

TERCEIRO TRABALHO PRÁTICO

HERBERTH AMARAL E MARCELINO MACEDO

Departamento de Ciência da Computação

Universidade Estadual de Montes Claros

Professor Dr. Renato Dourado Maia

3 de novembro de 2015

1 Introdução

2 Objetivos

Os objetivos deste trabalho são:

1. Explorar opções de algoritmos de classificação;
2. Analisar o desempenho desses algoritmos segundo critérios de precisão, tempo de treinamento e de execução.

3 Metodologia

Falar que usamos validação cruzada com 10-fold.

4 Desenvolvimento

O presente trabalho foi desenvolvido na linguagem Python 2 e seu respectivo código (com as devidas instruções de execução) pode ser encontrado no endereço <https://github.com/herberthamaral/mestrado/tree/master/MD/pratica3>.

Ferramentas auxiliares foram utilizadas no desenvolvimento deste trabalho:

1. Numpy;
2. Scikit-learn (colocar referências);

Todos os testes foram executados em um Intel Core i5 de segunda geração (2 processadores, 4 *threads*) com 6GB de RAM.

Os seguintes algoritmos foram avaliados:

1. `sklearn.linear_model.SGDClassifier`;
2. `sklearn.linear_model.Perceptron`;
3. `sklearn.linear_model.PassiveAggressiveClassifier`;
4. `sklearn.lda.LDA`;
5. `sklearn.kernel_ridge.KernelRidge`;
6. `sklearn.svm.SVC`;
7. `sklearn.svm.NuSVC`;
8. `sklearn.svm.LinearSVC`;
9. `sklearn.linear_model.SGDClassifier`;

10. `sklearn.neighbors.RadiusNeighborsClassifier`;
11. `sklearn.neighbors.KNeighborsClassifier`;
12. `sklearn.naive_bayes.GaussianNB`;
13. `sklearn.naive_bayes.MultinomialNB`;
14. `sklearn.naive_bayes.BernoulliNB`;
15. `sklearn.tree.DecisionTreeClassifier`;
16. `sklearn.ensemble.GradientBoostingClassifier`;

4.1 Técnicas de implementação

Aproveitando do fato que as classes que implementam os algoritmos de classificação seguem a mesma interface, implementamos o algoritmo de testes dos classificadores utilizando técnicas de reflexão. Essas técnicas permitiram que o algoritmo de teste ficasse mais generalista e resumido, uma vez que não é necessário implementar um teste específico para cada algoritmo.

Devido ao alto tempo de execução e a alta possibilidade de paralelismo, utilizamos o módulo de multiprocessing do Python para diminuir o tempo de execução dos testes e fazer melhor uso dos recursos computacionais. A quantidade de subprocessos é determinada pela quantidade de processadores disponíveis no ambiente que o algoritmo é executado (4 subprocessos utilizando a máquina de testes descrita anteriormente).

Além da execução paralela, duas pequenas otimizações foram feitas com o intuito de diminuir o tempo de processamento. Duas técnicas foram utilizadas: *lazy load* dos datasets e uma otimização do algoritmo *minmax* em que reduzimos a complexidade de $O(n^2)$ para $O(n)$.

Pelo mesmo motivo de tempo de execução apontado anteriormente, implementamos um mecanismo de retomada da execução do algoritmo de testes: a cada uma das cem iterações salvamos o estado da execução. O algoritmo continua a execução de onde parou caso uma parada aconteça.

4.2 Pré-processamento de dados

Com exceção da base de dados *banknot*, as bases de dados contêm atributos não-numéricos que precisam ser tratados antes. Esses atributos foram substituídos por valores inteiros com o intuito de permitir o uso nos classificadores. Esse pré-processamento pode ser analisado no arquivo *main.py* na função *trata_datasets()*.

Além disso, todos os dados numéricos (exceto as classes) foram normalizados utilizando a técnica **minmax** ($\text{minmax}(X) = \frac{x - \text{Min}(X)}{\text{Max}(X) - \text{Min}(X)}, \forall x \in X$). O minmax normaliza os dados no intervalo $[0, 1]$ e isso é especialmente interessante para os classificadores Bayesianos, os quais não aceitam entradas negativas.

4.3 Execução e pós-processamento

Com 16 algoritmos de classificação, três datasets e validação cruzada 10-fold, cada iteração demora cerca de seis minutos com a configuração detalhada anteriormente. Como executamos cem iterações, a execução total do nosso algoritmo ficou em cerca de dez horas.

A implementação de recursos de paralelismo ajudou a diminuir o tempo de execução, porém, notamos que boa parte do tempo é gasta sintetizando os resultados do teste, o que usa apenas um processador. Fazendo uma analogia com o modelo de programação MapReduce (colocar referência), uma quantidade significativa de tempo foi gasta na fase de redução, porém uma boa fatia de tempo foi economizada na fase de mapeamento.

Salvamos um arquivo com os dados produzidos pelo nosso algoritmo no formato JSON a cada iteração. Esse arquivo contém as seguintes informações:

1. Tamanho do dataset;
2. Tempo (em segundos) usado para treinamento;
3. Tempo (em segundos) usado para validação;
4. Precisão (no intervalo $[0..1]$) do modelo;
5. Quantidade de erros de validação;
6. *Datset* utilizado;
7. Algoritmo de classificação utilizado;
8. Matriz de confusão;

Para facilitar a análise dos dados, desenvolvemos um utilitário para unificar os arquivos JSON e converte-los para CSV (com exceção da matriz de confusão).

5 Resultados

6 Considerações finais

7 Referências

8 Anexos

8.1 Matrizes de confusão