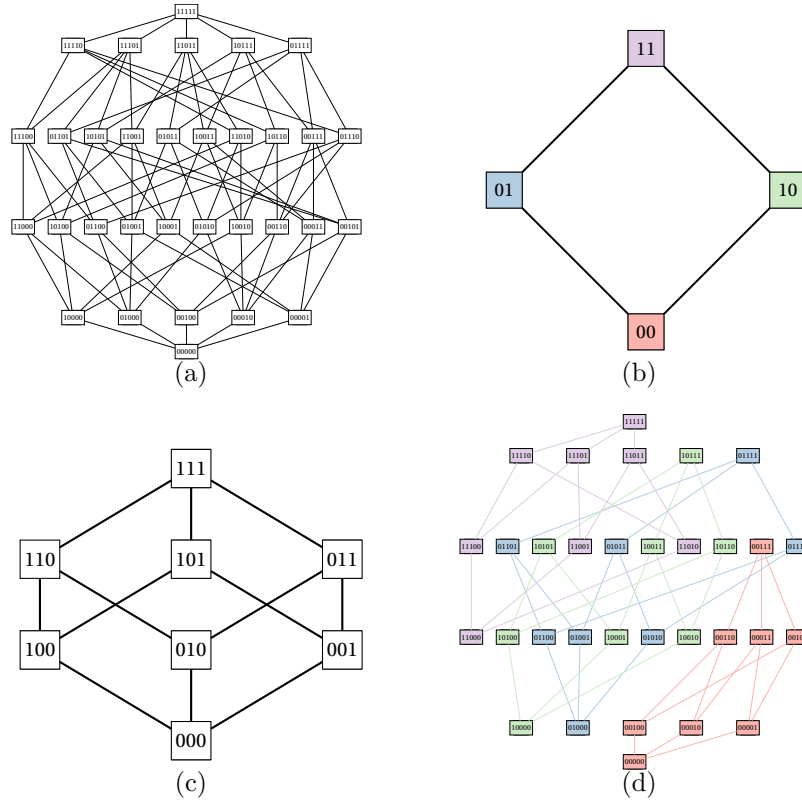


Figura 2: Exemplo de particionamento feito pelo algoritmo *PUCS* em uma instância com cinco características; o reticulado Booleano desta instância é representado na figura 2(a). Neste particionamento, as duas primeiras variáveis formam o conjunto de variáveis fixadas, definindo o reticulado externo (figura 2(b)) enquanto as outras três definem os reticulados internos, que são cópias do reticulado da figura 2(c). A figura 2(d) mostra o reticulado Booleano original, sem as arestas que ligam duas partes diferentes, e a cor de cada nó representa a qual parte tal nó pertence, de acordo com as cores do reticulado externo em 2(b). Note que, de fato, cada parte forma um reticulado pequeno de mesmo tamanho e com mesma estrutura que o reticulado da figura 2(c).

ponta inferior do reticulado interno de Q .

- se a ponta superior do reticulado interno de P tem custo menor do que a ponta superior do reticulado interno de Q , então todas as partes do intervalo $[\emptyset, Q]$ podem ser podadas, pois possuem reticulados internos cujos elementos terão custo maior do que o custo da ponta superior do reticulado interno de P .

Em nossa implementação, a solução de partes candidatas a conterem o mínimo global foi feita de maneira paralela. Note que assim o trabalho é melhor distribuído quando comparado ao UBB-PFS, pois o tamanho das partes do PUCS são iguais. O algoritmo que resolve o

problema auxiliar de achar o mínimo local de uma parte é um dos parâmetros do algoritmo e será discutido em seguida.

O algoritmo **PUCS** possui três parâmetros de funcionamento. Chamamos de algoritmo base o parâmetro que define o algoritmo que será utilizado para se resolver cada parte candidata. O parâmetro l define quantas chamadas recursivas do **PUCS** serão aplicadas nas soluções das partes antes que o algoritmo base seja chamado. Por último, o parâmetro p define a proporção de variáveis do problema que serão fixadas no particionamento.

Observamos em nossos testes que quando o algoritmo base utilizado é uma heurística, então os parâmetros p e l definem o tempo de execução e ao mesmo tempo a qualidade da solução obtida. Quanto maior o valor destes parâmetros, mais próximo a resposta estará da solução ótima, entretanto, maior será o tempo de execução. A figura 3 mostra para instâncias de duzentas características como os parâmetros p e l afetam a qualidade da solução e o tempo de execução do **PUCS** quando o algoritmo base é a heurística **Sequential Forward Floating Selection (SFFS)** [5].

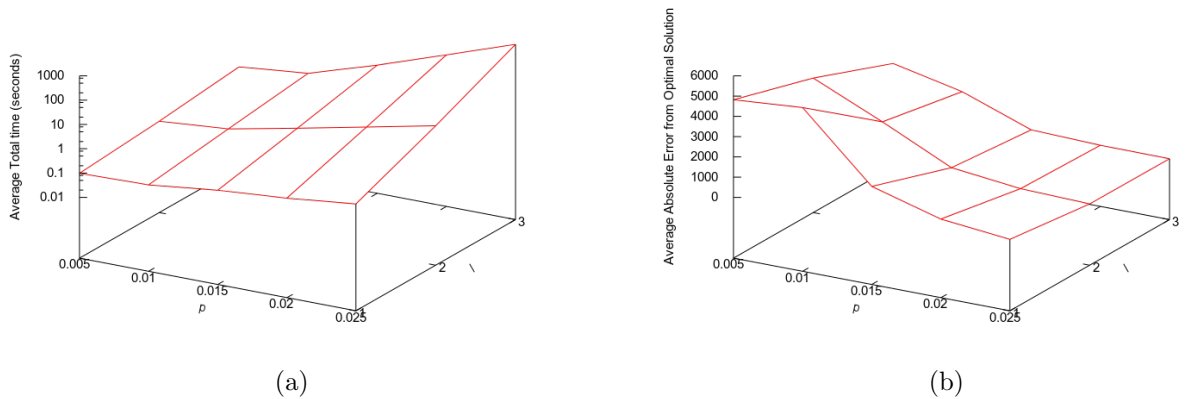


Figura 3: Desempenho do algoritmo **PUCS** para uma instância artificial de 200 características utilizando o **SFFS** como algoritmo base e variando os parâmetros p e l . A figura 3(a) mostra o tempo médio de execução, note que este aumenta quando aumentamos os parâmetros. A figura 3(b) mostra a diferença absoluta média entre o custo do conjunto obtido e do conjunto ótimo.

Separamos os testes do **PUCS** em testes ótimos, em que as instâncias tem tamanho pequeno o suficiente para poderem ser resolvidas otimamente, e em testes sub-ótimos, em que o

Instance		Total time (sec)			
$ S $	$2^{ S }$	UBB	PFS	UBB-PFS	PUCS
10	1024	0.006 ± 0.001	0.011 ± 0.002	0.022 ± 0.004	0.039 ± 0.019
11	2048	0.007 ± 0.002	0.017 ± 0.004	0.026 ± 0.005	0.045 ± 0.022
12	4096	0.009 ± 0.003	0.029 ± 0.009	0.033 ± 0.008	0.048 ± 0.024
13	8192	0.013 ± 0.006	0.054 ± 0.016	0.047 ± 0.014	0.053 ± 0.021
14	16384	0.026 ± 0.012	0.103 ± 0.035	0.074 ± 0.017	0.057 ± 0.024
15	32768	0.048 ± 0.027	0.195 ± 0.080	0.116 ± 0.044	0.056 ± 0.025
16	65536	0.097 ± 0.055	0.354 ± 0.176	0.198 ± 0.089	0.090 ± 0.080
17	131072	0.142 ± 0.120	0.676 ± 0.375	0.350 ± 0.200	0.255 ± 0.276
18	262144	0.319 ± 0.228	1.512 ± 0.764	0.751 ± 0.338	0.680 ± 0.592
19	524288	0.684 ± 0.464	2.875 ± 1.554	1.387 ± 0.707	1.492 ± 1.323
20	1048576	1.249 ± 0.975	5.295 ± 3.509	2.594 ± 1.569	2.701 ± 2.908
21	2097152	2.671 ± 1.948	11.136 ± 6.947	5.460 ± 3.392	6.118 ± 5.961
22	4194304	5.420 ± 4.202	19.825 ± 14.519	9.709 ± 7.319	11.729 ± 11.613

Tabela 7: Comparação de tempo de execução de algoritmos ótimos para o problema U-curve. O PUCS foi o segundo algoritmo mais lento, sendo mais rápido do que o PFS apenas.

tamanho da instância é muito grande para ser resolvida otimamente. Nos testes ótimos, utilizamos como algoritmo base o UBB, enquanto que nos testes sub-ótimos utilizamos a heurística SFFS. Em ambos os testes utilizamos valores de p tal que o número de variáveis fixas é 10, e $l = 1$. Estes valores de p e l foram escolhidos empiricamente como os melhores para a máquina utilizada para os testes.

Os resultados dos experimentos ótimos são apresentados nas tabelas 7 e 8, e nelas podemos observar que o PUCS teve desempenho similar ou pouco pior do que UBB-PFS. Já nos experimentos sub-ótimos, apresentados na figura 4, podemos observar que, apesar do PUCS ter tempo de execução maior do que as outras heurísticas, SFFS e Best-First-Search (BFS), a proporção de vezes em que o PUCS encontrou a melhor solução entre os três algoritmos foi de 100%. Desta forma, podemos considerar o PUCS como uma alternativa competitiva para instâncias grandes do problema U-curve.

Instance		# Calls of cost function			
$ S $	$2^{ S }$	UBB	PFS	UBB-PFS	PUCS
10	1024	668.2 \pm 359.2	614.2 \pm 162.7	630.3 \pm 235.9	753.6 \pm 493.6
11	2048	1171.5 \pm 742.5	1139.2 \pm 354.0	1174.8 \pm 499.3	1305.0 \pm 881.8
12	4096	2611.6 \pm 1528.5	2137.1 \pm 793.3	2271.2 \pm 894.1	2370.5 \pm 1615.7
13	8192	4583.2 \pm 2910.2	4218.9 \pm 1306.7	4308.7 \pm 1732.2	4780.0 \pm 2797.7
14	16384	10781.4 \pm 5565.5	8211.1 \pm 3054.5	9050.4 \pm 2638.1	7050.8 \pm 4755.7
15	32768	20891.9 \pm 12757.0	15134.0 \pm 6528.5	15900.7 \pm 6927.4	14080.1 \pm 11373.1
16	65536	43529.6 \pm 25318.9	26447.0 \pm 13446.1	28783.6 \pm 12934.2	26001.3 \pm 21699.6
17	131072	65301.0 \pm 56215.8	49694.5 \pm 27621.8	51032.5 \pm 29984.3	50145.2 \pm 46799.0
18	262144	145594.5 \pm 103597.8	105603.1 \pm 52652.2	110538.0 \pm 51589.7	111296.6 \pm 84922.4
19	524288	313096.0 \pm 209913.1	194572.5 \pm 104802.3	204604.5 \pm 100305.4	233717.7 \pm 186182.0
20	1048576	578319.0 \pm 445912.2	340052.5 \pm 221271.6	362007.0 \pm 207411.2	387082.0 \pm 389417.4

Tabela 8: Comparação de número de chamadas da função custo em algoritmos ótimos do problema U-curve. O PUCS faz menos chamadas da função custo que o UBB, porém faz mais chamadas do que o PFS e UBB-PFS.

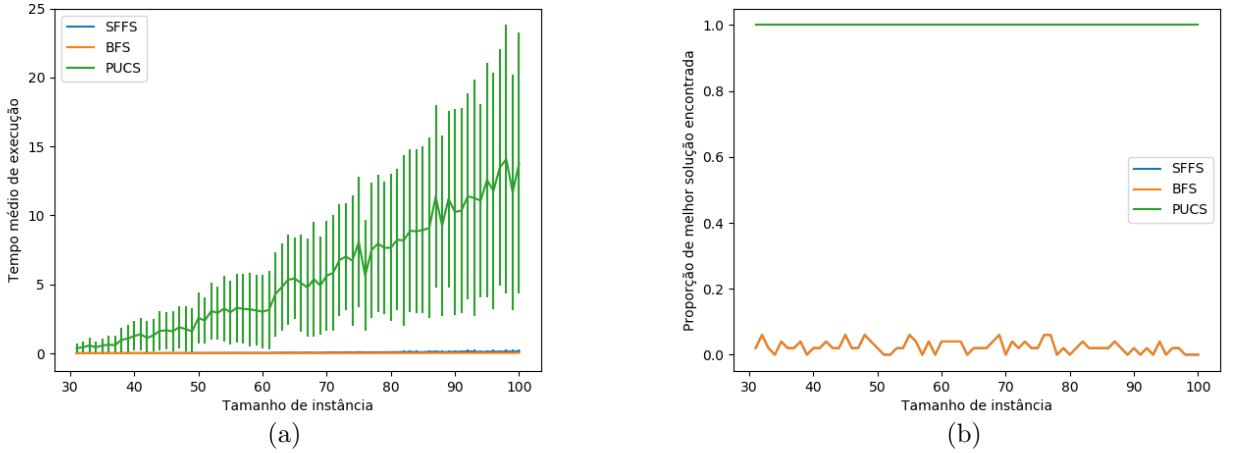


Figura 4: Comparação entre os algoritmos SFBS, BFS e PUCS. A figura 4(a) mostra o tempo médio de execução dos algoritmos. A figura 4(b) mostra a proporção de vezes em que cada algoritmo achou a melhor resposta entre os três.

2.7 Testes com instâncias reais do problema de seleção de características

Como última atividade deste projeto, investigamos o uso de seleção de características na seleção de modelos de aprendizado. Com conjuntos de dados do University of California Irvine (UCI) Machine Learning Repository [6], criamos modelos com e sem seleção de características

e avaliamos os erros médios de seus classificadores. Foi empregado o algoritmo PUCS para esta seleção.

Em ambos modelos (com e sem seleção de características) definimos classificadores do tipo Support Vector Machine (SVM), com kernel linear. Como parâmetro C de regularização, utilizamos o valor 100, que é propositalmente alto, pois queremos observar a regularização vinda da própria seleção de características. O erro de ambos modelos foi estimado com o esquema de validação cruzada; para conjuntos de dados com mais de 100 amostras, utiliza-se validação cruzada 10-*fold*, caso contrário, utiliza-se a validação *leave-one-out*.

A figura 5 mostra os resultados deste teste. Podemos observar que o número médio de características selecionadas é significativamente menor para a maioria destes conjuntos de dados; no conjunto Promoters, por exemplo, menos de 10 características são selecionadas em um total de 57. Esta diminuição implica na diminuição da complexidade do modelo, o que é benéfico em aprendizado de máquina tanto para efeito de tempo de execução quanto para o número de amostras requeridas para treinamento de um classificador que generaliza bem um conjunto de dados. Além disso, a figura 5(b) mostra que mesmo com a simplificação do modelo feita pela seleção de características, a qualidade dos classificadores não é significativamente afetada; inclusive, para o conjunto de dados Lung Cancer, o erro médio dos classificadores diminui quando existe seleção de características.

3 Avaliação e disseminação de resultados

3.1 Participação em conferências

3.2 Monografia de conclusão de curso

3.3 Publicações

4 Conclusão

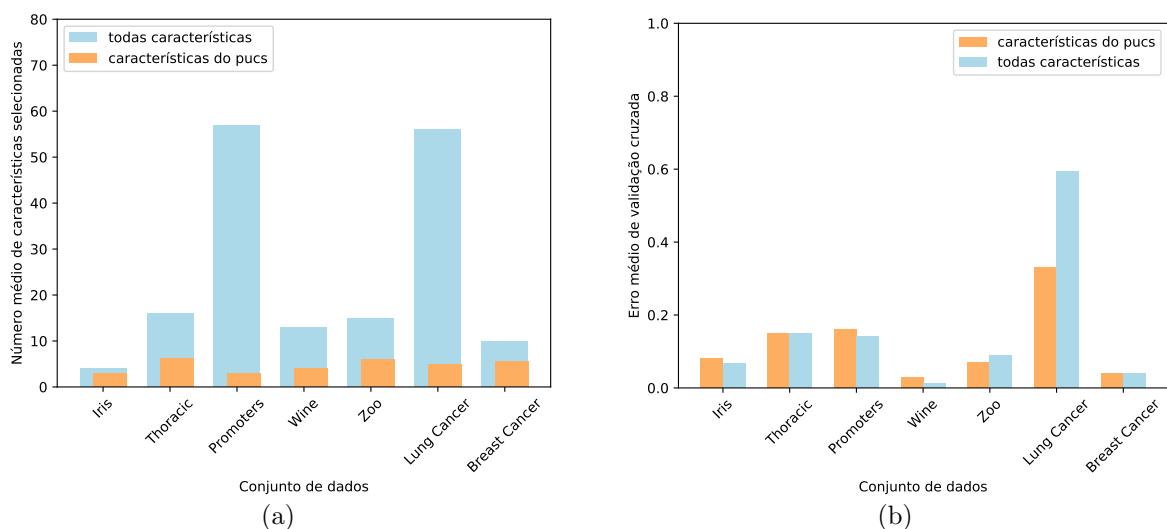


Figura 5: Comparações de modelos feitos com todas características e com a características selecionadas pelo PUCS. A figura 5(a) mostra a quantidade de atributos selecionados pelo algoritmo em cada conjunto de dados, enquanto a figura 5(b) mostra o erro de validação cruzada dos modelos feitos com todas características e apenas com as características selecionadas.

Referências

- [1] Reis, Marcelo S. “Minimization of decomposable in U-shaped curves functions defined on poset chains—algorithms and applications.” PhD thesis, Institute of Mathematics and Statistics, University of São Paulo, Brazil, (2012).
- [2] Reis, Marcelo S., Gustavo Estrela, Carlos E. Ferreira e Junior Barrera. “featsel: A framework for benchmarking of feature selection algorithms and cost functions”. *SoftwareX* 6 (2017), pp. 193-197.
- [3] Bryant, Randal E. “Graph-based algorithms for boolean function manipulation.” *IEEE Transactions on Computers*, 100.8 (1986): 677-691.
- [4] Gustavo E. Matos e Marcelo S. Reis. “Estudos de estruturas de dados eficientes para abordar o problema de otimização U-curve”. Relatório científico final FAPESP, Instituto Butantan, Brasil (2015).

- [5] P. Pudil, J. Novovičová, e J. Kittler. “Floating search methods in feature selection”. Pattern Recognition Letters 15.11 (1994), pp. 1119-1125.
- [6] M. Lichman. UCI Machine Learning Repository. 2013.
URL:<https://archive.ics.uci.edu/ml>.