Projeto Integrador V: Learning Vector Quantization

Danilo Mative[1], **Fernando Alves**[1], **Victor Eleuterio**[1]

[1]Departamento de Ciência da Computação – Centro Universitário Senac (SENAC)
São Paulo – SP – Brasil

danilo.amative/eleuteriotrindade@gmail.com, fernandoimp@outlook.com.br

1. Ferramentas

Tendo como foco a aplicação do algoritmo de Learning Vector Quantization, foi imprescindível verificar ao início do projeto o planejamento de construção para cada uma das fases, tanto no que se refere à linguagem e macro utilizada, quanto ao conjunto de dados para teste.

1.1. Python

O desenvolvimento do algoritmo e sua demonstração necessitavam de grandes operações com trocas de valores entre vetores e matrizes, além da leitura e demonstração de dados. Verificando o caso, foi realizada uma pesquisa afim de obter a melhor e mais simples forma de realizar estes tipos de operações, tendo sido escolhida a linguagem Python.
Uma vez que a linguagem já havia sido utilizada no algoritmo de KNN anteriormente, a adaptação se mostrou como muito mais simples, tendo como única diferença algumas funções da versão 2.7 que não haviam sido utilizadas até então.

1.2. Conjunto de Dados Utilizados

Para realização do trabalho, se fez necessária a utilização de seis conjuntos de dados utilizados para Machine Learning, sendo eles: Abalone, Adults, Breast-Cancer-Winsconsin, Iris, Wine e Wine Quality, todos retirados do UCI Machine Learning Repository [A. Asuncion 2007].
Uma vez que os dados apresentavam blocos com texto, nulo ou classes, foi preciso efetuar a normalização dos dados, transformando todos os valores em valores do intervalo [0;1] e classes e texto em números de [1;n], sendo n o número de instancias de texto diferentes.
Além da normalização padrão, definiu-se ainda que todas as bases de dados utilizadas teriam a classe como último valor/coluna, assim facilitando a identificação da instancia no software independente da base utilizada.

2. Versão do Algoritmo LVQ

Uma vez normalizadas todas as bases a serem utilizadas, foi realizada a implementação do algoritmo Learning Vector Quantization, no qual dado um conjunto de dados e uma entrada sem classe definida, organiza e identifica valores contidos no raio R afim de obter a classe do dado inserido[Jason Brownlee 2016].
Foram feitas implementações do algoritmo com diferentes valores para o raio de busca, uma vez que o tipo de classe do dado inserido era definida pela maior quantidade de comparações com um mesmo valor de classe contidos no raio R.
Os valores utilizados para o raio foram: 1, 3, N/2 e N, onde N é a largura (ou altura) da

rede de neurônios.
Os valores para os raios foram testados sequencialmente durante as buscas, de modo a obter todos os resultados no menor número de chamadas para facilitar o cálculo estatístico das fases seguintes.

2.1. Funcionamento do Código

O algoritmo inicia com a chamada da função *lvq*, que recebe como parâmetro os conjuntos de teste e treino, além da taxa de aprendizado, bloco de códigos(valor necessário para definir o tamanho da matriz neural), taxa de decaimento e raio.
A partir deste ponto, é desencadeada uma série de chamadas, por sua vez precedidas pela função *train_codebooks*, que gera a matriz de neurônios necessária para o funcionamento dos passos seguintes do código, tendo como base o bloco de códigos.
A matriz de neurônios é utilizada no momento em que a função *predict* é chamada, realizando o necessário para obter o valor em porcentagem das predições. A obtenção deste valor é dada pela chamada de duas das principais funções do código: *get_matching_units* e *getNeighbor*.

2.1.1. Função get_matching_units e getNeighbor

Para que o algoritmo possa obter as instâncias corretas para associação da entrada teste, se faz necessário ter de forma ordenada uma lista de distâncias para cada instância de treino. Sendo assim, *get_matching_units* realiza chamadas a fim de calcular as distâncias(por meio do método de distância Euclidiana), e com base nos valores obtidos, armazena numa matriz e realiza a ordenação.
Com a ordenação pronta, *getNeighbor* recebe a lista de vizinhos e passa a fazer as verificações necessárias para definir a classe do dado de teste, acrescentando ao peso das variáveis em *sortedVotes* de acordo com a quantidade de instâncias de mesma classe.

2.2. Divisão Euclidiana

Para medir a distância entre os pontos do espaço de características, foi aplicado o cálculo de Divisão Euclidiana, definida por:

$$\mathrm{d}(\mathbf{p}, \mathbf{q}) = \mathrm{d}(\mathbf{q}, \mathbf{p}) = \sqrt{(q_1 - p_1)^2 + (q_2 - p_2)^2 + \cdots + (q_n - p_n)^2}$$

$$= \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}.$$

Onde p e q são pontos de um espaço com n dimensões(em nosso caso, n linhas de instâncias e n linhas de características.

3. Cross-Validation

O Cross-Validation é uma técnica utilizada para estimar a performance de modelos preditivos, sendo assim, estabelece comparações com toda a base de dados a fim de verificar todos os resultados possíveis com valores internos, retornando porcentagens preditivas de

acertos.
Para que a técnica seja aplicada, se faz necessário dividir a base em conjuntos de treino e teste, de modo que os dados de teste são valores testados pelo algoritmo, gerindo respostas corretas ou errôneas, e os dados de treino é um conjunto em que os dados de teste são inseridos como entrada para a verificação.
Neste projeto os dados de teste foram aplicados após as instâncias Q serem divididas em 10, método conhecido pelo nome 10-fold Cross Validation, que consiste em realizar 10 rodadas de teste, sendo cada uma com uma das partes separadas como teste e o restante como treino[Anonymous Author 2007].

3.1. Funcionamento do código

Para realizar o Cross-Validation foram utilizadas as funções *crossValidation* e *k_foldError*, chamadas por *evaluate_algorithm*, configuradas de forma a separar os k-folds e calcular o erro amostral e validação cruzada respectivamente.
A separação dos folds é feita de acordo com a técnica 10-fold Cross Validation, repartindo quaisquer conjuntos enviados desta maneira.

4. Experimentos e Resultados

Após a finalização do algoritmo e execução do Cross-Validation, foram realizados testes para verificar a precisão de cada modelo de vizinhos, sendo estes os testes binários e multi-classe.
Com estes testes, buscou-se compreender quais os desvios de código, ou seja, para onde o erro se encaminhava, passando-nos a noção dos valores mais comuns a causarem a falta de precisão.
Todas as saídas demonstradas abaixo podem ser melhor visualizadas no arquivo results.txt contido na mesma pasta do código.

4.1. Teste Binário

Foca em modelos de dados com duas classes descritivas, onde considera-se uma como positiva e outra como negativa(0 ou 1).
Para cada resultado apresentado, o teste checa e verifica se o valor condiz ou não com a informação correta, demonstrando se a resposta obtida é verdadeira ou falso[Gary Ericson]. Ao final de todas as checagens, se faz possível realizar a criação de uma tabela com valores representativos sobre acertos e falhas para os dois casos.

4.2. Teste multi-classes

O teste multi-classes é aplicado ao caso do número de quantidades das características ser maior do que dois.
Uma vez que o multi-classes passa a ser aplicado, verifica-se em seu todo quais valores foram calculados de maneira correta e incorreta, sendo ideal ao fim desse processo gerar uma matriz de confusão multinível, sendo de NxN e demonstrando, para cada classe característica, a quantidade de acertos e para onde foram os erros(em quais classes caíram).

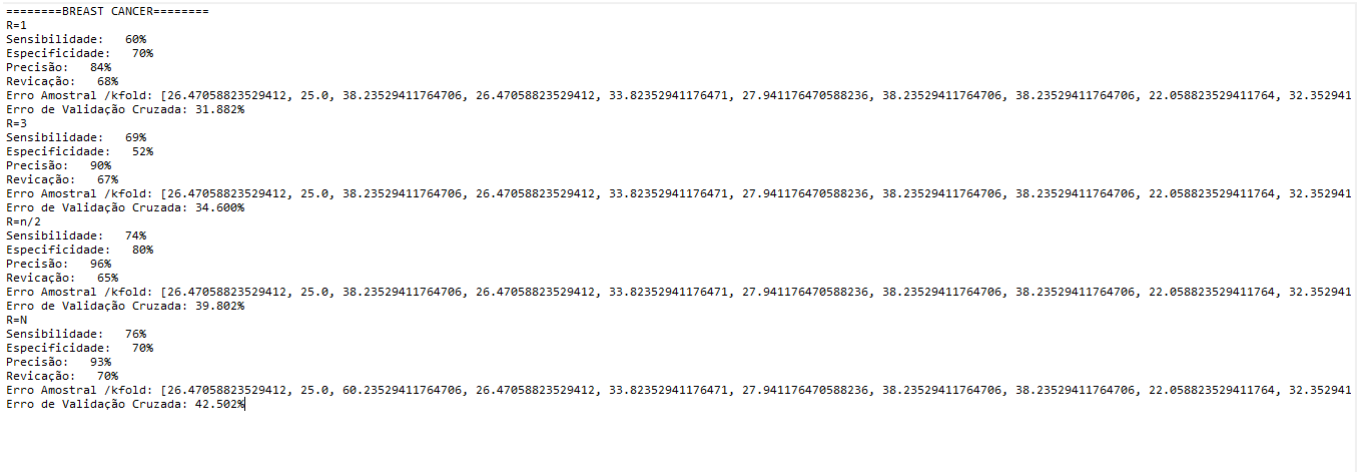

Figura 1. Resultados de saída para um conjunto binário, nesse caso, Breast Cancer Wisconsin

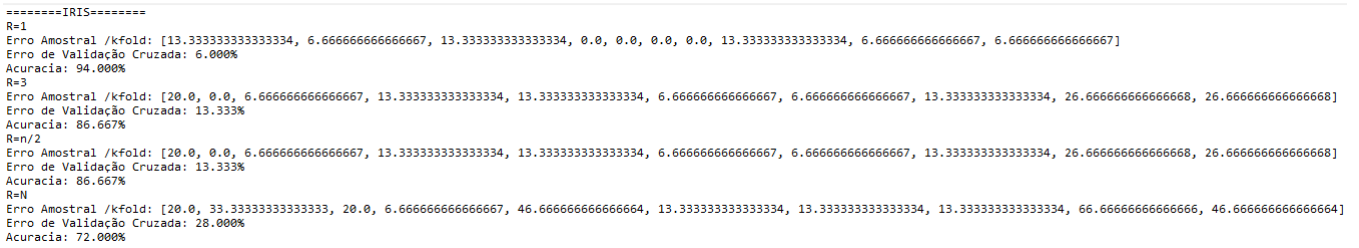

Figura 2. Resultados de saída para um conjunto multi-classe, nesse caso, Iris

4.3. Melhor Raio

De acordo com os testes realizados, definiu-se que o melhor raio utilizado foi 1, dado que raio trabalhava de maneira mais próxima ao neurônio vencedor, sem saturar a comparação com neurônios a mais.
Com raios muito maiores e considerando que o valor estivesse muito próximo às margens da classe, uma grande quantidade de neurônios acaba por obter um resultado errôneo.
Comparando o raio 1 ao M10NN(Melhor valor de vizinhos para o KNN), podemos verificar um valor de precisão muito abaixo. Embora esse valor esteja muito abaixo, pode ser verificado que o tipo de parâmetro é parecido para os dois casos, sendo uma quantidade de 10 vizinhos no KNN e uma comparação com 9 neurônios no LVQ.

5. Comparação com KNN

Notou-se que o LVQ apresentou muito mais velocidade se comparado com o KNN, dado que sua matriz de instâncias não se tratava do conjunto inteiro, e sim de uma parte dos dados cabíveis ao tamanho NxN. Claramente, ao se considerar os 500 testes solicitados, sua execução demorou muito mais.
Por outro lado, por utilizar somente parte das instâncias, o algoritmo se mostrou menos preciso. Quando consideramos que o KNN faz o cálculo de distância com toda a base, temos que os dados para fazer a análise dentro do valor k escolhido são os ideais, o contrário do LVQ, que considera somente um trecho das instâncias aleatório, podendo até mesmo não conter a classe descritiva incluída.

Sendo assim, podemos afirmar que o LVQ preza velocidade e consumo de memória, o que pode ser vantajoso no caso de bases muito grandes, enquanto o KNN preza por precisão.

6. Conclusão

Com base nos resultados obtidos por meio de todos os experimentos, chegamos a conclusão de que o algoritmo Learning Vector Quantization se mostrou incrivelmente eficaz no que diz respeito ao tempo de processamento e utilização de memória, mas com uma precisão levemente menor se comparado com o algoritmo KNN.
Por outro lado, em bases menores e utilizando raio 1, o algoritmo se mostrou próximo ao KNN, uma vez que de acordo com o tamanho da matriz, poucas instâncias eram deixadas de lado e a comparação de distância tomava um valor próximo ao número de instâncias totais da base.
Levando em consideração todos os pontos aqui levantados, podemos afirmar que a utilização do algoritmo deve ser estudada de acordo com a base de teste, prezando a observação da quantidade de instâncias utilizadas e classes dos dados observados para a geração da matriz, além do que o usuário necessita mais, velocidade ou precisão.

Referências

A. Asuncion, D. N. (2007). UCI machine learning repository. Access date: 10 mar. 2018.

Anonymous Author (2007). Cross-validation: evaluating estimator performance. Access date: 12 mar. 2018.

Gary Ericson. Como avaliar o desempenho do modelo no azure machine learning. Access date: 13 mar. 2018.

Jason Brownlee (2016). How to implement learning vector quantization from scratch with python. Access date: 29 mar. 2018.