[PSI5790 aula 5. Início.]

As imagens usadas nesta aula podem ser baixadas em Linux com os comandos:

$ wget -U 'Firefox/50.0' http://www.lps.usp.br/hae/apostila/aprendizagem.zip
$ unzip aprendizagem

A biblioteca de funções *procimagem.h* abaixo deve estar localizada no mesmo diretório do seu programa para compilar programas em OpenCV/C++ puro:

```
//procimagem.h 2024
#include <opencv2/opencv.hpp>
#include <iostream>
#include <float.h>
using namespace std;
using namespace cv;

void erro(string s1="") {
  cerr << s1 << endl;
  exit(1);
}

Mat_<float> filtro2d(Mat_<float> ent, Mat_<float> ker, int borderType=BORDER_DEFAULT)
{ Mat_<float> sai;
  filter2D(ent,sai,-1,ker,Point(-1,-1),0.0,borderType);
  return sai;
}

Mat_<Vec3f> filtro2d(Mat_<Vec3f> ent, Mat_<float> ker, int borderType=BORDER_DEFAULT)
{ Mat_<Vec3f> sai;
  filter2D(ent,sai,-1,ker,Point(-1,-1),0.0,borderType);
  return sai;
}

Mat_<float> dcReject(Mat_<float> a) { // Elimina nivel DC (subtrai media)
  a=a-mean(a)[0];
  return a;
}

Mat_<float> dcReject(Mat_<float> a, float dontcare) {
  // Elimina nivel DC (subtrai media) com dontcare
  Mat_<uchar> naodontcare = (a!=dontcare); Scalar media=mean(a,naodontcare);
  subtract(a,media[0],a,naodontcare);
  Mat_<uchar> simdontcare = (a==dontcare); subtract(a,dontcare,a,simdontcare);
  return a;
}

Mat_<float> somaAbsDois(Mat_<float> a) { // Faz somatoria absoluta da imagem dar dois
  double soma = sum(abs(a))[0]; a /= (soma/2.0);
  return a;
}

Mat_<float> matchTemplateSame(Mat_<float> a, Mat_<float> q, int method,
  float backg=0.0) {
  Mat_<float> p{ a.size(), backg };
  Rect rect{ (q.cols-1)/2, (q.rows-1)/2, a.cols-q.cols+1, a.rows-q.rows+1};
  Mat_<float> roi{ p, rect };
  matchTemplate(a, q, roi, method);
  return p;
}

template<typename T> class ImgXyb: public Mat_<T> {
public:
  using Mat_<T>::Mat_; //inherit constructors

  T backg;
  int lc=0, cc=0;
  int minx=0, maxx=this->cols-1, miny=1-this->rows, maxy=0;

  void centro(int _lc, int _cc) {
    lc=_lc; cc=_cc;
    minx=-cc; maxx=this->cols-cc-1;
    miny=-(this->rows-lc-1); maxy=lc;
  }

  T& operator()(int x, int y) { // modo XY centralizado
    unsigned li=lc-y; unsigned ci=cc+x;
    if (li<unsigned(this->rows) && ci<unsigned(this->cols))
      return (*this).Mat_<T>::operator()(li,ci);
    else return backg;
  }
};
```

As rotinas de aprendizado de máquina sofreram grandes mudanças de OpenCV v2 para OpenCV v3/v4. Vários métodos de aprendizado de máquina que funcionavam corretamente na OpenCV2 não funcionam mais em OpenCV v3/v4, mesmo fazendo as adaptações necessárias. Usando Cekeikon, é possível especificar qual versão de OpenCV quer usar, com as opções -v2 ou -v3.

```
$ compila programa -c -v2  (ou >compila programa -cek -ver2) - default
$ compila programa -c -v3  (ou >compila programa -cek -ver3)
```

As distribuições recentes de Linux trazem OpenCV v4. Para compilar com OpenCV do seu sistema:

```
$ g++ -std=gnu++14 prog.cpp -o prog -fmax-errors=2 `pkg-config opencv4 --libs --cflags` -O3 -s
```

As rotinas de SkLearn parecem ser melhores que as de OpenCV para aprendizado de máquina.

Aprendizado de Máquina para Projetar Filtros

Na aula passada, utilizamos os algoritmos de vizinho mais próximo e árvore de decisão para resolver “problema ABC”, classificar a espécie de flor “Iris” (a partir de comprimento e largura de pétala) e para detectar notas de dólar falsas (a partir de 4 atributos).

Nesta aula, vamos continuar estudando os algoritmos clássicos de aprendizado aplicado num problema de processamento de imagens: projetar filtros espaciais. Os algoritmos que estudaremos são:

- Vizinho mais próximo via força-bruta e sua aceleração usando tabela e kd-tree (FlaNN do OpenCV).
- Árvore de decisão e como melhorar a sua qualidade com Boosting.
- Classificador de Bayes.

1. Aprendizado supervisionado:

Recordando o que já vimos e a convenção adotada, no aprendizado supervisionado um “professor” fornece amostras de treinamento entrada/saída (*ax*, *ay*). O “aprendiz” *M* classifica uma nova instância de entrada *qx* baseado nos exemplos fornecidos pelo professor, gerando a classificação *qp*. Para cada instância de teste *qx* existe uma classificação correta *qy*, desconhecida pelo aprendiz. Se a classificação do aprendiz *qp*=*M*(*qx*) for igual à classificação verdadeira *qy* então o aprendiz acertou a classificação (figura F1).

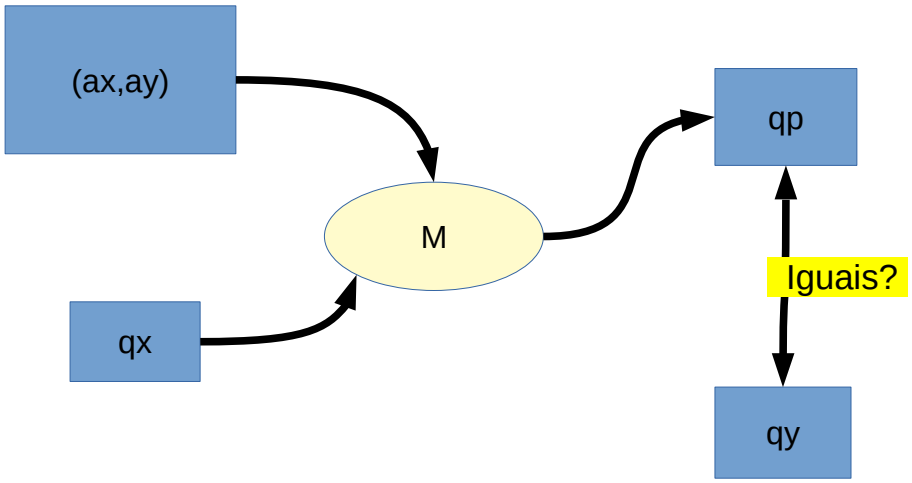


Figura F1: Esquema geral de aprendizado de máquina supervisionada.

2. Projetar filtro 3×3 binário pelo aprendizado de máquina

2.1 Busca do vizinho mais próximo "força bruta"

Vamos projetar filtros espaciais com a aprendizado de máquina. Veja a figura 1 que mostra a detecção de pontas de retas pelo aprendizado de máquina.

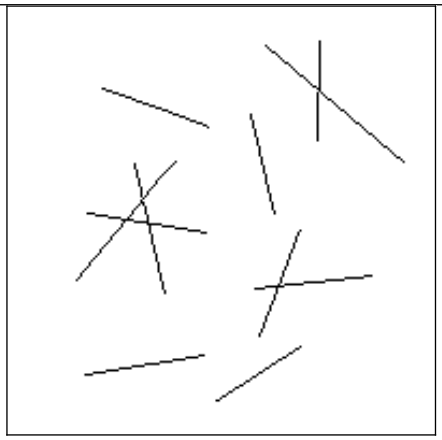
As imagens AX e AY (figuras 1a e 1b) são respectivamente exemplos de entrada e saída, isto é, estamos informando ao aprendiz *M* que, se a entrada do sistema for uma imagem com segmentos de reta como AX (figura 1a), queremos obter a saída AY (figura 1b) onde todas (e somente) as pontas das retas estão marcadas de preto. Depois de alimentar o algoritmo de aprendizado com as imagens-exemplos, fornecemos ao aprendiz *M* a imagem de teste QX (figura 1c) com segmentos de retas. Existe uma imagem de saída ideal QY (figura 1d), mas esta imagem é desconhecida pelo *M*. O aprendiz *M* deve processar QX da melhor forma possível, gerando uma imagem processada QP (figura 1e) tal que esta seja o mais parecido possível da saída ideal desconhecida QY (figura 1d).

Aqui, a medida de desempenho poderia ser: (a) A diferença média absoluta entre QY e QP, isto é, MAE(QY, QP) ou (b) A distância de Hamming entre QY e QP, isto é, o número de pixels diferentes entre as duas imagens.

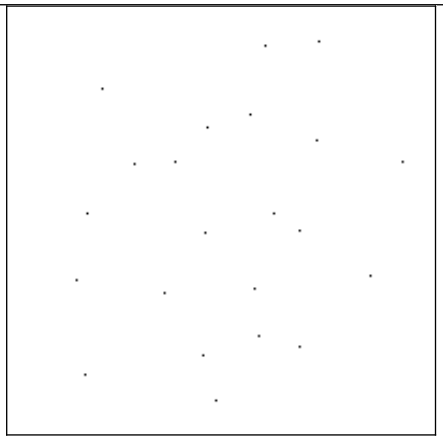
O que devemos fazer aqui? Em vez de pensar como transformar uma imagem de entrada inteira QX numa imagem de saída inteira QP (o que é uma tarefa bem difícil), podemos pensar em como transformar um pedaço da imagem de entrada num pixel de saída. Isto é, como projetar um filtro *F* que usa uma janela 3×3 para descobrir se a cor do pixel de saída correspondente deve ser branca ou preta (uma tarefa bem mais simples, figura 2).

Na figura 2, o filtro *F* olha a vizinhança 3×3 em torno do pixel *p* na imagem de entrada *A* para escolher a cor de saída *B*(*p*). Assim, precisamos projetar somente uma função $f:\{0,1\}^9 \rightarrow \{0,1\}$ que caracteriza o filtro *F* (chamada de função característica). Note que coube a nós, seres humanos, escolher o tamanho da janela adequada para resolver este problema. A janela adequada é a janela de menor tamanho possível que permite predizer corretamente a saída. Se você escolher uma janela pequena demais, não será possível solucionar o problema. Se você escolher uma janela maior do que a necessária, estará tornando o problema mais difícil sem necessidade.

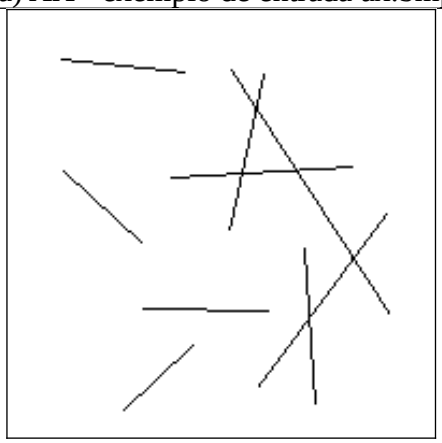
Usando o algoritmo de vizinho mais próximo "força bruta" (isto é, sem nenhum truque para acelerar o seu processamento), dadas as imagens AX, AY e QX, uma janela 3×3 percorre a imagem QX. Para cada pixel *p* de QX, vamos denotar o conteúdo da janela 3×3 de QX centrada em *p* de *qx*(*p*). Para cada *qx*(*p*), devemos procurar um pixel *q* de AX tal que *ax*(*q*) (isto é, o conteúdo da janela 3×3 de AX centrada em *q*) seja o mais semelhante possível à *qx*(*p*). Esta semelhança pode ser medida em número de pixels diferentes (distância de Hamming). Aí, copiamos o valor de saída AY(*q*) para QP(*p*).



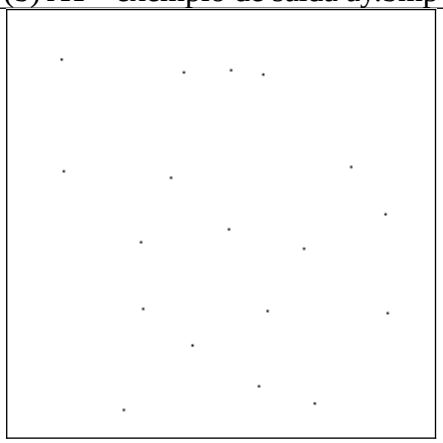
(a) AX - exemplo de entrada ax.bmp



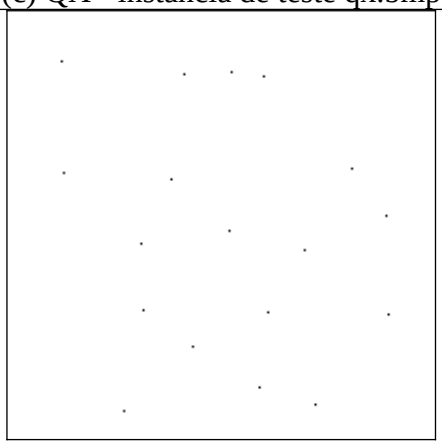
(b) AY - exemplo de saída ay.bmp



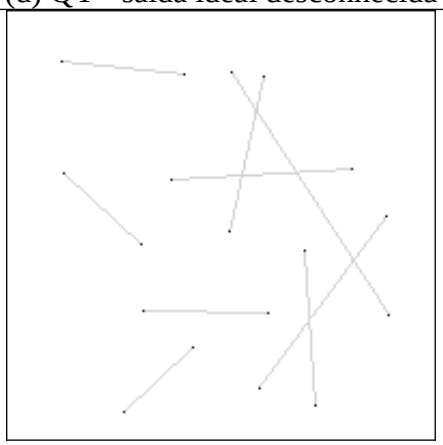
(c) QX - instância de teste qx.bmp



(d) QY - saída ideal desconhecida



(e) QP - saída processada qp.bmp



(f) saída processada sobreposta à imagem de entrada

Figura 1: Detecção de ponta de reta em imagens binárias pelo aprendizado de máquina.

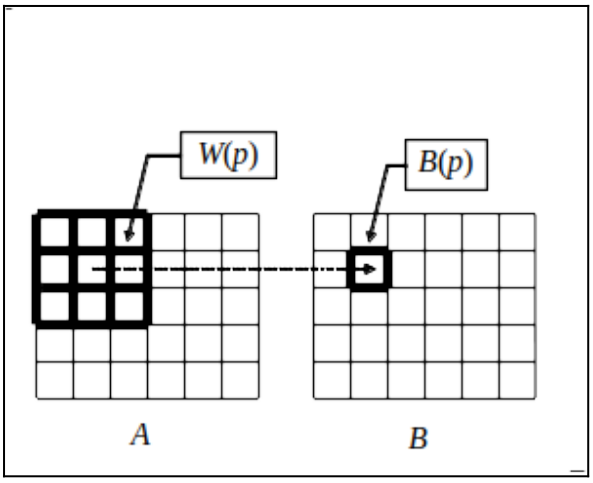


Figura 2: Filtro ou operador restrito à janela 3×3. A transformação é caracterizada pela função característica $f:\{0,1\}^9 \rightarrow \{0,1\}$.

Este algoritmo "força bruta" está implementado no programa nn.cpp (algoritmo 1). As linhas 23-25 lêem as imagens AX, AY e QX. A linha 26 cria a imagem de saída QP e a preenche com pixels brancos. As linhas 27-29 geram índices $p=(l,c)$ que percorrem a imagem QX. Na linha 30, para cada índice $p=(l,c)$, calcula-se o conteúdo da janela $qx(p)$ usando a função "vizinhanca33".

Para cada índice $p=(l,c)$, as linhas 34-36 geram índices $q=(l2,c2)$ que percorrem a imagem AX, procurando o conteúdo da janela $ax(q)$ que seja mais semelhante à $qx(p)$, usando função "hamming". Na linha 38, armazena-se a menor distância de Hamming encontrada "mindist" e a saída AY(q) no pixel q que deu a menor distância "minsai". Este valor de saída é armazenada na imagem de saída QP na linha 40. Por fim, a linha 43 imprime a imagem de saída obtida.

A saída do programa *nn* (figura 1e) é exatamente igual à saída ideal (figura 1d). O único problema é que este processo "força bruta" demora 250s (mais de 4 minutos)[1]! O problema da busca do vizinho mais próximo pela força bruta é o tempo de processamento, pois precisa comparar cada janela de QX com todas as janelas de AX. Se AX tem n pixels, QX tem m pixels e a janela tem l pixels, o algoritmo este algoritmo efetua da ordem de $O(nml)$ operações (chamada de complexidade de tempo). Sem entrar em formalizações, a complexidade tempo de um algoritmo é a ordem do número de operações que o algoritmo executa em função de tamanho dos dados.

Número de operações: 40.000*40.000*9 = 14.400.000.000

Exercício 1: Traduza este programa para Python e verifique o tempo de processamento.

Exercício 2: Você consegue imaginar uma forma de acelerar o programa *nn* (algoritmo 1) fazendo somente duas pequenas alterações? Com estas alterações, o tempo de processamento cai de 4 min para algo como 4 seg.

1 Os tempos de processamento desta apostila são medidos num computador com Intel i7 2.6 GHz.

```cpp
//nn.cpp 2024
#include "procimagem.h"

Mat_<uchar> vizinhanca33(Mat_<uchar> a, int lc, int cc) {
  Mat_<uchar> d(1,9);
  int i=0;
  for (int l=-1; l<=1; l++)
    for (int c=-1; c<=1; c++) {
      d(i)=a(lc+l,cc+c); i++;
    }
  return d;
}

int hamming(Mat_<uchar> a, Mat_<uchar> b) {
  if (a.total()!=b.total()) erro("Erro hamming");
  int soma=0;
  for (int i=0; i<a.total() ; i++)
    if (a(i)!=b(i)) soma++;
  return soma;
}

int main() {
  Mat_<uchar> AX=imread("ax.bmp",0);
  Mat_<uchar> AY=imread("ay.bmp",0);
  Mat_<uchar> QX=imread("qx.bmp",0);
  Mat_<uchar> QP(QX.size(),255);
  for (int l=1; l<QP.rows-1; l++) {
    if (l%10==0) printf("%d\n",l);
    for (int c=1; c<QP.cols-1; c++) {
      Mat_<uchar> qx=vizinhanca33(QX,l,c);
      //acha vizinho mais proximo de qx em AX
      int mindist=INT_MAX;
      uchar minsai=0;
      for (int l2=1; l2<AX.rows-1; l2++)
        for (int c2=1; c2<AX.cols-1; c2++) {
          Mat_<uchar> ax=vizinhanca33(AX,l2,c2);
          int dist=hamming(qx,ax);
          if (dist<mindist) { mindist=dist; minsai=AY(l2,c2); }
        }
      QP(l,c)=minsai;
    }
  }
  imwrite("qpnn.pgm",QP);
}
```

Algoritmo 1 – *nn.cpp*: Detecção de pontas de reta pela busca do vizinho mais próximo “força bruta”. Demora aproximadamente 4 minutos!

2.2 Acelerar a busca do vizinho mais próximo usando tabela

Resolvendo corretamente o exercício 2, o tempo de processamento diminui de 4 minutos para 4 segundos. Quatro segundos ainda é lento, longe de poder ser usado em aplicações de tempo real. Para ser tempo real, teria que processar aproximadamente 30 quadros por segundo.

Como acelerar mais o algoritmo? Uma ideia é construir uma tabela com as saídas para cada uma das 512 entradas possíveis (representando as $2^9 = 512$ configurações binárias dentro da janela 3×3). Com isso, para cada *qx*(*p*) basta fazer um único acesso à tabela (em vez de ter que percorrer toda a imagem AX). Fazendo isso, o tempo de processamento diminui para algo como 0,04s, aproximadamente 6000 vezes mais rápido que o algoritmo original!

Este algoritmo tem complexidade de tempo (quantidade de operações) $O(nl+ml+2^l)$, onde *n* de pixels de AX, *m* é número de pixels de QX e *l* é o número de pixels da janela.

A figura 3 mostra como o conteúdo de uma janela 3×3 pode ser convertida num número decimal entre 0 e 511 (ou número binário entre 000.000.000 e 111.111.111). Assim, uma tabela com 512 entradas pode armazenar as saídas das 512 possíveis configurações da janela 3×3 (0=ponta de reta ou 1=fundo).

1	1	1
1	0	0
1	1	1

111100111 → 0
487
(a) É ponta de reta

1	0	1
0	0	0
1	0	1

101000101 → 1
325
(b) Não é ponta de reta

1	1	1
1	1	1
1	1	1

111111111 → 1
511
(c) Não é ponta de reta

1	1	0
1	0	1
1	1	1

110101111 → 0
431
(d) É ponta de reta

0	1	0
1	0	1
0	1	0

010101010 → 1
170
(e) Não é ponta de reta

Figura 3: Exemplos de alguns conteúdos da janela e as respectivas saídas (0=ponta 1=fundo).

índice decimal	índice binário	conteúdo da tabela
0	000000000	1
...	...	...
(e) 170	010101010	1
...	...	...
(b) 325	101000101	1
...	...	...
(d) 431	110101111	0
...	...	...
(a) 487	111100111	0
...	...	...
(c) 511	111111111	1

Figura 4: Tabela que representa a função *f*. A tabela, na verdade, consiste somente da última coluna.

```
 1 //tabelann.cpp 2024
 2 #include "procimagem.h"
 3
 4 int hamming(int a, int b) {
 5   assert(0<=a && a<512);
 6   assert(0<=b && b<512);
 7   int soma=0;
 8   int bit=1;
 9   for (int i=0; i<9; i++) {
10     if ( (a & bit) != (b & bit) ) soma++;
11     i = i*2;
12   }
13   return soma;
14 }
15
16 int main(int argc, char** argv) {
17   if (argc!=5) erro("tabelann ax.bmp ay.bmp qx.bmp qp.bmp");
18   Mat_<uchar> AX=imread(argv[1],0);
19   Mat_<uchar> AY=imread(argv[2],0);
20   vector<uchar> tabela(512, 128); // Padrao nao visto vira cinza
21
22   //Percorre imagem AX e AY, preenchendo tabela
23   for (int l=1; l<AX.rows-1; l++)
24     for (int c=1; c<AX.cols-1; c++) {
25       int indice=0;
26       for (int l2=-1; l2<=1; l2++)
27         for (int c2=-1; c2<=1; c2++) {
28           int t=1;
29           if (AX(l+l2,c+c2)==0) t=0;
30           indice = 2*indice+t;
31         }
32       //Aqui, indice vai ser um numero entre 0 e 511
33       tabela[indice]=AY(l,c);
34     }
35
36   //Processa as entradas cinza (padroes nao vistos) da tabela.
37   for (int indice=0; indice<512; indice++) {
38     if (tabela[indice]==128) {
39       //procura o vizinho mais proximo na tabela
40       int mindist=INT_MAX;
41       uchar minsai=0;
42       for (int j=0; j<512; j++) {
43         if (tabela[j]!=128) {
44           int dist=hamming(indice,j);
45           if (dist<mindist) {
46             mindist=dist;
47             minsai=tabela[j];
48           }
49         }
50       }
51       tabela[indice]=minsai;
52     }
53   }
54
55   Mat_<uchar> QX=imread(argv[3],0);
56   Mat_<uchar> QP(QX.size(),255);
57   for (int l=1; l<QX.rows-1; l++)
58     for (int c=1; c<QX.cols-1; c++) {
59       int indice=0;
60       for (int l2=-1; l2<=1; l2++)
61         for (int c2=-1; c2<=1; c2++) {
62           int t=1;
63           if (QX(l+l2,c+c2)==0) t=0;
64           indice = 2*indice+t;
65         }
66       //Aqui, indice vai ser um numero entre 0 e 511
67       QP(l,c)=tabela[indice];
68     }
69   imwrite(argv[4],QP);
70 }
```

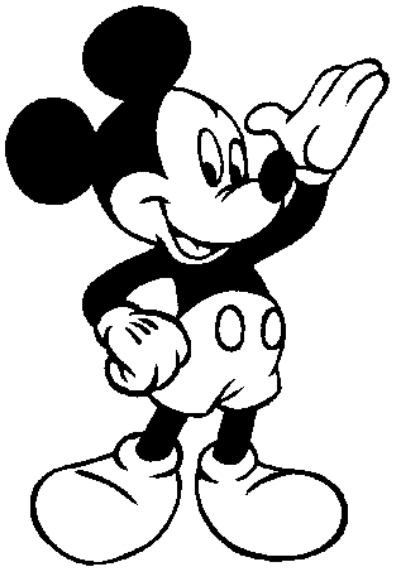
Algoritmo 2- *tabelann*: Projeto de filtro binário 3×3 usando vizinho mais próximo e tabela.

Este processo está implementado no programa tabelann (algoritmo 2). Este programa:
1) Linha 20: Cria uma tabela de 512 entradas e preenche-o de 128 (cinza).
2) Linhas 22-34: Percorre as imagens AX e AY, preenchendo a tabela com as saídas (0=ponta ou 255=fundo) das configurações das janelas que aparecem nos exemplos de treino.
3) Linhas 36-53: Percorre a tabela para detectar as configurações que não apareceram nos dados de treino (128=cinza). A saída de uma configuração não vista é aproximada para a configuração mais semelhante nos dados de treino, seguindo a estratégia do vizinho mais próximo.

4) Linhas 55-70: O algoritmo percorre a imagem QX. Para cada pixel *p*, o algoritmo pega a configuração da janela 3×3 ao seu redor e a converte num índice entre 0 e 511. Acessa a tabela usando esse índice para verificar qual é a saída daquela configuração e coloca o valor obtido na imagem de saída QP(*p*).

Programa *tabelann* (algoritmo 2) demora aproximadamente 0,04s. A saída obtida é exatamente igual à saída do *nn* "força bruta" (algoritmo 1).

O mesmo programa pode executar outras tarefas se fornecer exemplos de treinamento diferentes. Por exemplo, a figura 5 mostra esse mesmo programa detectando as bordas de imagens binárias, quando treinadas com imagens exemplos apropriadas. A figura 6 mostra o programa *tabelann* (algoritmo 2) emulando operadores morfológicos erosão 3×3 e abertura 2×2 a partir de imagens exemplos de entrada e saída.



Exemplo de treino de entrada AX mickey_reduz.bmp

Exemplo de treino de saída AY borda.png

Imagem a processar QX persuasion.png

Imagem processada QP

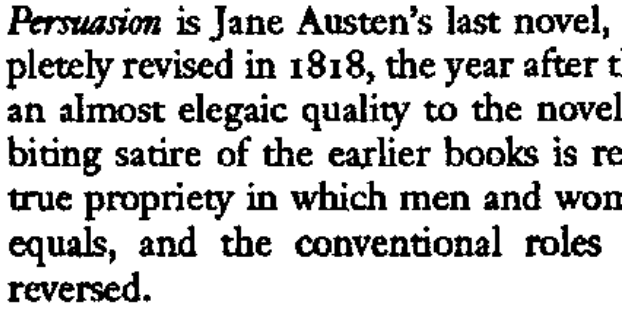


Imagem a processar QX pers.bmp

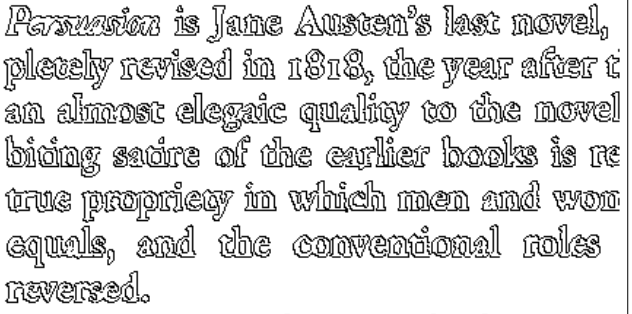
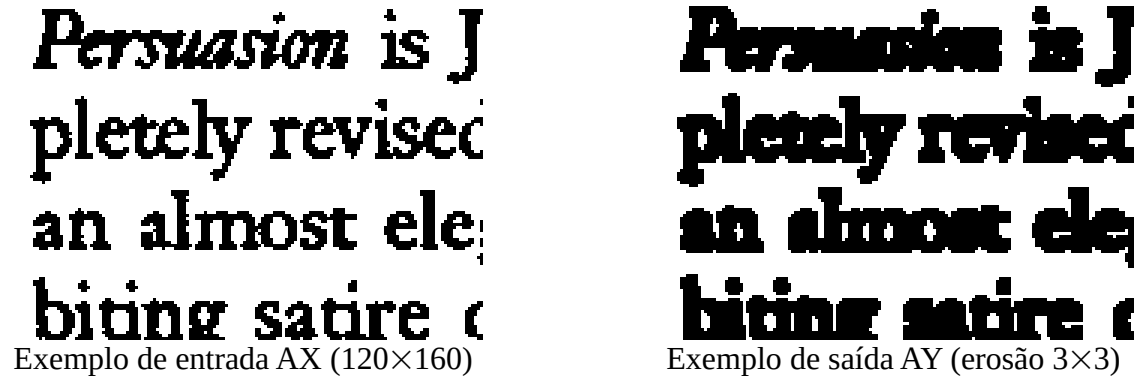


Imagem processada QP pers_borda.png

Figura 5: O programa tabelann (algoritmo 2) pode executar outras tarefas quando alimentado com imagens exemplos apropriados. Detecção de bordas.



Exemplo de entrada AX (120×160) Exemplo de saída AY (erosão 3×3)

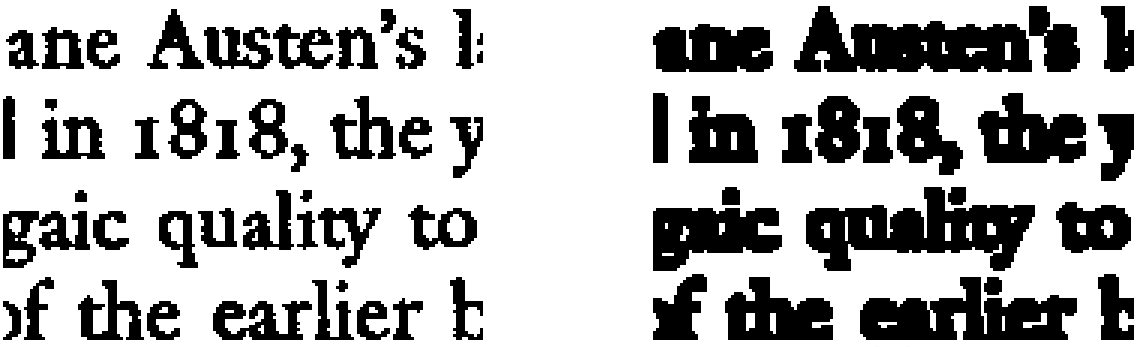
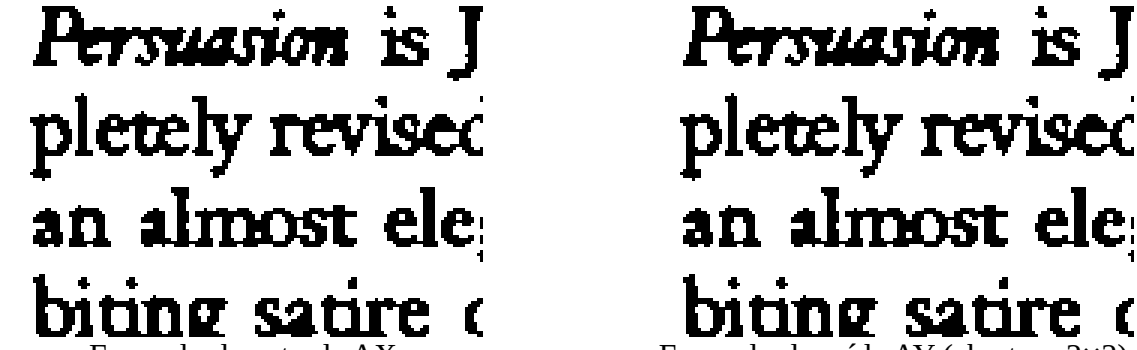


Imagem a processar QX (120×160) Erosão 3x3 emulado QP (erro zero)



Exemplo de entrada AX Exemplo de saída AY (abertura 2×2)

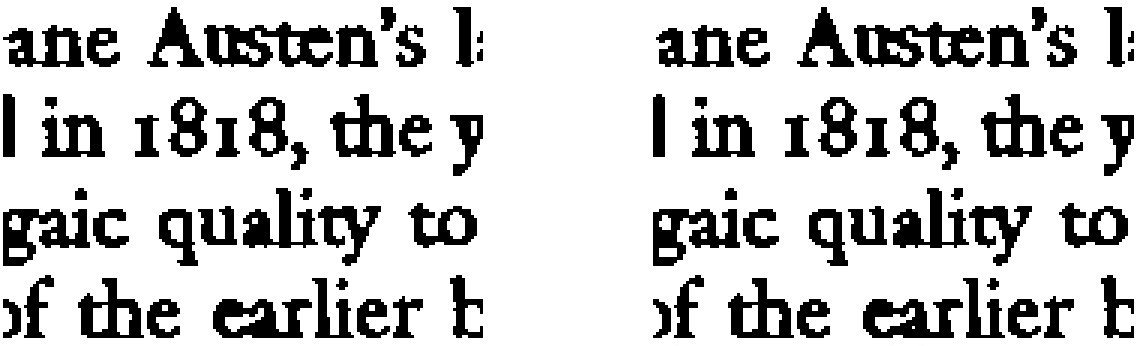


Imagem a processar QX Abertura 2×2 emulado QP (erro zero)

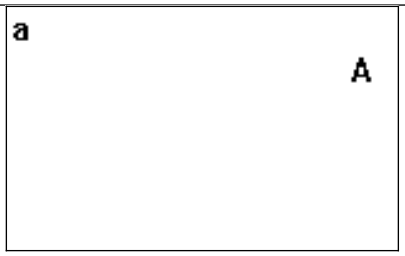
Figura 6: O programa tabelann (algoritmo 2) pode emular filtros 3×3 a partir de exemplos.

3. Detectar letras “a” e “A”.

3.1 O problema

Vimos como construir um filtro 3×3 para imagens binárias usando aprendizado vizinho mais próximo força-bruta e como acelerá-lo usando tabela. A figura 7 mostra a detecção de caracteres “a” e “A”, onde esses caracteres são mantidos intactos na saída e os outros caracteres são todos apagados. Para resolvermos este problema usando aprendizado do filtro espacial, precisamos usar uma janela suficientemente grande para poder reconhecer a letra “a” ou “A” olhando só o conteúdo dentro da janela. Experimentalmente, constata-se que janela 7×7 é uma escolha adequada. A função característica é $f:\{0,1\}^{7\times 7}\rightarrow\{0,1\}$.

Exemplo de entrada AX (lax.bmp)



Exemplo de saída AY (lay.bmp)

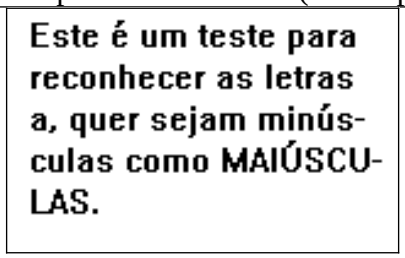


Imagem a processar QX (lqx.bmp)

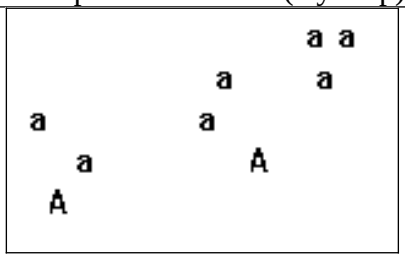


Imagem processada QP (lqp.bmp)

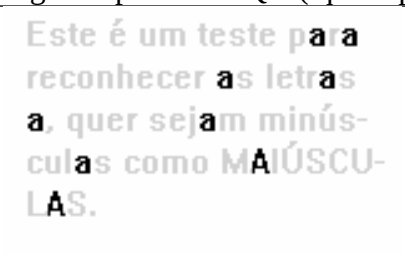


Imagem sobreposta (lpq.pgm)

Figura 7: Detecção de letras “A” e “a” com aprendizado de máquina.

Podemos fazer busca do vizinho mais próximo “força bruta”, semelhante ao programa *nn* (algoritmo 1), só que é lento. Por outro lado, não é possível usar tabela como no programa *tabelann* (algoritmo 2), pois aqui o tamanho da tabela seria 2^{49} = 562.949.953.421.312 (562 TB)!

Não dá para usar tabela, pois:
(a) A tabela não cabe na memória de nenhum computador;
(b) É impossível conseguir exemplos suficientes para preencher toda a tabela (ou uma parte considerável dela);
(c) O tempo para preencher toda a tabela seria astronômico.

Aumentando a quantidade de atributos de 9 para 49, o tamanho do espaço dos atributos aumenta exponencialmente e fica impossível treinar o modelo. Este fenômeno é conhecido como a "maldição da dimensionalidade" (curse of dimensionality) [Shetty2022, wiki-curse, Raj2021, Yiu2019].

Portanto, em aprendizado de máquina, é importante reduzir a dimensão dos dados de entrada tanto quanto possível.

Podemos evitar ter que criar uma tabela com 500TB ao mesmo tempo em que se acelera o processamento usando uma árvore de decisão em vez da tabela. Note que o tamanho da tabela $O(2^l)$ "explode" com o crescimento da janela (de l=9 para l=49). Por outro lado, o tamanho da árvore de decisão não depende do tamanho da janela l mas depende do número de amostras de treinamento n (ou o número de pixels das imagens AX e AY) e portanto não "explode" com o aumento da janela. Assim, é possível usar árvore de decisão para acelerar este problema. O problema é que árvore de decisão e vizinho mais próximo geram saídas diferentes.

3.2 Estratégias para generalização e para lidar com exemplos conflitantes

Não há dúvida sobre como um aprendiz deve classificar uma instância teste qx se essa instância aparece nos dados de treino, isto é, se há um dado de treino (ax, ay) onde ax=qx. Neste caso, deve-se gerar a mesma saída do exemplo de treino, fazendo $M(qx)=ay$. No problema da figura F2, obviamente $M(1)$=1 e $M(3)$=3 pois há exemplos de treino (1; 1) e (3; 3). Um aprendiz com esta característica é chamado de "consistente".

Porém, não são claras quais devem ser as saídas $M(1{,}5)$ e $M(2)$, pois não há exemplo de treino para x=1,5 e há várias saídas diferentes para x=2.

Todo algoritmo de aprendizado de máquina deve ter estratégias para resolver dois problemas:

(a) Generalização ou viés indutivo (inductive bias): Como classificar uma instância qx que nunca apareceu nos exemplos de treino? Para fazer predição do ponto x=1,5 na figura F2, devemos fazer uma generalização, pois não há exemplo de treino com x=1,5. A estratégia de generalização do vizinho mais próximo consiste em procurar o exemplo de treino mais parecido a qx. Árvore de decisão usa uma outra generalização. Procura-se subdividir sucessivamente o conjunto de treino em dois subconjuntos de forma a maximizar o ganho de informação ou minimizar o índice Gini, construindo uma árvore (veja o anexo B). A instância de teste qx é classificada de acordo com o valor de saída da folha a que se chega percorrendo a árvore.

(b) Exemplos com conflito: O que fazer quando uma amostra de entrada ax possui várias saída ay diferentes (conflitantes)? Qual deve ser a saída de x=2,0 na figura F2? Neste caso, o razoável é escolher a moda, a média ou a mediana das saídas. Cada uma das 3 soluções minimiza uma métrica de erro diferente.

A saída da árvore de decisão será diferente da saída do vizinho mais próximo, pois ambos possuem estratégias diferentes de generalização. Portanto, não é correto usar árvore de decisão para acelerar o vizinho mais próximo.

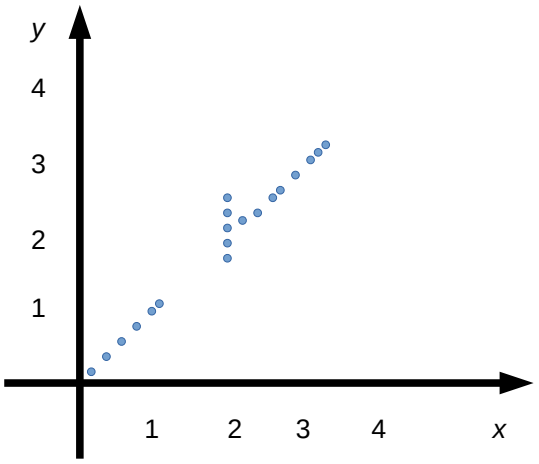


Figura F2: Neste conjunto de exemplos de treino, não há exemplo para entrada x=1,5 e há conflitos para entrada x=2. Quais devem ser as saídas para x=1,5 e x=2?

3.3 Árvore de decisão e kd-árvore

Vimos que:

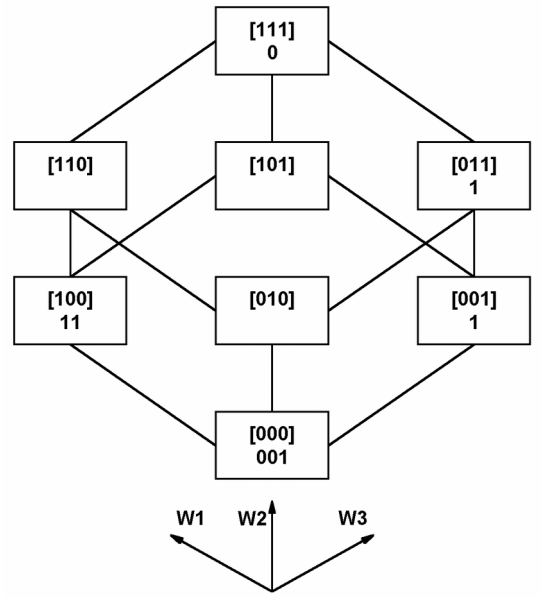
- Não é possível usar tabela para acelerar a busca de letras “a” e “A”.
- As estratégias de generalização do vizinho mais próximo e da árvore de decisão são diferentes e portanto, não é correto usar árvore de decisão para acelerar o vizinho mais próximo.

O que podemos fazer para acelerar a busca do vizinho mais próximo? Existem outras técnicas projetadas especialmente para acelerar a busca do vizinho mais próximo. Uma dela é árvore k-dimensional (*kd-tree* ou kd-árvore) [wikiKD, Bentley1975, Friedman1977]. Aqui, vou usar a versão kd-árvore otimizada de [Friedman1977], que é um pouco diferente da original [Bentley1975], e chamá-la simplesmente de kd-árvore (sem especificar que é otimizada). Nem sempre kd-árvore retorna o vizinho mais próximo – pode retornar um vizinho “mais ou menos próximo”.

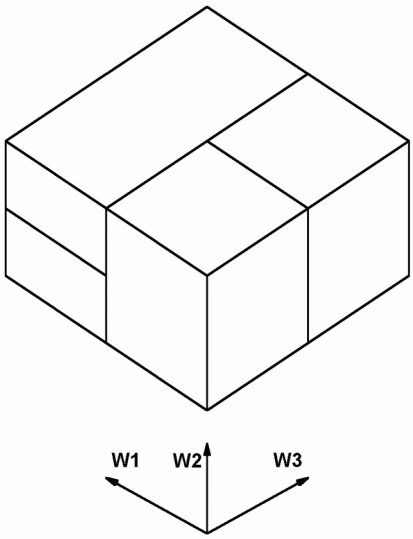
A construção de tanto árvore de decisão como kd-árvore consiste em cortar sucessivamente o espaço dos atributos (figura F3-b). Porém, o critério de escolha do plano e valor de corte diferem.

a) A árvore de decisão escolhe o plano e o valor de corte procurando maximizar o ganho de entropia ou minimizar o índice Gini (anexo B). Para isso, é necessário levar em consideração tanto dados de treino de entrada AX como de saída AY.

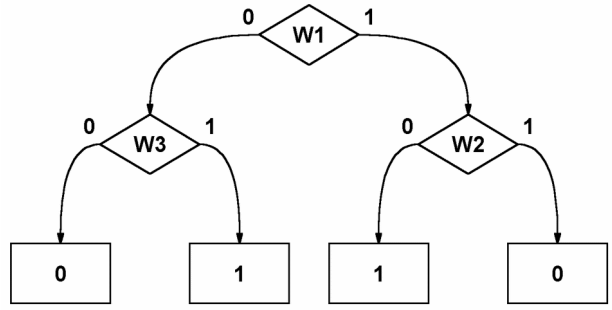
b) Para construir kd-árvore, somente os atributos de entrada AX são levados em em consideração (não utiliza AY). O processo de construção escolhe ciclicamente os planos de corte, isto é, W1, W2, W3, W1, W2, etc. Em cada corte, procura dividir os exemplos de forma a deixar a mesma quantidade de amostras nas duas sub-caixas resultantes, isto é, escolhe o valor mediano para efetuar o corte.



(a) Espaço dos atributos com exemplos de treino.



(b) Cortes no espaço dos atributos.



(c) Árvore construída correspondente aos cortes.

Figura F3: Construção de árvore de decisão e kd-árvore com exemplos de treino = { [000, 0], [000, 0], [000, 1], [001, 1], [100, 1], [100, 1], [011, 1], [111, 0] }.

Também há diferença durante a predição entre árvore de decisão e kd-árvore.

a) Na árvore de decisão, dada uma instância a classificar *qx*, percorre-se a árvore até chegar a uma folha (um sub-cubo onde não tem mais subdivisões). A saída *qp=M*(*qx*) fica definida pelo rótulo que encontrar nessa folha.

b) Na kd-árvore, para buscar um vizinho próximo de *qx*, percorre-se a árvore até chegar a uma folha. Na folha, tem uma amostra *ax* que é um vizinho "mais ou menos próximo" de *qx* mas não tem garantia de que *ax* seja o vizinho mais próximo de *qx*. Para minimizar a possibilidade de que kd-árvore devolva um vizinho *ax* distante de *qx*, costuma-se usar duas estratégias:

(b1) Faz-se "backtracking", isto é, faz buscas nos cubos vizinhos de *ax* para verificar se há um vizinho mais próximo do que *ax*.

(b2) Criam-se várias kd-árvores t_1, t_2, ..., t_n escolhendo, para cada uma, diferentes sequências de cortes. Na hora do teste, percorre-se todas as *n* árvores, obtendo *n* candidatos a vizinho mais próximo de *qx*. Dentre eles, escolhe-se aquele que for o mais próximo de *qx*.

As estratégias (b1) e (b2) podem ser usadas conjuntamente. Mesmo assim, não há garantias de que kd-árvore irá retornar sempre o vizinho mais próximo verdadeiro.

As duas estruturas estão implementadas nas bibliotecas OpenCV e SkLearn.

a) FlaNN (Fast Library for Approximate Nearest Neighbors) é uma sub-biblioteca da OpenCV que permite fazer busca rápida (mas aproximada) do vizinho mais próximo. FlaNN implementa kd-tree (e vários outros métodos) para acelerar a busca aproximada do vizinho mais próximo. Usei FlaNN durante muito tempo e nunca encontrei nenhum problema. SkLearn parece oferecer uma solução ainda mais completa em sklearn.neighbors [https://scikit-learn.org/stable/modules/neighbors.html] mas nunca a usei.

b) Árvore de decisão está implementada dentro de *OpenCV v2* na classe *CvDTree* e na classe *tree* da biblioteca *SkLearn*. A implementação de árvore de decisão de SkLearn é melhor do que a da OpenCV2.

Em Cekeikon para Windows, há uma implementação de kd-árvore sem backtracking. A seguinte sequência de comandos (só roda em Cekeikon para Windows – é possível rodar Cekeikon para Windows em Linux, usando emulador de Windows Wine) resolve o problema usando árvore:

```
>mle cutb lax.bmp lay.bmp c:\cekeikon5\proeikon\ee\e77.ges lfiltro.ctb
>mle appb lqx.bmp lfiltro.ctb lqp.bmp
>img sobrmcb lqx.bmp lqp.bmp lqp.pgm g
```

Sendo 0.34s para construir o filtro e 0.19s para aplicar o filtro.

[PSI5790 aula 5. Lição de casa #1 (de 2).] Resolva o problema de manter as letras “a” e “A” e apaga as outras letras da imagem, ilustrada na figura 7, usando algum método de aprendizado de máquina (pode escolher o método livremente). As sugestões são vizinho mais próximo “força bruta”, vizinho mais próximo aproximado usando kd-árvore ou árvore de decisão. Pode se usar como exemplos os programas que vão aparecer nesta apostila mais adiante. O seu programa só precisa funcionar para o tipo de fonte das imagens fornecidas (lax.bmp, lay.bmp e lqx.bmp).

4. Segmentação de feijões pela cor

4.1 O problema

Vamos considerar a segmentação de uma imagem pela cor dos pixels. Considere a figura 11c com feijões sobre cartolina branca. Gostaríamos de pintar os feijões de preto e o fundo de branco, baseado somente nas cores dos pixels. Se você tentar resolver este problema escrevendo manualmente um programa (que não seja baseado em aprendizado de máquina) irá constatar que é mais difícil do que pode parecer à primeira vista. Feijão possui várias tonalidades e há reflexos de luz. A cartolina tem sombras que fazem mudar a tonalidade.

A figura 12 mostra as projeções de todos as cores da figura 11c (f1.jpg) nos planos de cores RG, RB e GB. Pode-se observar que não há uma separação nítida entre as cores dos feijões (marrom) e as cores da cartolina (cinza) no espaço RGB. Isto dificulta separar feijão do fundo.

Para resolver este problema usando aprendizado de máquina, podemos pegar uma sub-imagem de f1.jpg (AX figura 11a) e pintar manualmente de preto o feijão e de branco o fundo (imagem AY, figura 11b). Aí, fornecemos este par de imagens AX, AY como exemplos de treinamento para projetar um filtro *F*. Aplicamos este filtro *F* à imagem QX e esperamos que gere uma imagem de saída QP onde os feijões estão pintados de preto e o fundo de branco (figura 11d).

Pergunta: Qual é o tamanho de janela adequada para este filtro? A resposta correta é 1×1, pois o valor de um pixel de saída depende exclusivamente da cor do pixel de entrada. Não depende de uma vizinhança. A função característica deste filtro é $f:[0,255]^3 \rightarrow \{0,1\}$ e o seu domínio (espaço das características) é o espaço das cores RGB com 256^3 = 16.777.216 elementos.

Exercício: Use rede neural convolucional para segmentação semântica para resolver este problema . A segmentação semântica irá usar uma janela muito maior. Compare com os resultados obtidos usando pelos algoritmos de aprendizado clássicos.

Vimos quatro métodos até agora:
(a) busca do vizinho mais próximo “força bruta”;
(b) vizinho mais próximo usando tabela;
(c) árvore de decisão; e
(d) vizinho mais próximo aproximado usando kd-árvore.

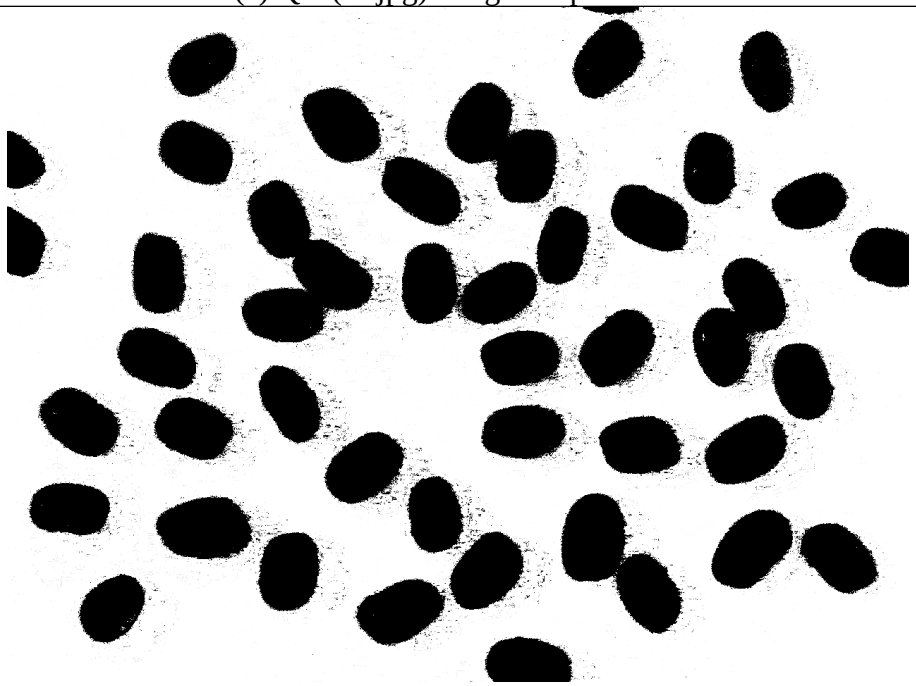
Para este problema, método (a) demoraria muito e seria difícil usar tabela (método b) pelo tamanho do domínio (256^3 = 16.777.216). É possível usar árvore de decisão (método c) e vizinho mais próximo aproximado usando kd-árvore/FlaNN (método d).

Exercício: Resolva este problema usando vizinho mais próximo “força bruta” $O(nml)$.
Resultado: O programa demora aproximadamente 8 minutos.

(a) AX amostra de entrada

(b) AY amostra de saída

(c) QX (f1.jpg) imagem a processar



(d) QP imagem processada pelo 1-NN (vizinho mais próximo).

Figura 11: Segmentação de uma imagem pela cor dos pixels.

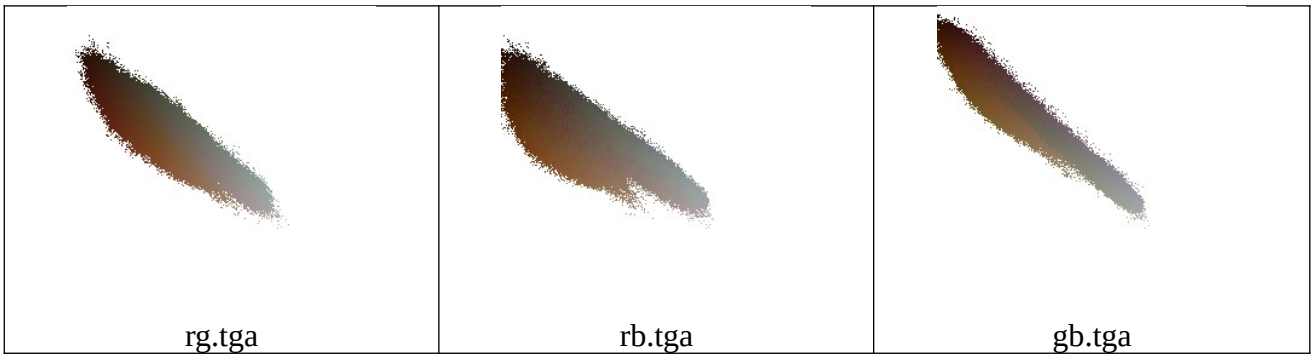


Figura 12: Projeções do conjunto de pixels da imagem f1.jpg nos planos RG, RB e GB do cubo RGB.

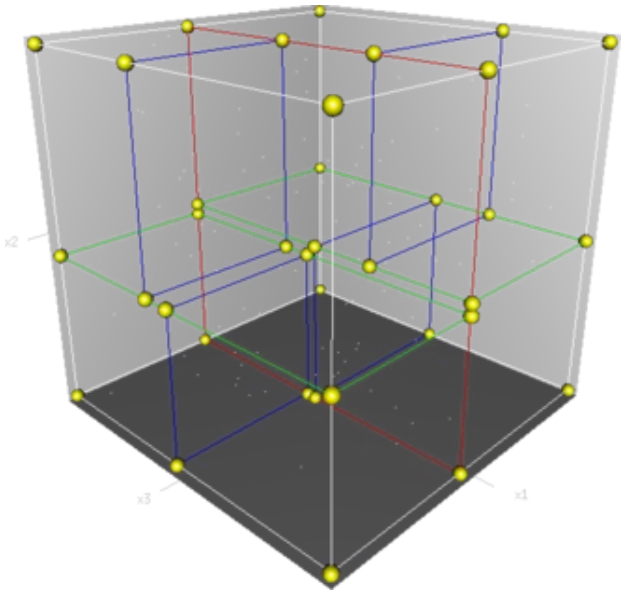


Figura F4: Cortes num espaço de atributos não-binário.

```
1  //flann.cpp 2024
2  //segmentacao de feijao usando FlaNN
3  //Pode usar OpenCV v2, v3 ou v4
4  #include "procimagem.h"
5
6  int main() {
7    Mat_<Vec3b> ax=imread("ax.png",1);
8    Mat_<uchar> ay=imread("ay.png",0);
9    Mat_<Vec3b> qx=imread("f1.jpg",1);
10   if (ax.size()!=ay.size()) erro("Erro dimensao");
11   Mat_<uchar> qp(qx.rows,qx.cols);
12
13   //Cria as estruturas de dados para alimentar OpenCV
14   Mat_<float> features(ax.rows*ax.cols,3);
15   Mat_<int> saidas(ax.rows*ax.cols,1);
16   int i=0;
17   for (int l=0; l<ax.rows; l++)
18     for (int c=0; c<ax.cols; c++) {
19       features(i,0)=ax(l,c)[0]/255.0;
20       features(i,1)=ax(l,c)[1]/255.0;
21       features(i,2)=ax(l,c)[2]/255.0;
22       saidas(i)=ay(l,c);
23       i=i+1;
24     }
25   flann::Index ind(features,flann::KDTreeIndexParams(4));
26   // Aqui, as 4 arvores estao criadas
27
28   Mat_<float> query(1,3);
29   vector<int> indices(1);
30   vector<float> dists(1);
31   for (int l=0; l<qp.rows; l++)
32     for (int c=0; c<qp.cols; c++) {
33       query(0)=qx(l,c)[0]/255.0;
34       query(1)=qx(l,c)[1]/255.0;
35       query(2)=qx(l,c)[2]/255.0;
36       // Zero indica sem backtracking
37       ind.knnSearch(query,indices,dists,1,flann::SearchParams(0));
38       qp(l,c)=saidas(indices[0]);
39     }
40   imwrite("f1-flann.png",qp);
41 }
42
```

Algoritmo 3: Uso de FlaNN (vizinho mais próximo aproximado usando kd-tree) para segmentar feijões.

O algoritmo 3 mostra o uso de FlaNN (de OpenCV 2/3/4) para segmentar feijões. As linhas 7-11 lêem as imagens AX, AY, QX e cria a imagem de saída QP.

A maioria das rotinas de aprendizado de máquina de OpenCV2 exigem que as amostras de treinamento sejam colocados em duas matrizes tipo “float” de 32 bits. Parece que no OpenCV v3/v4 a matriz de saída muitas vezes deve ser obrigatoriamente “int” de 32 bits enquanto que algumas vezes permite que seja “float32.”

OpenCV2	OpenCV v3/v4
`Mat_<float> features(namostras,nfeatures);` `Mat_<float> saidas(namostras,1);`	`Mat_<float> features(namostras,nfeatures);` `Mat_<int> saidas(namostras,1);`

onde cada amostra (*ax*, *ay*) deve ser colocada numa linha da matriz. As linhas 14-24 do algoritmo 3 fazem a conversão das imagens AX, AY para o formato desejado por OpenCV. Matriz *features* tem 3 colunas, para armazenar as cores BGR. Estou adotando a convenção de que cores BGR em uint8 (0 a 255) são convertidas para intervalo [0,1] quando forem armazenadas como *float*.

Features ax		
B	G	R
...		

Saidas ay
Saida

A linha 25 cria 4 kd-árvores para buscar o vizinho próximo. As buscas serão feitas nas 4 árvores e será escolhido o exemplo mais parecido com a instância de teste *qx*.

A rotina que faz busca na kd-árvore está na linha 37:

```
ind.knnSearch(query,indices,dists,1,flann::SearchParams(0));
```

Esta rotina deve ser alimentada com matriz *query* com 3 colunas BGR. Os vetores *indices* e *dists* devolvem respectivamente o índice do vizinho mais próximo na matriz *features* e a distância entre *qx* e o vizinho devolvido. Este método pode devolver mais de um vizinho mais próximo se especificarmos $k>1$ (em *k*-nearest neighbors) e portanto *indices* e *dists* são vetores.

Este programa demora 4s para processar e a saída obtida está na figura 13a.

4.3 Árvore de decisão da OpenCV 2.X

```
 1 //dtree.cpp grad2017
 2 //Usa OpenCV2
 3 #include <cekeikon.h>
 4
 5 int main() {
 6   Mat_<COR> ax; le(ax,"ax.png");
 7   Mat_<GRY> ay; le(ay,"ay.png");
 8   if (ax.size()!=ay.size()) erro("Erro: Dimensoes diferentes");
 9
10   Mat_<FLT> features(ax.total(),3);
11   Mat_<FLT> saidas(ay.total(),1);
12   for (unsigned i=0; i<ax.total(); i++) {
13     features(i,0)=ax(i)[0]/255.0;
14     features(i,1)=ax(i)[1]/255.0;
15     features(i,2)=ax(i)[2]/255.0;
16     saidas(i,0)=ay(i)/255.0;
17   }
18
19   CvDTree arvore;
20   arvore.train(features,CV_ROW_SAMPLE,saidas);
21
22   Mat_<COR> qx; le(qx,"f1.jpg");
23   Mat_<GRY> qp(qx.rows,qx.cols);
24   for (unsigned i=0; i<qx.total(); i++) {
25     Mat_<FLT> query(1,3);
26     query(0)=qx(i)[0]/255.0;
27     query(1)=qx(i)[1]/255.0;
28     query(2)=qx(i)[2]/255.0;
29     CvDTreeNode* no=arvore.predict(query);
30     int r=255*(*no).value;
31     qp(i)=r;
32   }
33   imp(qp,"f1-dtree.png");
34 }
```

Algoritmo 4a: Uso de árvore de decisão para segmentar feijões em OpenCV2/C++. Só dá para compilar usando Cekeikon: compila dtree -ocv.

Nota: Estou obtendo erro de execução quando traduzo o programa acima de OpenCV2 para OpenCV v3/v4.

O mesmo problema pode ser resolvido usando árvore de decisão (algoritmo 4). As linhas 19-20 criam a árvore de decisão. As linhas 29-30 percorrem a árvore e devolvem o valor de saída.

Este programa demorou 0,2s para processar e a saída está na figura 13b. A qualidade da saída é pobre.

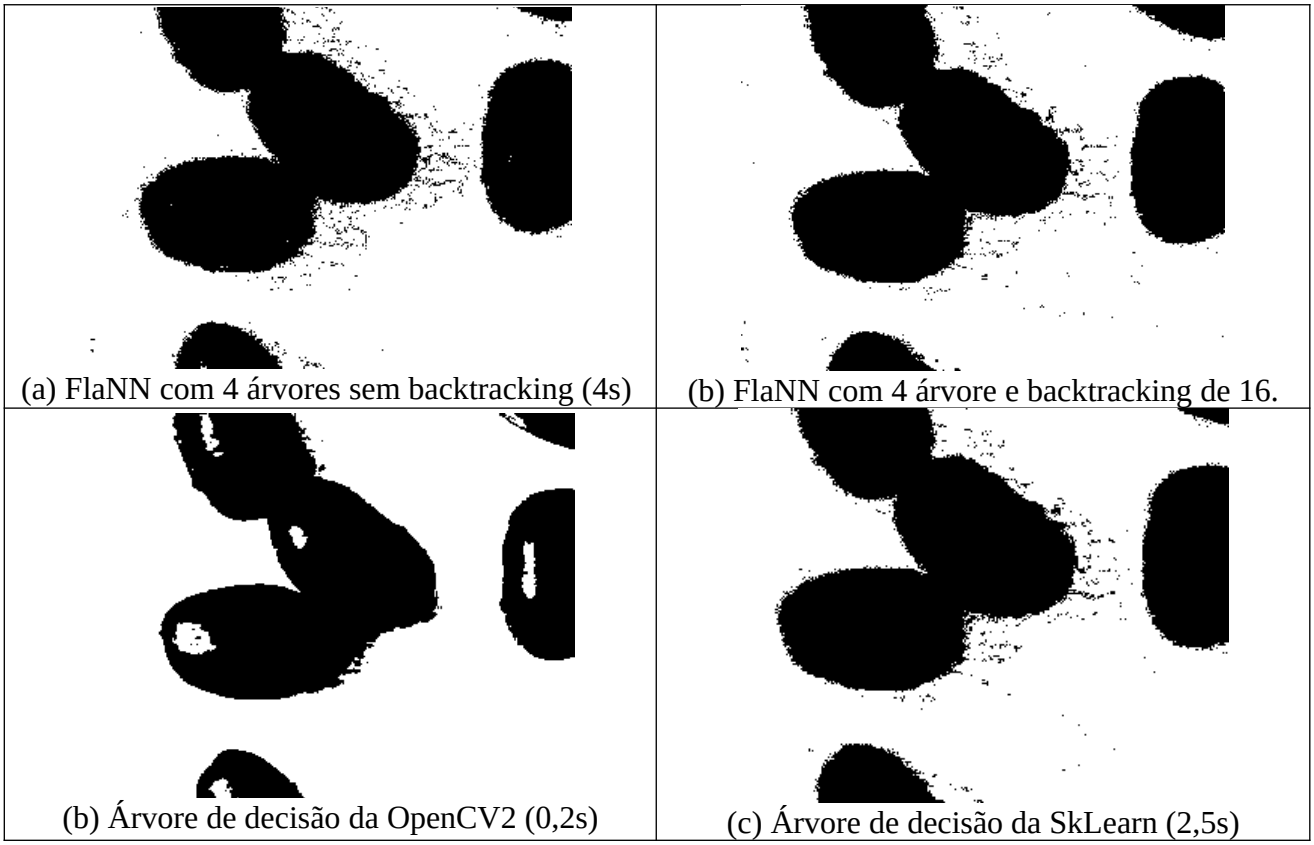
4.4 Árvore de decisão da SkLearn

```
#dtree.py - grad2023
import numpy as np
from sklearn import tree
import sys
import cv2

#<<<<<<<<<<<<<<< main <<<<<<<<<<<<<<<<<
ax=cv2.imread("ax.png",1)
ay=cv2.imread("ay.png",0)
if ax.shape[0]!=ay.shape[0] or ax.shape[1]!=ay.shape[1]:
  sys.exit("Erro: Imagens com dimensoes diferentes");

features=(ax/255).reshape((ax.shape[0]*ax.shape[1],3))
saidas  =(ay/255).reshape((ay.shape[0]*ay.shape[1],))

arvore= tree.DecisionTreeClassifier()
arvore= arvore.fit(features, saidas)

qx=cv2.imread("f1.jpg",1)
query=(qx/255).reshape((qx.shape[0]*qx.shape[1],3))

qp=arvore.predict(query)
qp=qp.reshape((qx.shape[0],qx.shape[1]))
cv2.imwrite("f1-dt-py.png",255*qp)
```

Algoritmo 4b: Uso de árvore de decisão para segmentar feijões em SkLearn/Python.

O mesmo problema pode ser resolvido usando árvore de decisão de SkLearn.

Este programa demorou 0,6s para processar e a saída está na figura 13c. Note que a qualidade de saída é melhor do que a obtida usando OpenCV.



(a) FlaNN com 4 árvores sem backtracking (4s)

(b) FlaNN com 4 árvore e backtracking de 16.

(b) Árvore de decisão da OpenCV2 (0,2s)

(c) Árvore de decisão da SkLearn (2,5s)

Figura 13: Saídas da segmentação de feijões. A qualidade de árvore de decisão de OpenCV2 é ruim.

4.5 Boosting

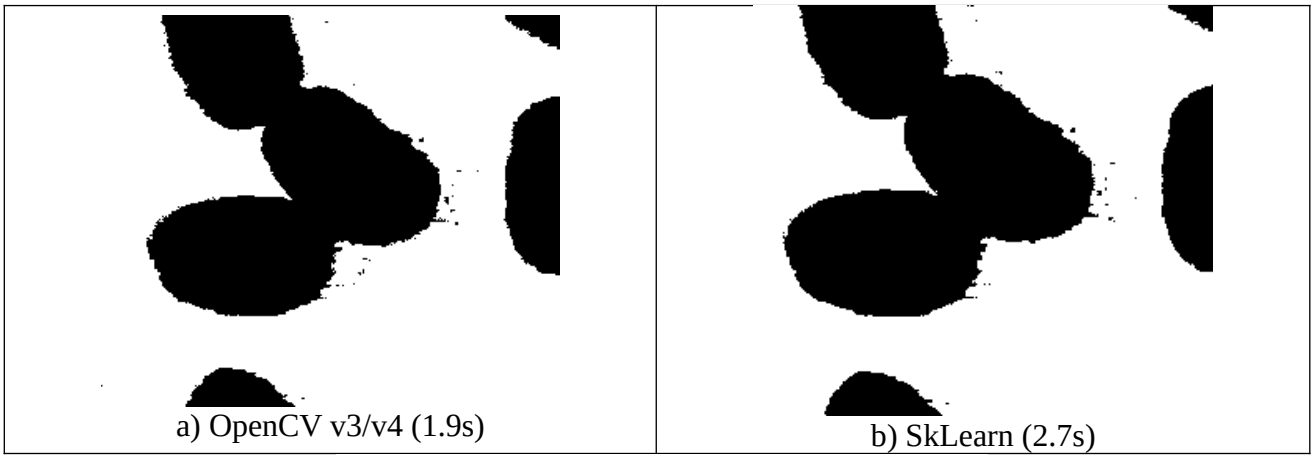
Boosting calcula a média ponderada das saídas de muitos classificadores “fracos” para construir um classificador “forte”. Os classificadores “fracos” são tipicamente árvores de decisão e os seus pesos são calculados a partir das suas taxas de erro. Algoritmo 5a usa boost do OpenCV 2.X para segmentar feijões e algoritmo 5b usa boost do SkLearn. Ambas as saídas são semelhantes (figuras 17a e 17b). Há mais explicações sobre boosting no anexo A.

```
1  //boosting.cpp 2024 OpenCV v3/v4
2  #include "procimagem.h"
3
4  int main() {
5    Mat_<Vec3b> ax=imread("ax.png",1);
6    Mat_<uchar> ay=imread("ay.png",0);
7    if (ax.size()!=ay.size()) erro("Erro: Dimensoes diferentes");
8
9    Mat_<float> features(ax.total(),3);
10   Mat_<int> saidas(ay.total(),1);
11   for (unsigned i=0; i<ax.total(); i++) {
12     features(i,0)=ax(i)[0]/255.0;
13     features(i,1)=ax(i)[1]/255.0;
14     features(i,2)=ax(i)[2]/255.0;
15     saidas(i,0)=ay(i);
16   }
17
18   Ptr<ml::Boost> ind = ml::Boost::create();
19   ind->train(features,ml::ROW_SAMPLE,saidas);
20   Mat_<Vec3b> qx=imread("f1.jpg",1);
21   Mat_<uchar> qp(qx.rows,qx.cols);
22
23   for (unsigned i=0; i<qx.total(); i++) {
24     Mat_<float> query(1,3);
25     query(0)=qx(i)[0]/255.0;
26     query(1)=qx(i)[1]/255.0;
27     query(2)=qx(i)[2]/255.0;
28     qp(i)=ind->predict(query);
29   }
30   imwrite("f1-boosting.png",qp);
31 }
```

Algoritmo 5a - boost: Uso de boosting para segmentar feijões em OpenCV v3/v4.

```python
#boost.py - grad2023
import numpy as np
from sklearn import ensemble
import sys, cv2

#<<<<<<<<<<<<<<< main <<<<<<<<<<<<<<<<
ax=cv2.imread("ax.png",1)
ay=cv2.imread("ay.png",0)
if ax.shape[0]!=ay.shape[0] or ax.shape[1]!=ay.shape[1]:
  sys.exit("Erro: Imagens com dimensoes diferentes");

features=(ax/255).reshape((ax.shape[0]*ax.shape[1],3))
saidas  =(ay/255).reshape((ay.shape[0]*ax.shape[1],))

clf = ensemble.GradientBoostingClassifier()
clf = clf.fit(features,saidas)

qx = cv2.imread("f1.jpg",1)
query = (qx/255).reshape((qx.shape[0]*qx.shape[1],3))

qp = clf.predict(query)
qp = qp.reshape((qx.shape[0],qx.shape[1]))
cv2.imwrite("f1-bo-py.png",255*qp)
```

Algoritmo 5b - boost: Uso de boosting para segmentar feijões em SkLearn.



a) OpenCV v3/v4 (1.9s)

b) SkLearn (2.7s)

Figura X: Segmentação de feijão usando Boosting.

4.6 Classificador de Bayes

Classificador *Normal Bayes* assume que os vetores de características de cada classe são distribuições normais, mas não necessariamente independentes. Assim, assume que a distribuição total dos dados é uma mistura de gaussianas, um componente por classe.

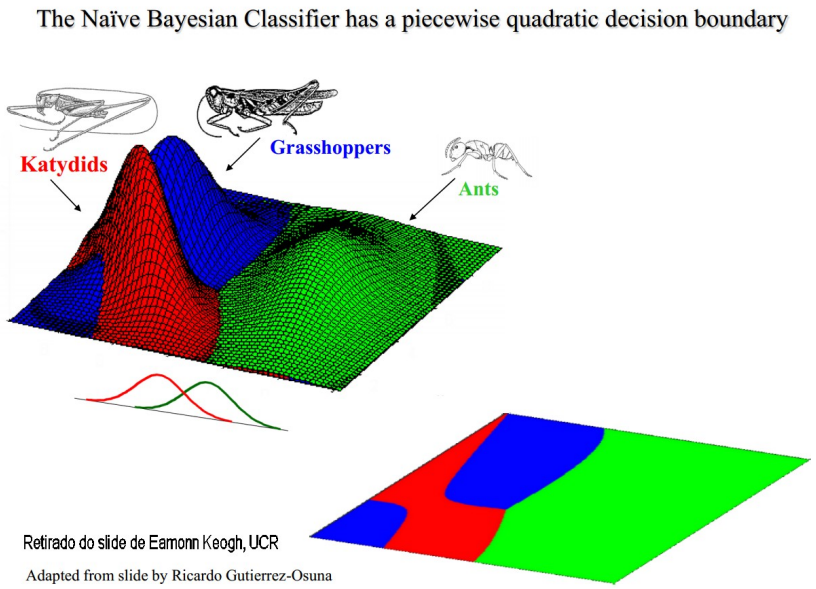


Figura 15: Classificação em grilo, gafanhoto e formiga usando classificador Naive Bayes.

Na figura 15, a partir dos pesos e comprimentos de grilos, gafanhotos e formigas, o classificador Normal Bayes calculou as médias vetoriais e as matrizes de covariância das três classes, que modelam as 3 distribuições normais. Plotando os três modelos obtidos num gráfico 2D, obtém gráfico como mostrado na figura 15-superior. Com isso, é possível dividir o espaço dos atributos em regiões onde é mais provável que o inseto seja de uma certa classe (figura 15-inferior). Com isso, dado peso e comprimento de um inseto, é possível escolher a classe de inseto mais provável com essas características.

O classificador *Naive Bayes* (mais conhecido e implementado no SkLearn) é diferente de *Normal Bayes* (implementado no OpenCV). O primeiro assume que as variáveis são independentes entre si enquanto que o segundo não faz esta suposição.
[https://blog.actorsfit.com/a?ID=00500-abdcbbaa-9619-43b8-8ff0-b08b85cee7e2]

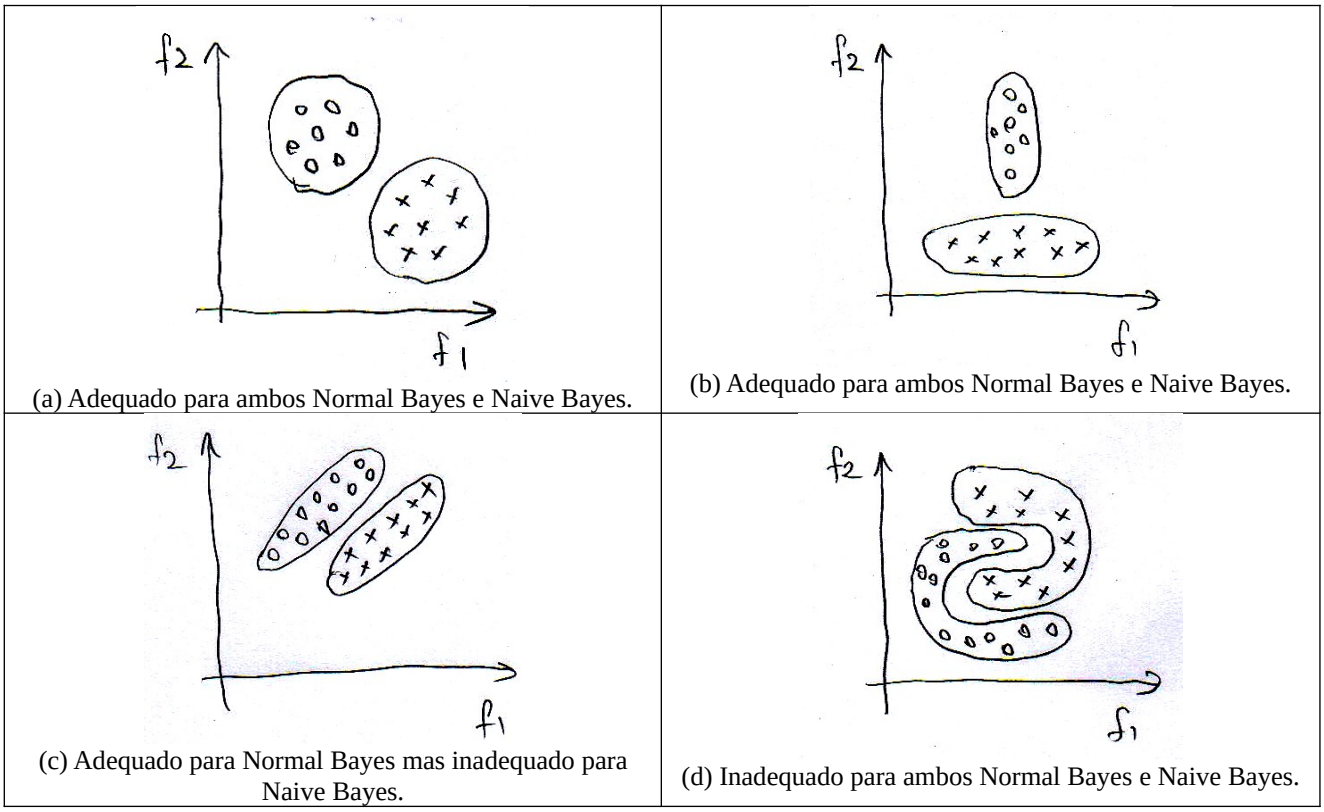
A figura 16 mostra algumas distribuções de classes adequadas e inadequadas para serem resolvidas usando classificadores de Bayes. Diremos que um algoritmo de aprendizado é inadequado para resolver um determinado problema se a sua taxa de erro esperada é alta, mesmo usando parâmetros adequados e um número grande de amostras de treinamento. O classificador Naive Bayes só consegue classificar corretamente distribuições gaussianas alinhadas aos eixos do sistema de coordenadas, devido à suposição de independência das variáveis. O classificador Normal Bayes consegue classificar corretamente também distribuições gaussiana inclinadas.

Veja na figura 12 como são as distribuições das cores dos feijões e da cartolina: as distribuições lembram gaussianas, mas não estão alinhadas aos eixos. Neste caso, não se pode assumir que as variáveis sejam independentes. Consequentemente, a qualidade do classificador Naive Bayes será

muito pior do que Normal Bayes para este problema. As saídas obtidas na figura 17 confirmam esta previsão.

Existem muitas outras técnicas clássicas de aprendizado de máquina que não trataremos aqui: regressão logística, random forest, SVM, etc.

[Nota para próximo ano: Escrever sobre regressão logística, random forest, SVM.]



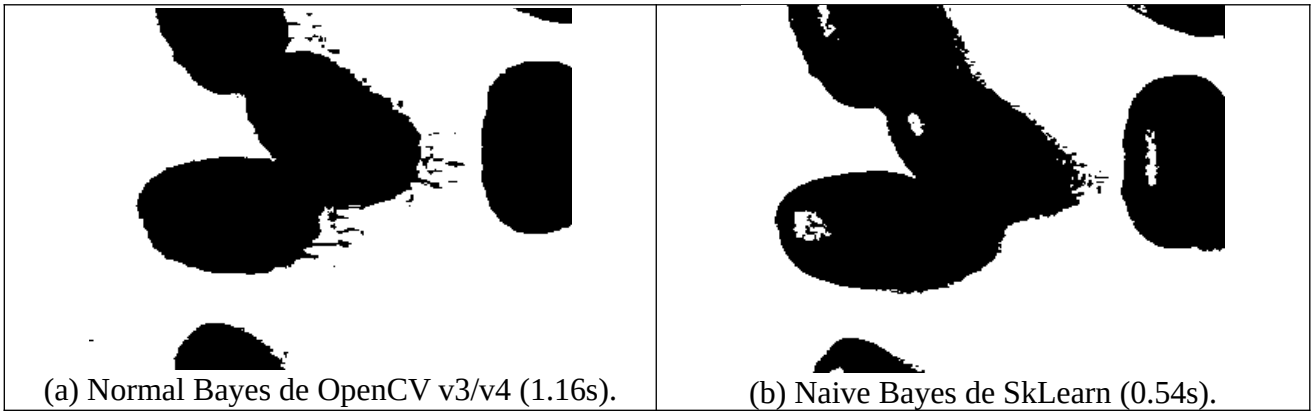
(a) Adequado para ambos Normal Bayes e Naive Bayes.

(b) Adequado para ambos Normal Bayes e Naive Bayes.

(c) Adequado para Normal Bayes mas inadequado para Naive Bayes.

(d) Inadequado para ambos Normal Bayes e Naive Bayes.

Figura 16: Algumas distribuições de classes adequadas e inadequadas para serem classificadas usando Bayes. Veja na figura 12 que este problema poderia ser resolvido adequadamente usando Normal Bayes mas não usando Naive Bayes.



(a) Normal Bayes de OpenCV v3/v4 (1.16s).

(b) Naive Bayes de SkLearn (0.54s).

Figura 17: Segmentação de feijão usando (a) Normal Bayes e (b) Naive Bayes. Neste problema, a qualidade de Normal Bayes (a) é bem superior a Naive Bayes (b), como era esperado pela distribuição das classes representada na figura 12.

Exercício: Resolva este problema usando classificador de Bayes de alguma outra biblioteca, por exemplo, Normal Bayes de OpenCV4.

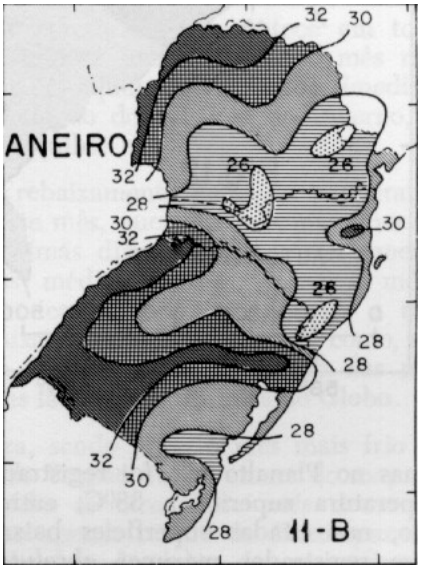
```cpp
//bayes.cpp 2024 OpenCV v3/v4
#include "procimagem.h"

int main() {
  Mat_<Vec3b> ax=imread("ax.png",1);
  Mat_<uchar> ay=imread("ay.png",0);
  if (ax.size()!=ay.size()) erro("Erro: Dimensoes diferentes");

  Mat_<float> features(ax.total(),3);
  Mat_<int> saidas(ay.total(),1);
  for (unsigned i=0; i<ax.total(); i++) {
    features(i,0)=ax(i)[0]/255.0;
    features(i,1)=ax(i)[1]/255.0;
    features(i,2)=ax(i)[2]/255.0;
    saidas(i,0)=ay(i);
  }

  Ptr<ml::NormalBayesClassifier> ind = ml::NormalBayesClassifier::create();
  ind->train(features,ml::ROW_SAMPLE,saidas);
  Mat_<Vec3b> qx=imread("f1.jpg",1);
  Mat_<uchar> qp(qx.rows,qx.cols);

  for (unsigned i=0; i<qx.total(); i++) {
    Mat_<float> query(1,3);
    query(0)=qx(i)[0]/255.0;
    query(1)=qx(i)[1]/255.0;
    query(2)=qx(i)[2]/255.0;
    int pred=ind->predict(query);
    qp(i)=pred;
  }
  imwrite("f1-bayes.png",qp);
}
```

Algoritmo 6: Classificador Normal Bayes de OpenCV v3/v4 para segmentar feijões.

```python
#naivebayes.py - grad2023
import numpy as np
from sklearn.naive_bayes import GaussianNB
import sys, cv2

#<<<<<<<<<<<<<<< main <<<<<<<<<<<<<<<<
ax=cv2.imread("ax.png",1)
ay=cv2.imread("ay.png",0)
if ax.shape[0]!=ay.shape[0] or ax.shape[1]!=ay.shape[1]:
  sys.exit("Erro: Imagens com dimensoes diferentes");

features=(ax/255).reshape((ax.shape[0]*ax.shape[1],3))
saidas  =(ay/255).reshape((ay.shape[0]*ax.shape[1],))

gnb = GaussianNB()
modelo = gnb.fit(features, saidas)

qx = cv2.imread("f1.jpg",1)
query = (qx/255).reshape((qx.shape[0]*qx.shape[1],3))
qp = modelo.predict(query)
qp = qp.reshape((qx.shape[0],qx.shape[1]))
cv2.imwrite("f1-naivebayes.png",255*qp)
```

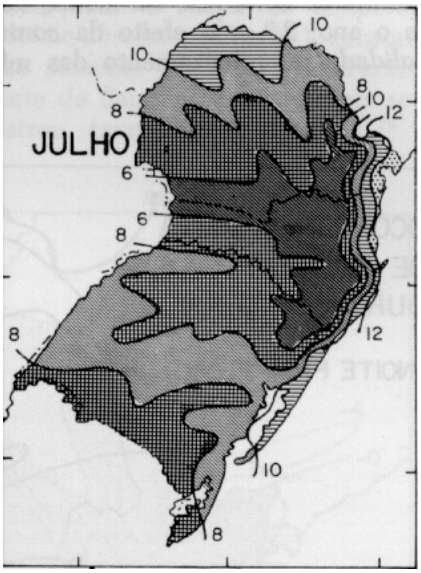
Algoritmo A1: Classificador Naive Bayes de SkLearn para segmentar feijões.

[PSI5790 aula 5. Lição de casa #2 (de 2).] Faça um programa que efetua a sequência das seguintes operações:
1) Recebe as imagens *janei.pgm* e *janei-1.pgm* como amostras de treinamento (AX, AY) e cria um filtro pelo aprendizado de máquina.
2) Aplica o filtro aprendido na imagem *julho.pgm* (QX) gerando uma imagem semelhante a *julho-p1.pgm* (QP).
3) Filtra essa imagem com filtro mediano adequado.
4) Sobrepõe a imagem filtrada à imagem original, obtendo uma imagem semelhante à *julho-c1.png*.
Dica 1: Usei filtro 3×3 e usei FlaNN (você pode usar outras técnicas).
Dica 2: Para sobrepor máscara vermelha, deixei componente *R* da imagem de saída 255.



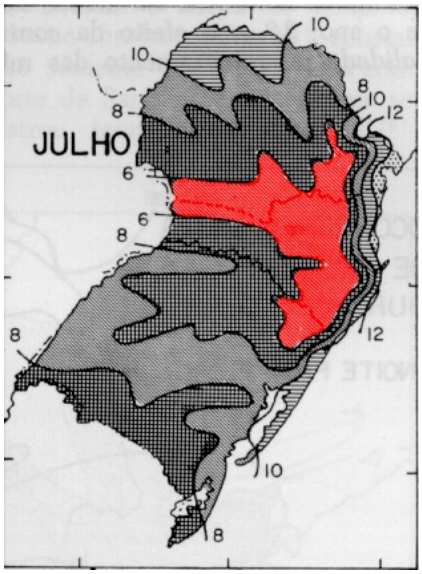
janei.pgm (AX)


janei-1.pgm (AY)



julho.pgm (QX)

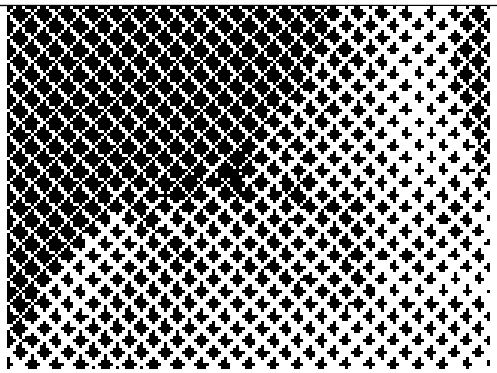

julho-p1.pgm (QP)



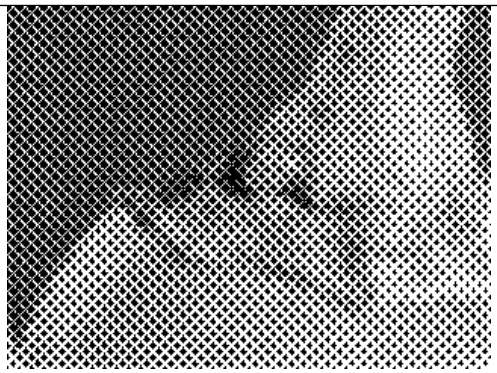
julho-c1.png

5. Curiosidades

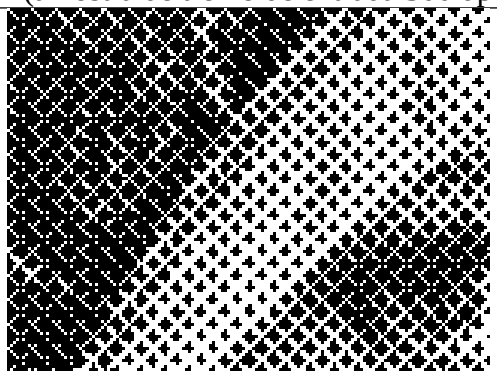
5.1 Zoom de imagem halftone:
Baixar e rodar os exemplos em: http://www.lps.usp.br/~hae/software/halfzoom/index.html



ax (amostra de treino de entrada 300 dpi)



ay (amostra de treino de saída 600 dpi)

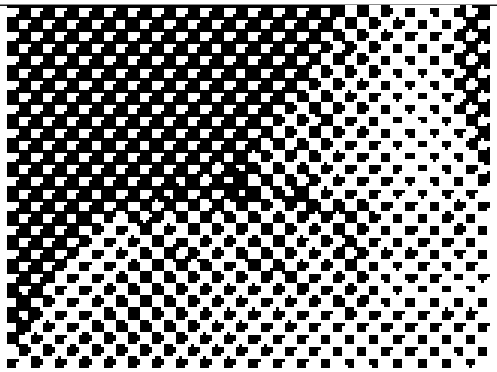
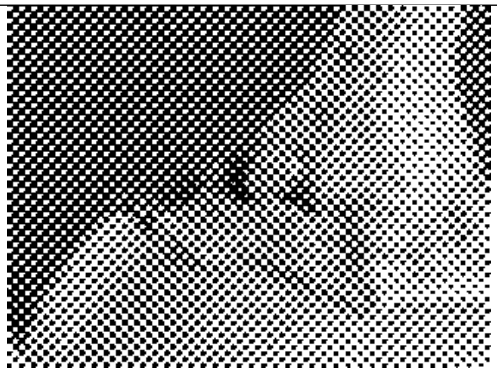
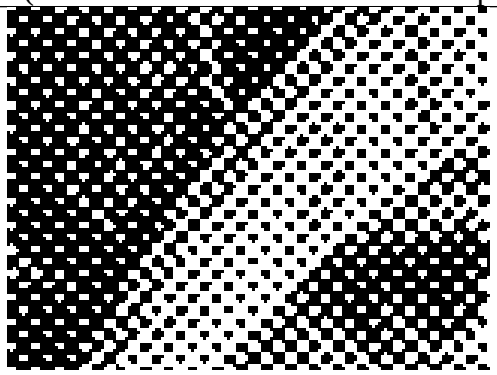
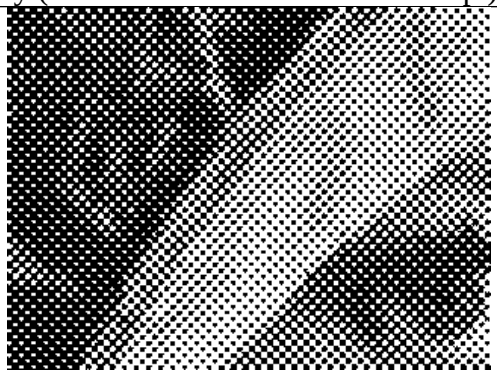


qx (imagem a processar 300 dpi)

qp (imagem processada 600 dpi)

qy (saída ideal 600 dpi)

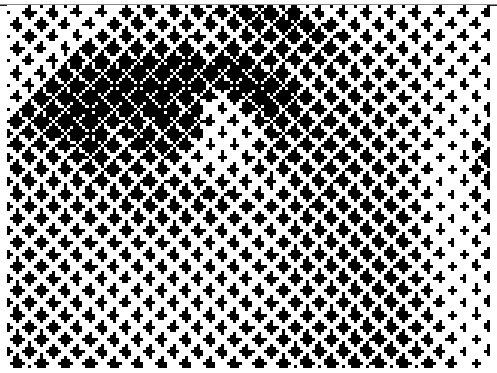
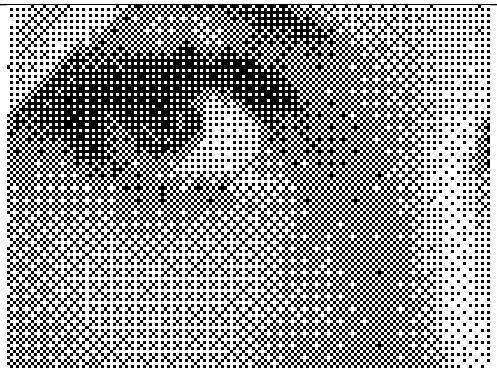
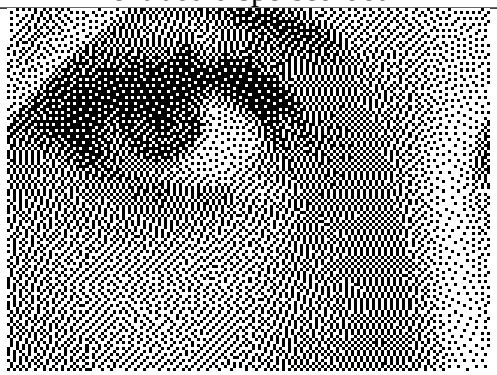
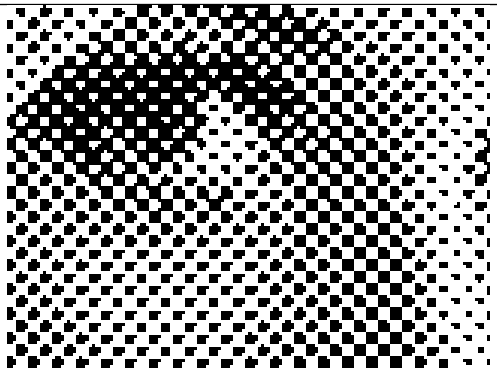
Halftone images generated by clustered-dot ordered dithering.

 ax (amostra de treino de entrada 300 dpi)	 ay (amostra de treino de saída 600 dpi)
 qx (imagem a processar 300 dpi)	 qp (imagem processada 600 dpi)
 qy (saída ideal 600 dpi)	

Halftone images generated by HP LaserJet Driver option "coarse dot."

5.2 Halftone inverso:

Baixar e rodar os exemplos em: http://www.lps.usp.br/~hae/software/invhalf/index.html

 entrada clustered-dot	 imagem processada de clustered-dot
 entrada dispersed-dot	 imagem processada de dispersed-dot
 entrada difusão de erro	 imagem processada de difusão de erro
 entrada HP laserjet coarse-dot	 imagem processada de HP laserjet coarse-dot

Referências

[Mitchell1997] Tom. M Mitchell, *Machine Learning*, WCB/McGraw-Hill, 1997

[WikiML] https://en.wikipedia.org/wiki/Machine_learning, acessado em 3 de abril de 2020.

[WikiKD] https://en.wikipedia.org/wiki/K-d_tree, acessado em 8 de abril de 2020.

[Bentley1975] Bentley, J. L. (1975). "Multidimensional binary search trees used for associative searching". Communications of the ACM. 18 (9): 509–517. doi:10.1145/361002.361007.

[wikiDecision] https://en.wikipedia.org/wiki/Decision_tree, acessado em 8 de abril de 2020.

[Kim2004] H. Y. Kim, “Binary Halftone Image Resolution Increasing by Decision-Tree Learning,” IEEE Trans. on Image Processing, vol. 13, no. 8, pp. 1136-1146, Aug. 2004.

[Kim2000] H. Y. Kim, “Binary Operator Design by k-Nearest Neighbor Learning with Application to Image Resolution Increasing,” Int. J. Imaging Systems, vol. 11, no. 5, pp. 331-339, 2000.

[PSI5790 aula 5. Fim.]

Anexo A: Boosting

Copio o trecho abaixo sobre boosting do manual do OpenCV. Para maiores detalhes, veja o próprio manual do OpenCV e artigos referenciados.

Boosting is a powerful learning concept that provides a solution to the supervised classification learning task. It combines the performance of many “weak” classifiers to produce a powerful committee [HTF01]. A weak classifier is only required to be better than chance, and thus can be very simple and computationally inexpensive. (...) Decision trees are the most popular weak classifiers used in boosting schemes. (...) The desired two-class output is encoded as -1 and +1. Different variants of boosting are known as Discrete Adaboost, Real AdaBoost, LogitBoost, and Gentle AdaBoost [FHT98]. (...) This chapter focuses only on the standard two-class Discrete AdaBoost algorithm, outlined below.

Initially the same weight is assigned to each sample (step 2). Then, a weak classifier $f_m(x)$ is trained on the weighted training data (step 3.1). Its weighted training error and scaling factor c_m is computed (step 3.2). The weights are increased for training samples that have been misclassified (step 3.3). All weights are then normalized, and the process of finding the next weak classifier continues for another M -1 times. The final classifier F(x) is the sign of the weighted sum over the individual weak classifiers (step 4).

Algoritmo AdaBoost escrito de forma informal (extraído de Wikipedia):

Form a large set of simple features
Initialize weights for training images
For T rounds

1. Normalize the weights
2. For available features from the set, train a classifier using a single feature and evaluate the training error
3. Choose the classifier with the lowest error
4. Update the weights of the training images: increase if classified wrongly by this classifier, decrease if correctly

Form the final strong classifier as the linear combination of the T classifiers (coefficient larger if training error is small)

Two-class Discrete AdaBoost Algorithm (do manual do OpenCV):

Step 1. Set N examples $(x_i, y_i)_{i = 1, ..., N}$, with $x_i \in R^K$, $y_i \in -1, +1$.
Step 2. Assign weights as $w_i = 1/N$, i = 1, ..., N.
Step 3. Repeat for m = 1, 2, ..., M :
3.1. Fit the classifier $f_m(x) \in -1, 1$, using weights w_i on the training data.
3.2. Compute $err_m = E_w[1_{(y \neq fm(x))}]$, $c_m = \log((1 - err_m)/err_m)$.
3.3. Set $w_i \Leftarrow w_i \exp[c_m 1_{(yi \neq fm(xi))}]$, i = 1, 2, ..., N, and renormalize so that $\Sigma_i w_i = 1$.
Step 4. Classify new samples x using the formula: $\text{sign}(\Sigma_{m=1}^{M} c_m f_m(x))$.

Anexo B: Ganho de informação e índice Gini

[Nota para mim: Escolher um exemplo mais complexo para ilustrar melhor os dois conceitos.]

a) Ganho de informação ou redução de entropia [Mitchell, 1997]

Definição da entropia: "the entropy of a physical system is the minimum number of bits you need to fully describe the detailed state of the system"
[http://www.science20.com/hammock_physicist/what_entropy-89730]

A entropia de um conjunto de amostras *S* contendo amostras positivas e negativas nas proporções $p_{\oplus}$ e $p_{\ominus}$ é:

$$Entropy(S) = -p_{\oplus}\log_2 p_{\oplus} - p_{\ominus}\log_2 p_{\ominus}$$

Por exemplo, a entropia assume valor 1 se $p_{\oplus} = p_{\ominus} = 0.5$ (precisa de um bit para informar se um exemplo é positivo ou negativo). Assume valor 0 se $p_{\oplus} = 0$ ou $p_{\ominus} = 0$ (não precisa de nenhum bit para informar se um exemplo é positivo ou negativo, pois só há exemplos de um único tipo). Assume valores entre 0 e 1 para as outras proporções.

O ganho de informação (ou a redução de entropia) ao dividir o conjunto de amostras *S* usando um certo atributo *A* é:

$$Gain(S,A) = Entropy(S) - \sum_{S_v \subset Divisao(S,A)} \left[\frac{|S_v|}{|S|} Entropy(S_v) \right]$$

onde Divisao(*S*,*A*) é o conjunto de todos os subconjuntos de *S* obtidos ao dividir o conjunto *S* usando o atributo *A*.

Exemplo: Considere as seguintes amostras de treinamento:

atributos		rótulos
peso	cor da pele	classificação
baixo	escura	bebê
baixo	clara	bebê
alto	clara	adulto
alto	escura	adulto

Evidentemente, é melhor usar o atributo "peso" pois o atributo "cor" não ajuda em nada a classificar os indivíduos em bebê ou adulto. Vamos verificar se é possível descobrir isto usando ganho de informação. A entropia da amostra de treinamento é:

$$Entropy(S) = -p_{\oplus}\log_2 p_{\oplus} - p_{\ominus}\log_2 p_{\ominus} = -0.5\log_2 0.5 - 0.5\log_2 0.5 = 0.5 + 0.5 = 1$$

Se escolher o atributo "peso" para dividir as amostras, o ganho de informação é um bit.

$$Gain(S, peso) = Entropy(S) - \sum_{v \in Values(A)} \left[\frac{|S_v|}{|S|} Entropy(S_v) \right] = 1 - \frac{2}{4}\times 0 - \frac{2}{4}\times 0 = 1$$

Pois, após a divisão, ficamos com um subconjunto com somente adultos (entropia 0) e um outros subconjunto somente com bebês (entropia 0). Assim, a entropia diminuiu de 1 para 0 e o ganho de informação foi de 1 bit.

Se escolher o atributo "cor da pele" para dividir as amostras, o ganho de informação é zero bit.

$$Gain(S, cor) = Entropy(S) - \sum_{v \in Values(A)} \left[\frac{|S_v|}{|S|} Entropy(S_v) \right] = 1 - \frac{2}{4}\times 1 - \frac{2}{4}\times 1 = 0$$

Após a divisão, temos dois subconjuntos com um adulto e uma criança (ambos subconjuntos têm entropia 1). Antes da subdivisão, a entropia era 1. Depois da subdivisão a entropia média ponderada continua sendo 1. Assim, o ganho de informação é nula.

Portanto, não se ganha nada ao escolher o atributo “cor da pele” para dividir as amostras. Por outro lado, escolhendo o atributo “peso” para dividir a amostra, os indivíduos ficam perfeitamente classificados em bebê ou adulto.

b) Índice Gini

https://medium.com/analytics-steps/understanding-the-gini-index-and-information-gain-in-decision-trees-ab4720518ba8
https://blog.quantinsti.com/gini-index/
https://www.upgrad.com/blog/gini-index-for-decision-trees/#:~:text=The%20Gini%20Index%20or%20Gini,feature%20that%20was%20classified%20incorrectly
https://scikit-learn.org/stable/modules/tree.html#tree-mathematical-formulation

Índice Gini é outro critério para escolha do atributo de corte.

$$Gini(atributo = valor) = 1 - \sum_{i=1}^{nclasses} (p_i)^2$$

onde p_i é a probabilidade da classificação ser *i* quando o *atributo* for *valor*. Árvore de decisão escolhe atributo com a menor média ponderada dos índices Gini. Exemplo:

atributos		rótulos
peso	cor da pele	classificação
baixo	escura	bebê
baixo	clara	bebê
alto	clara	adulto
alto	escura	adulto

Vamos calcular a média ponderada de índice Gini para peso:

P(peso=baixo)=0.5; P(peso=alto)=0.5

(baixo & bebê) → probabilidade=1
(baixo & adulto) → probabilidade=0
Gini(peso=baixo) = $1-(1^2+0^2)=0$

(alto & bebê) → probabilidade=0
(alto & adulto) → probabilidade=1
Gini(peso=alto) = $1-(0^2+1^2)=0$

Soma ponderada dos índices Gini para peso:
0.5×Gini(peso=baixo) + 0.5×Gini(peso=alto) = $0.5\times 0+0.5\times 0=0$

Agora, vamos calcular a média ponderada de índice Gini para cor da pele:

P(pele=escura)=0.5; P(pele=clara)=0.5

(escura & bebê) → probabilidade=0.5
(escura & adulto) → probabilidade=0.5
Gini(pele=escura) = $1-\left(0.5^2+0.5^2\right)=0.5$

(clara & bebê) → probabilidade=0.5
(clara & adulto) → probabilidade=0.5
Gini(clara) = $1-\left(0.5^2+0.5^2\right)=0.5$

Soma ponderada dos índices Gini para pele:
0.5×Gini(pele=escura) + 0.5×Gini(pele=clara) = $0.5\times 0.5+0.5\times 0.5=0.5$

Portanto, “peso” possui índice Gini menor que “cor da pele” e deve ser escolhida como atributo de corte.