

# Uma framework para a classificação de objetos

Felipe Blassioli

30 de Novembro de 2015

## Conteúdo

<b>1</b>	<b>Introdução</b>	<b>3</b>
1.1	Motivação . . . . .	3
1.2	Objetivo . . . . .	4
1.3	Organização do Trabalho . . . . .	4
<b>2</b>	<b>Fundamentos Teóricos</b>	<b>5</b>
2.1	Reconhecimento de Objetos . . . . .	5
2.2	Detectores de Pontos de Interesse . . . . .	7
2.2.1	Harris corner-based (Harris) . . . . .	7
2.2.2	Hessian Matrix-based (Hessian) . . . . .	10
2.2.3	Scale Invariant Harris (Harris-Laplace) . . . . .	10
2.2.4	Difference of Gaussian (DoG) . . . . .	10
2.2.5	Affine invariant Harris (Harris-affine) . . . . .	12
2.2.6	Maximally stable extremal regions (MSER) . . . . .	13
2.3	Descritores de Cacterísticas . . . . .	14
2.3.1	Scale-Invariant Feature Transform(SIFT) . . . . .	14
2.3.2	Speeded Up Robust Features (SURF) . . . . .	14
2.3.3	Local Binary Patterns (LBP) . . . . .	14
2.3.4	Granulometry Based-Descriptor (GRABED) . . . . .	15
2.4	Classificação de Objetos . . . . .	16
2.4.1	Support Vector Machines (SVM) . . . . .	16
2.4.2	Naive Bayes Classifier (Bayes) . . . . .	18
2.5	Saco de Palavras . . . . .	18
<b>3</b>	<b>Framework de treinamento e classificação</b>	<b>20</b>
3.1	Tecnologias . . . . .	20
3.2	Arquitetura . . . . .	23
3.2.1	Experimento (Experiment) . . . . .	23
3.2.2	Conjunto de Dados (Dataset) . . . . .	24
3.2.3	Extrator de Características (DescriptorExtractor) . . . . .	25
3.2.4	Classificador (Classifier) . . . . .	26
3.3	Instalação . . . . .	27
3.3.1	Instalando o OpenCV . . . . .	27
3.3.2	Instalando a Framework . . . . .	28
3.4	Uso . . . . .	28
3.4.1	Executando experimentos . . . . .	28

3.4.2	Visualizando resultados . . . . .	31
3.5	Extendendo a Framework . . . . .	36
3.5.1	Adicionando o descritor GRABED . . . . .	36
3.5.1.1	Obtendo o mapa de bordas da imagem . . . . .	36
3.5.1.2	Obter os conjuntos de famílias de EEs . . . . .	36
3.5.1.3	Obter Padrões de Espectro de cada família . .	40
3.5.1.4	Adicionando à framework . . . . .	42
3.5.2	Adicionando o descritor LBP . . . . .	43
3.5.2.1	Definição . . . . .	43
3.5.2.2	Adicionando à framework . . . . .	43
<b>4</b>	<b>Resultados e Trabalhos Futuros</b>	<b>44</b>
	<b>Referências Bibliográficas</b>	<b>45</b>

# 1 Introdução

## 1.1 Motivação

O reconhecimento de objetos em imagens é um problema antigo e conhecido sendo uma das áreas mais ativas em visão computacional[1]. Infelizmente, não há método conhecido que descreva como construir um sistema reconhecedor de objetos de interesse perfeito, o que não significa que não é possível construir sistemas de utilidade prática. São muitos os exemplos de aplicações de uso real, algumas delas são:

- Identificação de produtos via fotos tiradas pelo celular em aplicativos como *Amazon Flow* [2].
- Detecção de obstáculos em carros sem motorista como anunciado pela *Nvidia* [3, 4].
- Coleta de dados via câmeras para fins de planejamento urbano para cidades inteligentes. (ou para emitir multas devido a excesso de velocidade)

Na prática, resolver esse problema é equivalente a construir um modelo que seja capaz de identificar categorias de interesse em imagens desconhecidas. A construção de um modelo consiste na combinação de diferentes técnicas/algoritmos e pode ser feita seguindo diferentes abordagens. Por exemplo, o classificador pode ser generativo ou discriminativo, os objetos podem ser descritos através de características globais ou locais, ou ainda, devido à natureza das imagens de entrada, um algoritmo de detecção de pontos de interesse pode trazer resultados melhores do que outro.

Há muitos algoritmos publicados, bem como são muitas as ferramentas disponíveis gratuitamente que auxiliam na resolução do problema, alguns exemplos incluem bibliotecas focadas em aplicações de visão computacional que fornecem implementações de muitos algoritmos presentes na literatura: *OpenCV*[5], *CCV*[6], *VFeat*[7] e *Torch*[8], esta última em uso por gigantes do mundo da tecnologia da informação como *Google* e *Facebook*[9].

Como são muitas as ferramentas e abordagens disponíveis, é de interesse do desenvolvedor escolher a que trará melhores resultados para sua aplicação. Tal seleção deve ser feita de maneira objetiva, isto é de modo rigoroso, reproduzível e com métricas bem definidas. Em outras palavras, fazer uso das muitas ferramentas disponíveis é um modo de construir sistemas de utilidade prática, reduzindo o problema de reconhecimento de objetos a escolher qual combinação de ferramentas melhor se adequa às restrições do problema e isso deve ser feito de maneira científica, isto é via uma série de experimentos que validem as decisões tomadas.

Realizar experimentos de maneira sistemática, estabelecendo métricas e comparando objetivamente os resultados dos experimentos é uma tarefa longa, repetitiva e portanto vulnerável a erro humano. Auxiliar nesse processo é o que motivou esse trabalho.

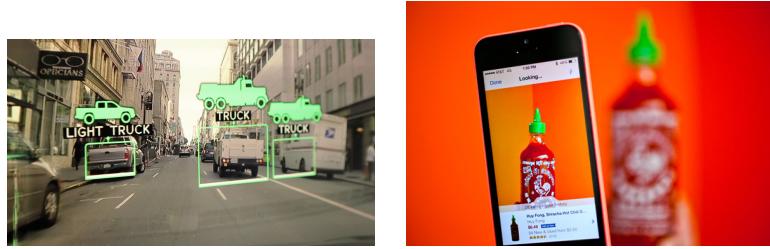


Figura 1: Aplicações de reconhecimento de objetos

## 1.2 Objetivo

Desenvolver uma *framework* que faça uso de ferramentas de processamento de imagens e aprendizado de máquina disponíveis gratuitamente para construir soluções para o problema de reconhecimento de objetos e que automatize as etapas repetitivas desse processo tais como a leitura de datasets e geração de métricas facilitando a comparação entre diferentes abordagens via análise de resultados de experimentos realizados sistematicamente a fim de encontrar a melhor solução para o problema.

## 1.3 Organização do Trabalho

Na seção 2 é feita uma revisão de conceitos importantes de visão computacional que são usados na framework. O problema de *Reconhecimento de Objetos* é apresentado na seção 2.1 junto a algoritmos conhecidos utilizados na sua resolução: *Detectores de Pontos de Interesse* (seção 2.2), *Descritores de Características* (seção 2.3) e *Classificadores* (seção 2.4); bem como técnicas conhecidas para a resolver o problema como a abordagem *Saco de Palavras* (seção 2.5).

Na seção 3 são dados todos os detalhes referentes à framework. São apresentadas as *Tecnologias* usadas em seu concebimento (seção 3.1), é descrita sua *Arquitetura* (seção 3.2), sua *Instalação* (seção 3.3), *Uso* (3.4) e como adicionar novos algoritmos à framework (seção 3.5).

Finalmente, possibilidades de *Trabalhos Futuros* são apresentados na seção 4 seguida pelas *Referências Bibliográficas*.

## 2 Fundamentos Teóricos

### 2.1 Reconhecimento de Objetos

Em linhas gerais, o problema de reconhecimento de objetos pode ser descrito como a tarefa de, dado um conjunto de categorias ou objetos de interesse e um conjunto imagens, atribuir a cada uma destas uma (ou mais) dessas categorias, como pode ser visto na Figura 2.

Resolver esse problema significa ser capaz de identificar na imagem de um objeto, isto é na projeção do objeto em relação à câmera, regiões que o caracterizem e que o distinguam de outros objetos. Em outras palavras, ser capaz de construir uma descrição objetiva de um objeto a partir de sua imagem e que esta possibilite a comparação deste com outros. Tal construção pode ser feita fazendo uso de diferentes abordagens:

- Abordagem baseada em modelos que tentam representar (aproximadamente) um objeto como uma coleção de primitivas tridimensionais, tais como: paralelepípedos, esferas, cones, cilindros, etc.
- Abordagem baseada em formas que representam um objeto de acordo com seu contorno.
- Abordagem baseada em aparências que representam um objeto de acordo com sua aparência que, no geral, é caracterizada pela visão em 2D em diferentes perspectivas do objeto.

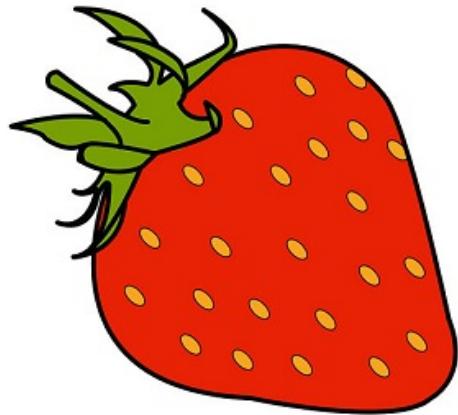
Essas abordagens ainda podem ser subdivididas em duas categorias que levam em consideração a natureza da extração de características dos objetos: globais e locais.

Características globais tentam exprimir a informação contida na imagem toda ou em todos os pixels de uma região. Tais características vão desde medidas estatísticas simples como histogramas ou valores médios de características até técnicas mais sofisticadas de redução de dimensionalidade, tais como Análise de Componentes Principais ou PCA [10], da sigla em inglês para *Principal Component Analysis*, Análise de Componentes Independentes ou ICA [11], da sigla em inglês para *Independent Component Analysis*, ou Fatorização não-negativa de matrizes ou NMF [12], da sigla em inglês para *Non-negative Matrix Factorization* que projetam os dados originais num subespaço no qual a representação destes é ótima segundo algum critério: variação mínima (PCA), independência (ICA), componentes aditivas (NMF).

Uma característica local é uma propriedade da imagem localizada num único ponto ou numa região pequena que, idealmente, seja uma propriedade distintiva da imagem do objeto. Exemplos de características locais são: cor, gradiente ou tom de cinza de um pixel ou de uma região pequena. Para a tarefa de reconhecimento de objetos tal característica deve ser invariante à ruído na imagem, mudanças de iluminação, de escala ou perspectiva, o que infelizmente não é possível na maioria dos casos devido à simplicidade das mesmas. Por esse motivo, é comum combinar várias características de um ponto ou região de interesse formando uma descrição mais complexa que é chamada de *descriptor da imagem*.



(a) abacaxi / fruta



(b) não encontrado / fruta



(c) gato / animal



(d) não encontrado / animal



(e) carro / automóvel



(f) não encontrado / automóvel

Figura 2: O conjunto de imagens a serem categorizadas consiste nas imagens *a,b,c,d,e*. O resultado esperado da classificação para os conjuntos de interesse [*abacaxi, gato, carro*] e [*fruta, animal, automóvel*] está sinalizado na legenda de cada imagem por *classe1 / classe2*.

Representar as características de maneira global possibilita a reconstrução da imagem. Enquanto, a abordagem de representação local dos dados consegue lidar de maneira melhor com oclusão parcial de objetos.

## 2.2 Detectores de Pontos de Interesse

Como a modelagem de objetos baseada em aparência (seção 2.1) trabalha sobre partes da imagem, é de grande importância determinar de modo altamente repetitivo quais regiões da imagem serão trabalhadas. Além disso, um detector de regiões deve, idealmente, retornar informações adicionais como forma, escala ou orientação de uma região de interesse. Chamamos de detector de pontos um detector de regiões que retorna uma posição exata dentro de uma imagem. Na figura 3 podem ser vistos resultados de diferentes algoritmos de detecção de pontos.

Atualmente, os detectores de região mais populares podem ser divididos em três categorias:

- Detectores baseados em cantos
- Detectores baseados em regiões (*blobs*)
- outros

Detectores baseados em cantos localizam pontos de interesse e regiões que contenham uma porção da estrutura da imagem como, por exemplo, a junção de arestas. Esses detectores não são adequados para tratar regiões uniformes ou regiões com transições suaves, isto é com pouco contraste. Detectores baseados em regiões consideram blocos de brilho uniforme como as partes mais relevantes da imagem e, portanto, são mais adequados a lidar com esses tipos de região. Outras abordagens consistem, por exemplo, em levar em consideração a entropia das regiões ou tentam encontrar regiões de interesse do mesmo modo que os seres humanos.

Alguns detectores populares são:

- Detectores baseado em pontos de Harris ou Hessian (*Harris*, *Harris-Laplace*, *Hessian-Laplace*) [13, 14]
- Detector de pontos baseado na diferença de Gaussianas (*DoG* da sigla em inglês *Difference of Gaussian*)[15]
- Detectores de regiões invariantes à transformações afins de Harris ou Hessian (*Harris-affine*)[16]
- Regiões Extremas Estáveis Maximais (*MSER* da sigla em inglês para *Maximally Stable Extremal Regions*)[17]

### 2.2.1 Harris corner-based (Harris)

A definição mais aceita de canto foi proposta em [13] e se baseia na matriz das derivadas de segunda ordem das intensidades da imagem (também conhecida como matriz de segundos momentos). Em linhas gerais, cantos são a intersecção de duas áreas e consistem num ponto onde a direção dessas duas arestas muda,

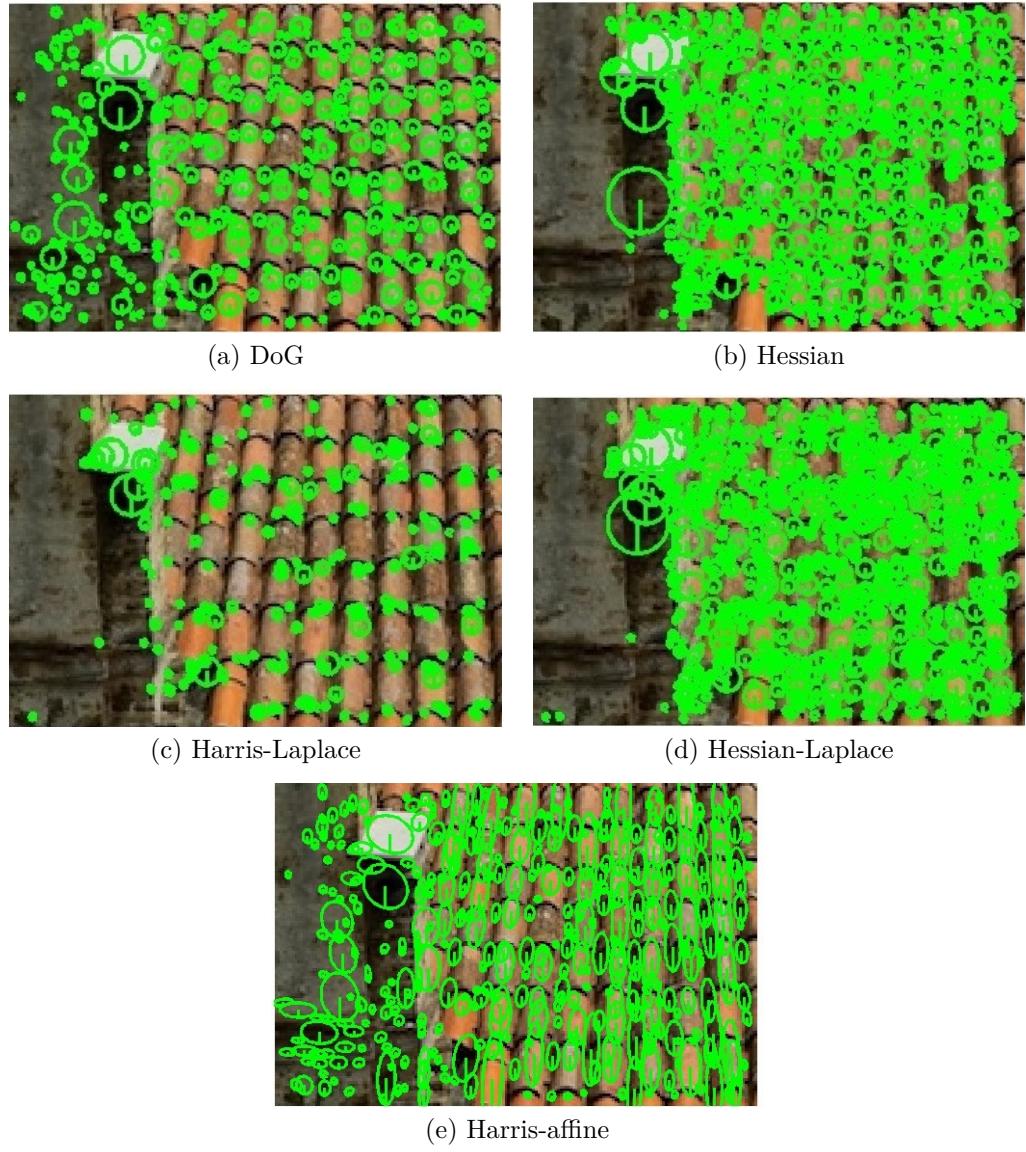


Figura 3: Regiões encontradas usando a função *vl\_covdet* da biblioteca VLFeat

portanto, o gradiente da imagem em ambas as direções tem alta variação. Desse modo, encontrar na imagem pontos de alta variação de gradiente é uma maneira de encontrar cantos, o que pode ser feito analisando as variações de intensidade devido ao deslocamento de uma janela local.

O detector Harris baseado em cantos faz precisamente isso.

Seja uma imagem  $I$  em tons de cinza. A variação de intensidade na imagem pode ser calculada deslizando uma janela  $w(x, y)$  com deslocamento  $u$  na direção  $x$  e  $v$  na direção  $y$  de acordo com a função:

$$E(u, v) = \sum_{x,y} w(x, y)[I(x + u, y + v) - I(x, y)]^2$$

onde:

- $w(x, y)$  é a janela na posição  $(x, y)$
- $I(x, y)$  é a intensidade na posição  $(x, y)$
- $I(x + u, y + v)$  é a intensidade na janela deslocada para a posição  $(x + u, y + v)$

Numa janela que contém cantos há grande variação de intensidade, logo queremos maximizar a função acima e, em particular, o termo:

$$\sum_{x,y} [I(x + u, y + v) - I(x, y)]^2$$

Usando a expansão de Taylor:

$$E(u, v) \approx \sum_{x,y} [I(x, y) + uI_x + vI_y - I(x, y)]^2$$

Expandindo o termo e cancelando adequadamente:

$$E(u, v) \approx \sum_{x,y} u^2 I_x^2 + 2uv I_x I_y + v^2 I_y^2$$

O que na forma matricial é:

$$E(u, v) \approx [u, v] \left( \sum_{x,y} w(x, y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} \right) \begin{bmatrix} u \\ v \end{bmatrix}$$

Assim, seja  $M = \sum_{x,y} w(x, y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$ , define-se o critério de seleção  $R = \det(M) - k(\text{tr}(M))^2$ , ou seja considera-se que uma janela  $w(x, y)$  contém um canto se  $R$  é maior que um pré-determinado valor.

Pontos detectados por esse algoritmo podem ser vistos na Figura 3.

### 2.2.2 Hessian Matrix-based (Hessian)

Detectores baseados na matriz de *Hessian* seguem uma ideia similar aos detectores de Harris, mas utilizam a matriz definida por:

$$M_{hessian}(p) = \begin{bmatrix} I_{xx}(p) & I_{xy}(p) \\ I_{xy}(p) & I_{yy}(p) \end{bmatrix}$$

onde  $I_{xx}$  e  $I_{yy}$  são as segundas derivadas da intensidade  $I$  da imagem no ponto  $p$  e  $I_{xy}$  é a derivada mista das direções  $x$  e  $y$  da imagem.

O critério de seleção de pontos de interesse é baseado no determinante da matriz de Hessian após o filtro de não-máximos. Esses detectores encontram *blobs* (regiões conexas de pixels que compartilham uma dada propriedade) e são invariantes a rotação.

Pontos detectados por esse algoritmo podem ser vistos na Figura 3.

### 2.2.3 Scale Invariant Harris (Harris-Laplace)

Os detectores de Harris e Hessian não são invariantes à escala. Mas é possível combiná-los com a representação no espaço escala Gaussiano a fim de obter invariança a escala. [14]

No caso dos detectores de Harris é necessário adaptar a matriz de segundos momentos para refletir a invariança a escala:

$$M = u(x, \sigma_I, \sigma_D) = \sigma_D^2 g(\sigma_I) * \begin{bmatrix} L_x^2(p, \sigma_D) & L_x L_y(p, \sigma_D) \\ L_x L_y(p, \sigma_D) & L_y^2(p, \sigma_D) \end{bmatrix}$$

, onde:

- $p = (x, y)$ .
- $*$  é o operador de convolução.
- $g(\sigma_I)$  é o núcleo Gaussiano na escala  $\sigma_I$ .
- $L(p, \sigma)$  denota a imagem suavizada por um filtro gaussiano (similar ao espaço escala Gaussiano). Como pode ser visto na Figura 4.
- $L_x(p, \sigma_D)$  e  $L_y(p, \sigma_D)$  são as derivadas nas respectivas direções na imagem suavizada de acordo com o núcleo gaussiano de escala  $\sigma_D$ .
- $\sigma_I$  determina a escala atual na qual os cantos de harris são encontrados.

O algoritmo de Harris-Laplace, então, utiliza a matriz acima para detectar cantos de Harris em múltiplas escalas e escolher automaticamente uma *escala característica* [18], obtendo assim pontos de interesse invariantes a escala.

Pontos detectados por esse algoritmo podem ser vistos na Figura 3.

### 2.2.4 Difference of Gaussian (DoG)

O detector de pontos de interesse baseado na diferença de gaussianas ou DoG [15], da sigla em inglês para *Difference of Gaussian*, é o detector de pontos de interesse do SIFT e é capaz de encontrar pontos reconhecíveis na imagem em diferentes visões e escalas.

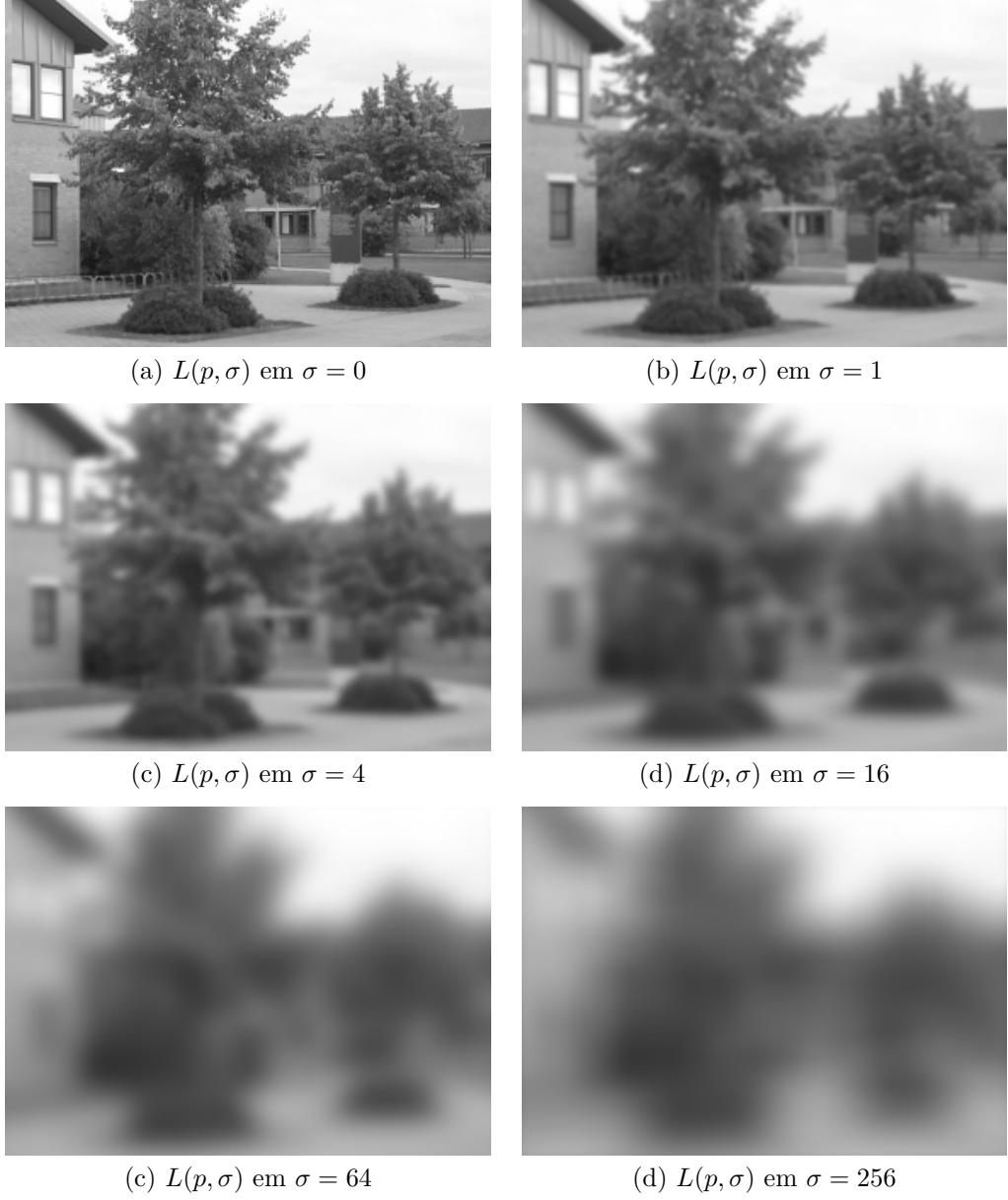


Figura 4: Representações espaço escala

Diferentemente de detectores de Harris-Laplace, esse detector não faz uso de um espaço escala Laplaciano normalizado, mas sim de uma aproximação do Laplaciano, um espaço escala construído a partir de um núcleo gaussiano:

$$L(x, y, \sigma) = g(x, y, \sigma) * I(x, y)$$

onde  $I(x, y)$  é a intensidade da imagem no ponto  $(x, y)$  e  $g(x, y, \sigma)$  é um núcleo gaussiano  $g(x, y, \sigma) = \frac{1}{2\pi\sigma^2} \exp\left(-\frac{(x^2+y^2)}{2\sigma^2}\right)$  e  $*$  é o operador de convolução.

Encontrar pontos de interesse estáveis invariantes a escala e a diferentes pontos de vista é detectar extremos locais em uma diferença de gaussianas:

$$DoG(x, y, \sigma) = L(x, y, k\sigma) - L(x, y, \sigma)$$

Além disso, elimina-se pontos de interesse gerados que possuem contraste baixo ou que estão localizados em bordas.

Pontos detectados por esse algoritmo podem ser vistos na Figura 3.

### 2.2.5 Affine invariant Harris (Harris-affine)

O objetivo de um detector invariante a transformações afins é identificar regiões na imagem relacionados por transformações afins. Mikolajczyk e Schmid [16] propuseram uma extensão à adaptação ao detector de Harris-Laplace para obter invariância a transformações afins.

A matriz de segundos momentos  $u$  no qual detectores de Harris-Laplace se baseiam pode ser definida de um modo mais genérico para regiões anisotrópicas:

$$u(p, \Sigma_I, \Sigma_D) = \det(\Sigma_D)g(\Sigma_I) * (\nabla L(p, \Sigma_D)\nabla L(p, \Sigma_D)^T)$$

onde  $\Sigma_D$  e  $\Sigma_I$  são matrizes de covariância que definem a integração e diferenciação de núcleos gaussianos de escala. Essa matriz  $u$  é idêntica à matriz da seção 2.2.3, onde  $u$  era a versão 2D-isotrópica na qual as matrizes de covariância  $\Sigma_D$  e  $\Sigma_I$  eram matrizes identidade 2x2 multiplicadas por fatores  $\sigma_D$  e  $\sigma_I$ .

O algoritmo conhecido por *Harris-Affine* parte da suposição de que a vizinhança local de cada ponto de interesse  $x$  numa imagem é uma região isotrópica ao redor de um ponto de interesse normalizado  $x^*$  que sofreu uma transformação afim. Assim, estimando os parâmetros da transformação, representado pela matriz  $U$ , é possível transformar a vizinhança local de um ponto de interesse  $x$  de volta à estrutura isotrópica  $x^*$ , isto é:

$$x^* = Ux$$

Ainda, a região invarianta afim obtida é representada pela estrutura anisotrópica normalizada da região isotrópica cuja forma estimada, tipicamente, é representada por uma elipse cuja razão dos eixos é proporcional à razão entre os autovalores da matriz de transformação. Tal estrutura anisotrópica local da imagem pode ser estimada pela inversa da matriz  $u$  calculada a partir da estrutura isotrópica, isto é:

$$x^* = u^{-\frac{1}{2}}x$$

Consequentemente, utiliza-se a concatenação de matrizes  $u^{(k)}$  otimizadas iterativamente pelo algoritmo que refina rumo ao ótimo sucessivamente uma

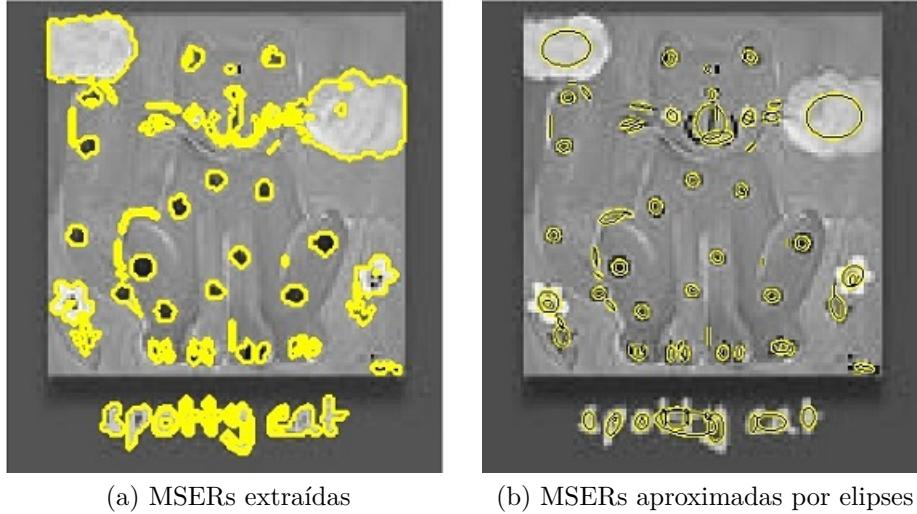


Figura 5: Uso do detector MSER presente na biblioteca VLFeat.

matriz de transformação  $U^{(0)}$  inicialmente desconhecida. O algoritmo é inicializado usando o algoritmo de Harris-Laplace para encontrar a localização aproximada do ponto  $x^{(0)}$  e da escala  $s^{(0)}$ :

$$U^{(k)} = \prod_k u^{-\frac{1}{2}(k)U^{(0)}}$$

Pontos detectados por esse algoritmo podem ser vistos na Figura 3.

#### 2.2.6 Maximally stable extremal regions (MSER)

Regiões Extremas Estáveis Maximais [17], da sigla MSER em inglês para *Maximally Stable Extremal Regions*, são componentes conexos de uma imagem limiarizada. Um algoritmo similar ao *Watershed* é aplicada às intensidades da imagem e fronteiras de segmentos que são estáveis em uma variedade grande de limiares definem uma região.

Em linhas gerais, seu funcionamento consiste em considerar todas as limiações de uma imagem em níveis de cinza. Todos os pixels cuja intensidade é menor que o limiar atual são pintados de preto (0) e os demais de branco (1). Intuitivamente, o algoritmo funciona como se estivéssemos vendo um filme cujo primeiro frame é completamente branco e à medida que vai progredindo os limiares vai crescendo e consequentemente pixels pretos e regiões que correspondem a intensidades mínimas locais aparecem. Quando essas regiões não mudam de forma para um conjunto de limiares consecutivos, tais regiões são chamadas de regiões extremas estáveis maximais (MSER).

MSERs extraídas pela função `vl_mser` podem ser vistas na Figura 5 .

## 2.3 Descritores de Cacterísticas

### 2.3.1 Scale-Invariant Feature Transform(SIFT)

A Transformada Características Invariante a Escala ou SIFT, da sigla em inglês *Scale Invariant Feature Transform* é o descritor mais utilizado atualmente, contando com suas variações. O SIFT original foi introduzido por [19] e consiste na combinação detector de pontos de interesse e descritor de características associados a esses pontos.

O detector de pontos de interesse do SIFT é o DoG (seção 2.2.4) que detecta pontos de interesse altamente repetitivos numa escala estimada. O descritor propriamente é comumente referido por SIFT-key e é invariante à rotação e consistem numa região círcular (com seu ponto central  $(x, y)$  e um raio que é a escala  $s$ ) juntamente com a orientação  $\theta$ .

Computar o descritor equivale a encontrar a orientação principal  $\theta$  dos pontos de interesse encontrados pelo *DoG*:

$$\theta(x, y) = \tan^{-1} \left( \frac{L(x, y + 1, \sigma) - L(x, y - 1, \sigma)}{L(x + 1, y, \sigma) - L(x - 1, y, \sigma)} \right)$$
$$m(x, y) = \sqrt{\left( \frac{L(x, y + 1, \sigma) - L(x, y - 1, \sigma)}{L(x + 1, y, \sigma) - L(x - 1, y, \sigma)} \right)}$$

Um histograma de 36 partes com as orientações é computado dentro de uma janela Gaussiana circular. O pico do histograma é a orientação dominante  $\theta$ . Se existir mais de uma com pelo menos 80% do valor do pico, um novo ponto de interesse é criado com a nova orientação e na mesma posição e escala do ponto original. A vizinhança do ponto de interesse é dividida em regiões 4x4, cada uma com 8 intervalos de orientação. O descritor final é a concatenação desse histogramas, o que resulta em 128 dimensões.

### 2.3.2 Speeded Up Robust Features (SURF)

O descritor chamado de Características Robustas Aceleradas ou SURF[20], da sigla em inglês de Speeded Up Robust Features, é um descritor invariante a escala e a rotação. Como o SIFT, ele consiste de um detector de pontos de interesse e de um descritor.

O detector é uma aproximação da matriz de Hessian (seção 2.2.2) que pode ser computada usando operações inteiras numa imagem integral pré-computada [21]. O determinante da matriz de Hessian é usado como medida local da variação ao redor do ponto e são escolhidos pontos cujo determinante é maximal. O descritor é baseado na resposta da ondaletas de Haar e usa 64 dimensões no vetor final.

### 2.3.3 Local Binary Patterns (LBP)

O descritor LBP proposto em [22] é bastante utilizado na classificação de texturas. A ideia do LBP é utilizar o valor do pixel central  $f(h, c)$  como limiar para comparar os pixels da vizinhança. Assim é verificado se cada pixel dos 8 vizinhos é maior ou igual ao pixel central. Este resultado consiste de 8 comparações verdadeiro ou falso. Um código é criado a partir destas comparações. Dá-se um peso a cada uma destas comparações de modo que este código de 8 comparações

gera um valor entre 0 e 255[23]. Um histograma é calculado sobre estes padrões binários e depois normalizados. O descritor final consiste na concatenação dos histogramas normalizados de todas as células da imagem.

#### 2.3.4 Granulometry Based-Descriptor (GRABED)

Descriptor baseado em granulometria aplicado ao mapa de borda ou GRABED proposto em [24], da sigla em inglês para *Granulometry Based-Descriptor Applied in map of Edges* é um descritor que faz uso de granulometria morfológica e que mede a orientação das bordas e sua extensão ao longo de uma orientação. O descritor consiste numa sequência de padrões de espectro que são resultado de uma sequência de aberturas morfológicas que fazem uso de famílias de elementos estruturantes lineares. Uma família é composta por elementos estruturantes de mesma orientação e tamanhos crescentes.



(a) Imagem original da classe  
*Faces\_easy*.

(b) Mapa de bordas.

(c) Resultados da abertura pelo EE linear de 2 pixels de tamanho.



(d) Resultados da abertura pelo EE linear de 4 pixels de tamanho.  
(e) Resultados da abertura pelo EE linear de 8 pixels de tamanho.  
(f) Resultados da abertura pelo EE linear de 16 pixels de tamanho.

Figura 6: GRABED: Sequência de aberturas morfológicas.[24]

## 2.4 Classificação de Objetos

Ao construir um classificador para reconhecimento de objetos, geralmente segue-se uma das duas filosofias: modelos generativos ou discriminativos. Formalmente, essas filosofias podem ser descritas do seguinte modo: Dada uma entrada  $x$  associada ao rótulo  $y$ , então um classificador generativo constrói o modelo de probabilidade conjunta  $p(x, y)$  e faz a classificação com base em  $p(y|x)$ , obtido fazendo uso da regra da Bayes. Em contrapartida, um classificador discriminativo modela *a posteriori*  $p(y|x)$  diretamente dos dados ou constrói um mapa das entradas para os rótulos:  $y = f(x)$ . Outro modo de encarar essa diferença de filosofias é: Modelos generativos partem da suposição de que existe algum processo oculto (geralmente aleatório) que gera os dados observados e, portanto, tentam inferir a partir desses dados os parâmetros de tal processo; Modelos discriminativos, por outro lado, não estão interessados em *como* os dados foram gerados, eles procuram apenas obter uma forma que efetivamente distingua as diferentes classes de dados.

Modelos generativos tais como a Análise de Componentes Principais ou PCA [10], da sigla em inglês para *Principal Component Analysis*, Análise de Componentes Independentes ou ICA [11], da sigla em inglês para *Independent Component Analysis*, ou Fatorização não-negativa de matrizes ou NMF [12], da sigla em inglês para *Non-negative Matrix Factorization*, tentam encontrar uma representação adequada para os dados originais por meio de uma aproximação que conserve a maior quantidade possível de informações destes. Classificadores discriminativos, por outro lado, tais como Análise de Discriminantes Lineares ou LDA, da sigla em inglês para *Linear Discriminant Analysis*[25], Máquinas de Vetores de Suporte ou SVM, da sigla em inglês para *Support Vector Machines*, ou *Boosting*[26] quando aplicadas à tarefa de classificação, encontram regiões de decisão ótimas. Assim sendo, a tarefa de atribuir um rótulo a uma entrada desconhecida num classificador generativo consiste em atribuir a entrada à classe mais provável de acordo com a verossimilhança da entrada e num classificador discriminativo a atribuição é feita com base nas regiões de decisão encontradas.

### 2.4.1 Support Vector Machines (SVM)

O classificador Máquinas de Vetores de Suporte ou SVM, da sigla em inglês de Support Vector Machines, foi proposto em [27] e é o classificador mais utilizado em problemas reconhecimento de objetos [28]. SVM são, em sua forma mais simples, classificadores lineares binários supervisionados. SVM constrói um hiperplano em um espaço multidimensional que divide as duas classes do conjunto de treinamento. O hiperplano é construído de forma a maximizar a distância aos pontos dos dados de treinamento mais próximos de ambas as classes, tal distância é chamada de **margem** e, portanto, o hiperplano ótimo encontrado pelo SVM é aquele que *maximiza* a margem em relação aos dados de treinamento, como pode ser visto na Figura 7.

#### Encontrando o hiperplano ótimo

Formalmente, define-se um hiperplano pela seguinte equação:

$$f(x) = \beta_0 + \beta^T x$$

onde  $\beta$  é conhecido como vetor de pesos e  $\beta_0$  é chamado de viés. Desse modo, escalando  $\beta$  e  $\beta_0$  o hiperplano ótimo pode ser representado de uma quantidade

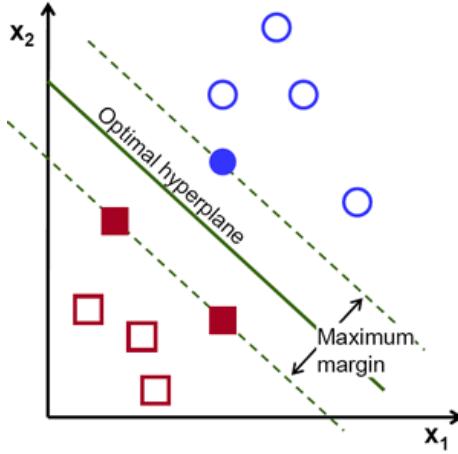


Figura 7: SVM: Hiperplano ótimo

infinita de modos. Assim, de todas as representação possíveis escolheremos:

$$|\beta_0 + \beta^T x| = 1$$

onde  $x$  representa as amostras de treinamento mais próximas do hiperplano, e que, em geral, são chamados de **vetores de suporte**. Tal representação é conhecida como **hiperplano canônico**.

A distância de um ponto  $x$  ao hiperplano descrito por  $(\beta, \beta_0)$  é dado por:

$$dist = \frac{|\beta_0 + \beta^T x|}{\|\beta\|}$$

E para o caso particular do hiperplano canônico obtemos a distância dos vetores de suporte do seguinte modo:

$$dist = \frac{|\beta_0 + \beta^T x|}{\|\beta\|} = \frac{1}{\|\beta\|}$$

E a margem  $M$  a ser maximizada é duas vezes a distância dos vetores de suporte:

$$M = \frac{2}{\|\beta\|}$$

O problema de maximizar  $M$  é equivalente ao problema de minimizar a função  $L(\beta)$  sujeita à restrições que garantem que o hiperplano classifique corretamente todas as amostras de treinamento  $x_i$ , isso é:

$$\min_{\beta, \beta_0} L(\beta) = \frac{1}{2} \|\beta\|^2 \text{ sujeito à } y_i(\beta^T x_i + \beta_0) \geq 1 \forall i$$

onde  $y_i$  consiste nos rótulos das amostras de treinamento  $x_i$ . Assim, encontrar o vetor de pesos  $\beta$  e o viés  $\beta_0$  do hiperplano ótimo é equivalente a resolver esse problema de optimização lagrangiana.

Caso os dados não sejam linearmente separáveis, é necessário usar o chamado truque do núcleo que utiliza uma função especial, a função núcleo, que mapeia pontos dos dados em um espaço de dimensão superior onde os dados são linearmente separáveis. A função núcleo depende da natureza do problema.

Há estudos sobre modos de encontrar automaticamente a função núcleo mais adequada [29].

#### 2.4.2 Naive Bayes Classifier (Bayes)

O classificador ingênuo de Bayes, do termo em inglês *Naive Bayes* é um classificador generativo multiclasse derivado de uma simples aplicação da regra de Bayes que assume as características são independentes, uma suposição ingênua já que este raramente é o caso (por exemplo: encontrar um olho normalmente implica que outro olho está numa região próxima).

A regra de Bayes é usada da seguinte maneira:

$$p(\text{objeto}|\text{características}) = \frac{p(\text{características}|\text{objeto})p(\text{objeto})}{p(\text{características})}$$

O que na prática significa que é computada alguma evidência e depois decidido qual objeto foi causador dela. Como tal evidência é a mesma para todos os objetos, esse termo pode ser desconsiderado. Além disso o numerador é a probabilidade conjunta  $p(\text{objeto}, \text{características})$  que pode ser encontrada a partir da definição de probabilidade condicional e da suposição de independência das características:

$$p(\text{objeto}, \text{características}) = p(\text{objeto}) \prod_c^{caractéristicas} p(c|\text{objeto})$$

Assim, para classificar objetos é necessário construir modelos (i.e definir as probabilidades) para os objetos de interesse.

## 2.5 Saco de Palavras

O saco de palavras, em inglês *bag-of-words* ou BoW, é uma técnica proposta em [30] para ser utilizada na área de reconhecimento de objetos. Ela é derivada da categorização de textos [31]. Posteriormente foi utilizada em classificação de texturas [32, 33]. Tornou-se uma das abordagens mais eficazes para reconhecimento de objetos.

A ideia de que textos podem ser classificados a partir da frequência de palavras é a base da técnica de classificação de textos usando saco de palavras. Para cada categoria de interesse presente na coleção de textos, cria-se um modelo que consiste no vocabulário específico da categoria usando um dicionário criado a partir do conjunto de todas as palavras presentes na coleção. Por exemplo, espera-se que textos com informações sobre gatos tenham uma frequência de palavras bastante diferente de textos sobre engenharia civil. Assim, representando cada texto da coleção pela frequência normalizada das palavras neste, um classificador é treinado para criar os modelos de diferentes categorias. Em posse dos modelos das categorias, a tarefa de atribuir uma categoria a um texto não presente na coleção, resume-se a analisar a frequência das palavras nele presentes e encontrar o modelo com frequências mais similar ao texto ainda não categorizado. Isto é, textos com frequência maior de palavras ligadas a gatos serão categorizados como *texto sobre gatos* e textos com frequência maior de palavras ligadas a engenharia civil como *texto sobre engenharia civil*.

A adaptação dessa técnica para a categorização de objetos consiste na construção de um dicionário visual que consiste num conjunto de palavras visuais. Cada imagem do conjunto de treinamento é então descrita como um conjunto de palavras visuais, assim como um texto consiste num conjunto de palavras. Esses conjuntos de palavras visuais servirão de entrada a um classificador que irá construir os modelos das categorias. Em posse desses modelos, ao se considerar uma imagem desconhecida, primeiro obtém-se o conjunto de palavras visuais associadas a esta imagem e depois, analisando a frequência dessas palavras, encontra-se o modelo que melhor descreve essa imagem.

Mais formalmente, o modelo do saco de palavras pode ser definido da seguinte maneira. Dado um conjunto de treinamento  $D$  contendo  $n$  imagens representadas por  $D = d_1, d_2, d_3, \dots, d_n$ , onde  $d$  são as características visuais extraídas. Um algoritmo de aprendizado não supervisionado, como o k-médias, é utilizado para agrupar  $D$  em um número fixo de categorias  $W$  representadas por  $W = w_1, w_2, w_3, \dots, w_v$  onde  $v$  é o número de aglomerações. Pode-se sumarizar os dados um uma tabela de co-ocorrências de tamanho  $v \times n$  com contagens  $N_{ij} = n(w_i, d_j)$ , onde  $n(w_i, d_j)$  é o número de ocorrências da aglomeração  $w_i$  na imagem  $d_j$  [34].

### 3 Framework de treinamento e classificação

A framework consiste num pacote Python que fornece classes básicas para a criação e execução de experimentos e de uma aplicação web responsável por visualizar resultados e facilitar sua análise.

Por exemplo, suponha que temos fotos de chinelos, botas e tênis e queremos desenvolver um reconhecedor de calçados. Além disso, queremos fazer uso de algoritmos populares como SIFT , SVM e MSERs, mas não sabemos qual combinação de algoritmos e quais parâmetros trarão os melhores resultados.

Para este fim, foram feitos os seguintes experimentos, todos usando o mesmo conjunto de treinamento e de testes:

- Experimento 1: MSER + SIFT + SVM usando kernel RBF com  $\gamma = 0.5$
- Experimento 2: MSER + SIFT + SVM usando kernel RBF com  $\gamma = 0.7$
- Experimento 3: DoG + SIFT + SVM usando kernel RBF com  $\gamma = 0.5$
- Experimento 4: DoG + SIFT + SVM usando kernel RBF com  $\gamma = 0.7$

Diante dessa situação a framework provê meios que facilitam a visualização e comparação de resultados com base em métricas conhecidas como (precisão, acurácia, F1 ...), além de permitir a execução de experimentos que utilizam diferentes conjuntos de treinamento e teste facilitando a identificação de problemas como *overfitting*.

As tecnologias usadas são descritas na seção 3.1, a arquitetura geral da *framework* é apresentada na seção 3.2, instruções de instalação e uso estão presentes nas seções 3.3 e 3.4, e o processo de acrescentar novos algoritmos à *framework* está na seção 3.5.

#### 3.1 Tecnologias

##### Python

Python foi a linguagem de programação escolhida pelos seguintes motivos:

- Instalada por padrão em distribuições linux populares como Ubuntu.
- Possui biblioteca madura para processamento numérico: *numpy*
- De uso comum na comunidade científica através de pacotes populares como *matplotlib* e *scikit-learn*[35].
- Possui interface para uso da biblioteca popular de visão computacional *OpenCV*.
- Possui interface de uso simples para comunicação entre código C e Python.

Em outras palavras, a adoção da ferramenta não seria difícil para pessoas familiarizadas com essas duas últimas bibliotecas, que são amplamente utilizadas pela comunidade científica, e o fato do Python já vir instalado no sistema facilita ainda mais essa adoção.

Além disso, é importante poder fazer a comunicação com código C quando necessário, pois existem muitas bibliotecas escritas em C/C++ que implementam algoritmos conhecidos de visão computacional, algumas delas são: *VLFeat*[7] e *CCV*[6].

### **OpenCV**

OpenCV é uma biblioteca de código aberto cujo objetivo é prover uma infraestrutura para aplicações de visão computacional[5]. Ela é escrita em C/C++ e por padrão disponibiliza uma interface que permite utilizá-la via código Python.

Ela foi escolhida, pois é de fácil uso e disponiliza muitos algoritmos conhecidos que são disponibilizados para a realização de experimentos, tais como (não se trata de uma lista completa):

- Descritores de Características: SIFT, SURF, ORB, BRISK.
- Detectores de Pontos de Interesse: MSER.
- Classificadores: SVM, Normal Bayes, Ada Boost, KNN.
- Outros algoritmos: KMeans, KD trees, FLANN matching.

Alguns dos algoritmos disponibilizadas pelo *OpenCV* podem ser vistos em ação na figura 8.

### **Flask**

Foi decidido desenvolver o visualizador de resultados como uma aplicação web para que ficasse mais fácil, no futuro, evoluí-la para uma plataforma mais poderosa que pudesse ficar disponível online.

Além disso, não sendo uma aplicação desktop é possível acessar os resultados dos experimentos via uma URL, não sendo necessário estar na frente do computador no qual foi instalada a framework.

Flask[36] foi a *microframework* escolhida para desenvolvimento da aplicação web devido a sua simplicidade. Em poucas linhas de código é possível disponibilizar uma página web que responde na URL <http://localhost/> :

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

### **ReactJS**

ReactJS[37] é uma biblioteca javascript desenvolvida pelo *Facebook* para a criação de interfaces web. Ela foi escolhida devido ao seu foco na criação de interfaces dinâmicas, popularidade e consequente riqueza de pacotes para tarefas importantes como a geração de gráficos.



Figura 8: OpenCV: Detecção de Pontos de Interesse, Limiarização e Detecção de bordas.

## 3.2 Arquitetura

É abstraída a noção de experimento de reconhecimento de objetos como a combinação de três noções principais:

- Conjunto de Dados
- Extrator de Características (Detector + Descritor de Pontos de Interesse)
- Classificador

O conjunto de dados nada mais é que o conjunto de imagens com que se pretende trabalhar (classificar ou usar para treinamento) e, no caso de classificadores supervisionados, devem existir rótulos associados às imagens. O extrator de características consiste no conjunto de algoritmos detectores de pontos de interesse (seção 2.2) e descritores de características (seção 2.3) e é o responsável por transformar as imagens num vetor que, por sua vez, é entregue ao classificador para que ele seja treinado ou associe um rótulo à imagem que originou o vetor. Essas noções foram abstraídas nas classes *Experiment* (seção 3.2.1), *Dataset* (seção 3.2.2), *Classifier* (seção 3.2.4) e *FeatureExtractor* (seção 3.2.3). E sua relação pode ser visualizada na figura 9.

### EXPERIMENT



Figura 9: Principais componentes da *framework*

#### 3.2.1 Experimento (Experiment)

A noção de experimento de reconhecimento de objetos foi abstraída na classe *Experiment*, o que significa que para a framework um experimento consiste num objeto que possui a ele associado um classificador, um extrator de características e que disponibiliza o método *run* que recebe como parâmetros os conjuntos de dados de treinamento e de testes (objetos da classe *Dataset*).

A implementação padrão da framework possui duas propriedades principais *classifier* e *descriptor\_extractor* e o método *run* que:

1. Treina a instância de objeto classificador presente na propriedade *classifier* usando o dataset de treinamento.
2. Obtém o vetor de características de cada imagem presente no dataset de testes usando o extrator presente na propriedade *descriptor\_extractor*, guardando-os num vetor.
3. Obtém os rótulos associados a cada amostra do dataset de testes, fazendo chamada ao método *predict\_all* do classificador presente na propriedade *classifier*. passando como parâmetro o vetor de vetores de características obtidos no passo anterior.

- Guarda o resultado (imagens e rótulos) da classificação num objeto interno (*ResultWrapper*) que é responsável por calcular medidas estatísticas a partir da comparação dos rótulos associados aos datasets, bem como tornar os resultados serializáveis para visualização e consulta futura.

#### **Exemplo de experimento**

Segue abaixo um exemplo de script com dois experimentos que utilizam *QEF* (Quantização espacial flexível)[24] junto ao extrator de características *GRABED* (seção 2.3.4) e *SVM* (seção 2.4.1) como classificador aplicados a diferentes conjuntos de treinamento.

```
from senjo.experiments import Experiment
from senjo.algorithms import QEF, GRABED, SVM

class QEFGABEDExperiment(Experiment):
    @property
    def classifier(self):
        return SVM()

    @property
    def descriptor_extractor(self):
        return QEF(GRABED(), shape=(3,1))

exp = QEFGABEDExperiment('instance/data/data01', 'instance/data/data01-tr')
exp.run().save("QEF-GRABEDResultado1.json")

exp = QEFGABEDExperiment('instance/data/data02', 'instance/data/data01-tr')
exp.run().save("QEF-GRABEDResultado2.json")
```

Os resultados ficam disponíveis nos arquivos de formato JSON:

- QEF-GRABED-resultado1.json
- QEF-GRABED-resultado2.json

Que podem ser visualizados e comparados (veja a seção 3.4.2).

#### **3.2.2 Conjunto de Dados (Dataset)**

Essa classe abstrai o *Conjunto de Dados* cuja função é descrever onde estão localizadas as imagens e quaisquer dados associados a elas que possam ser úteis à classificação ou à comparação de experimentos.

Tecnicamente, em termos da linguagem Python, um *Dataset* é um conjunto iterável de tuplas da forma (*caminho\_da\_imagem*, *lista\_de\_rótulos*). Por exemplo, um *Dataset* válido é:

```
# Rótulos: 1 - Ave , 2 - Mamífero
dataset_animais = [
    ('pato.png', [1]),
    ('penguin.png', [1]),
    ('macaco.png', [2])
    ('zebra.png', [2])
]
```

Note que se trata de uma lista de rótulos e não de um rótulo único.

A implementação padrão do conjunto de dados, no entanto, recebe um caminho de diretório (raiz) e considera cada subdiretório como uma classe de modo que o nome do diretório é associado a toda imagem contida naquele diretório. Isso foi feito por conveniência: para executar um experimento basta especificar onde estão as imagens que queremos usar.

Por exemplo, o código abaixo usa a implementação padrão de *Dataset*:

```
from senjo.experiments import Dataset

dataset = Dataset('instance/data/dataset-01/')
for img_path, label_list in dataset:
    print img_path, label_list
```

Para gerar a partir do diretório *instance/data/dataset-01/* que contém dois subdiretórios *01* e *02* cada um com três imagens, o equivalente à:

```
dataset = [
    ('dataset-01/maçã.jpg', ['1']),
    ('dataset-01/abacaxi.jpg', ['1']),
    ('dataset-01/tomate.jpg', ['1']),
    ('dataset-01/nabo.jpg', ['2']),
    ('dataset-01/rúcula.jpg', ['2']),
    ('dataset-01/cenoura.jpg', ['2'])
]
```

A implementação padrão de *Experiment* faz uso no método *run* da classe *senjo.experiments.Dataset* para carregar os conjuntos de dados de treinamento e testes do experimento a ser executado.

### 3.2.3 Extrator de Características (DescriptorExtractor)

Consiste num objeto que disponibiliza o método *compute* que recebe o caminho de uma imagem e retorna um vetor de características.

Segue um exemplo que utiliza do extrator SIFT implementado na biblioteca OpenCV:

```
import cv2
import numpy as np

class OpenCVSIFT(object):
    def compute(self, image_path):
        img = cv2.imread(image_path, cv2.CV_LOAD_IMAGE_GRAYSCALE)
        kp = cv2.FeatureDetector_create('SIFT').detect(img)
        kp, desc = cv2.DescriptorExtractor_create('SIFT').compute(img,kp)

    return np.float32(desc)
```

Seu uso é como no script a seguir:

```
descriptor_extractor = OpenCVSIFT()
desc = descriptor_extractor.compute('bota.jpg')
# Imprime as dimensões do vetor extraído
print desc.shape, desc.dtype, type(desc)
# Imprime o conteúdo do vetor
print desc

# Vetor do tipo numpy.ndarray
# 442 pontos de interesse com 128 dimensões cada
(442, 128) float32 <type 'numpy.ndarray'>
[[ 8.  4.  0. ... , 0.  0.  4.]
 [ 0.  0.  0. ... , 6.  1.  1.]
 [ 1.  0.  0. ... , 4.  2.  73.]
 ...
 [ 2.  14.  122. ... , 0.  0.  0.]
 [ 1.  9.  35. ... , 0.  0.  0.]
 [ 0.  0.  0. ... , 0.  0.  0.]]
```

Objetos que implementam o método *compute* como descrito acima são usados como extratores de características nos experimentos.

É fornecida uma classe base de conveniência que já faz a leitura da imagem (obtendo o vetor a ser processado) e transforma a imagem em tons de cinza quando necessário. A classe é *senjo.algorithms.BaseDescriptorExtractor*.

Veja a seção 3.5 para a adição dos descritores GRABED e LBP à *framework*.

### 3.2.4 Classificador (Classifier)

Consiste num objeto que disponibiliza os métodos:

- *train*: recebe um vetor de vetores de características e um vetor de rótulos.
- *predict*: recebe um vetor de características e retorna um rótulo.
- *predict\_all*: recebe um vetor de vetores de características e retorna um vetor de rótulos

Usar o SVM fornecido pelo OpenCV é tão simples quanto escrever uma classe wrapper para ele (em particular porque foram adotados os mesmos nomes de métodos):

```
class SVM(object):
    def __init__(self, C=1, gamma=0.5):
        self.svm = cv2.SVM()
        self.svm_params = dict(
            kernel_type = cv2.SVM_RBF,
            svm_type = cv2.SVM_C_SVC,
            C=C,
            gamma=gamma
        )
```

```

def train(self, training_data, labels):
    return self.svm.train(training_data, labels, params=self.svm_params)

def predict(self, *args, **kwargs):
    return self.svm.predict(*args, **kwargs)

def predict_all(self, *args, **kwargs):
    return self.svm.predict_all(*args, **kwargs)

```

### 3.3 Instalação

Para instalar a framework é necessário que esteja instalado o OpenCV 2.4.11 e Python 2.7 na máquina alvo. A instalação do OpenCV já exige a instalação do Python e pacotes como o Numpy, portanto, é suficiente instalar o OpenCV e seguir as instruções na seção 3.3.2 para fazer uso da mesma.

#### 3.3.1 Instalando o OpenCV

##### Instalando as dependências

De acordo com o guia de instalação do OpenCV a lista de pacotes necessários a sua instalação são:

- GCC 4.4.x ou acima
- CMake 2.6 ou acima
- Git
- GTK+2.x ou acima, incluindo headers (libgtk2.0-dev)
- pkg-config
- Python 2.6 or later and Numpy 1.5 or later with developer packages (python-dev, python-numpy)
- ffmpeg ou libav de desenvolvimento: libavcodec-dev, libavformat-dev, libswscale-dev
- **(opcional)** libtbb2 libtbb-dev
- **(opcional)** libdc1394 2.x
- **(opcional)** libjpeg-dev, libpng-dev, libtiff-dev, libjasper-dev, libdc1394-22-dev

Instalando tais pacotes no ubuntu/mint:

```

# [compiler]
$ sudo apt-get install build-essential
# [required]
$ sudo apt-get install cmake git libgtk2.0-dev
$ sudo apt-get install pkg-config libavcodec-dev libavformat-dev libswscale-dev
# [optional]
$ sudo apt-get install python-dev python-numpy libtbb2 libtbb-dev libjpeg-dev
$ sudo apt-get install libpng-dev libtiff-dev libjasper-dev libdc1394-22-dev

```

## Compilando e instalando

Para obter o código-fonte acesse <http://sourceforge.net/projects/opencvlibrary/files/opencv-unix/2.4.11/>. Depois, extraia-o para o diretório *opencv*.

E assumindo que o código-fonte esteja em */opencv* para compilá-lo, execute:

```
$ cd ~/opencv  
$ mkdir release  
$ cd release  
$ cmake -D CMAKE_BUILD_TYPE=RELEASE -D CMAKE_INSTALL_PREFIX=/usr/local ..
```

Depois instale no sistema:

```
$ make  
$ sudo make install
```

### 3.3.2 Instalando a Framework

A framework consiste num pacote Python que utiliza o setuptools para sua instalação. Para instalá-lo no linux, execute no terminal:

```
$ wget https://bootstrap.pypa.io/ez_setup.py -O - | sudo python
```

Depois disso, para instalar a framework, obtenha seu código-fonte, vá até o diretório do arquivo *setup.py* e execute:

```
$ python setup.py install
```

## 3.4 Uso

### 3.4.1 Executando experimentos

Para executar um experimento é necessário escrever um script que defina os experimentos herdando de uma das duas classes expostas pelo pacote da *framework*: *senjo.experiments.Experiment* e *senjo.experiments.BOWExperiment*. Essas classes permitem que se configure os experimentos, definindo os seguintes itens:

1. Datasets de treinamento e de testes
2. Extrator de características
3. Classificador

A única diferença entre as classes *BOWExperiment* e *Experiment* é que a primeira utiliza a abordagem saco de palavras (seção 2.5), utilizando como algoritmo de clusterização o *KMeans* para criar um dicionário visual utilizando o extrator de características especificado e depois treinando o classificador especificado com esse dicionário.

Por padrão, um dataset consiste num diretório cujos nomes dos subdiretórios são números de 1 até *n-1* onde *n* é o número de classes do experimento. As imagens presentes num subdiretório são rotuladas de acordo com o nome deste.

Eis a lista de algoritmos disponíveis no pacote:

### Extratores de Características

- *senjo.algorithms.SIFT*: Scale-invariant feature transform.
- *senjo.algorithms.DSIFT*: Dense SIFT.
- *senjo.algorithms.GRABED*: Granulometry Based descriptor applied to edges map.
- *senjo.algorithms.SURF*: Speeded Up Robust Features.

### Classificadores

- *senjo.algorithms.SVM*: Support Vector Machine

Assim, para executar dois experimentos que:

1. Utilizam o dataset de treinamento *instance/data/data02* e o de teste *instance/data/data02-tst*.
2. Utilizam o extrator de características GRABED.
3. Utilizam o classificador SVM.

Basta escrever o script *run\_experiments.py* como segue:

```
from senjo.experiments import Experiment, BOWExperiment
from senjo.algorithms import GRABED, SVM
class GRABEDEExperiment(Experiment):
    @property
    def classifier(self):
        return SVM()

    @property
    def descriptor_extractor(self):
        return GRABED()

class BOWGRABEDEExperiment(BOWExperiment):
    @property
    def descriptor_extractor(self):
        return GRABED()

res = GRABEDEExperiment('instance/data/data02', 'instance/data/data02-tst', name='GRABED')
res.save('result-grabed.json')

res = BOWGRABEDEExperiment('instance/data/data02', 'instance/data/data02-tst', name='BOW-GRABED')
res.save('result-bow-grabed.json')
```

E para executá-los, vá ao terminal e rode o script:

```
python run_experiments.py
```

Os resultados serão salvos nos arquivos *result-grabed.json* e *result-bow-grabed.json*.

### Experimento mais interessante

Um exemplo de experimentos mais interessante e que demonstra parte do poder da ferramenta é o seguinte: Vamos gerar uma série de experimentos que variam apenas os parâmetros  $C$  e  $\gamma$  do classificador SVM. Vamos variá-los de acordo com uma escala logarítmica.  $C$  irá de  $10^{-2}$  até  $10^{10}$  e  $\gamma$  irá de  $10^{-9}$  até  $10^3$ .

```
from senjo.experiments import Experiment
from senjo.algorithms import GRABED, SVM

import numpy as np
def experiments_generator(training_dataset, test_dataset):

    # list of tuples ( C, gamma )
    gamma_range = np.logspace(-9,3,13)
    C_range = np.logspace(-2,10,13)

    for C, gamma in zip( C_range, gamma_range )+[(1,0.5)]:
        class _Experiment(Experiment):
            @property
            def classifier(self):
                return SVM(C=C, gamma=gamma)
            @property
            def descriptor_extractor(self):
                return GRABED()
        print 'SVM with parameters', C, gamma
        yield _Experiment(training_dataset, test_dataset)

import os
EXPERIMENTS_DIR = 'instance/experiments'
EXPERIMENTS = experiments_generator('instance/data/data02', 'instance/data/data02-tr')
for i, exp in enumerate( EXPERIMENTS ):
    experiment_name = 'SVM-\%03d' \% i

    output_dir = os.path.join(EXPERIMENTS_DIR, experiment_name)
    res = exp.run(name=experiment_name, output_dir=output_dir)

    filename = 'result-\%s.json' \% experiment_name
    result_filepath = os.path.join(EXPERIMENTS_DIR, filename)
    res.save(result_filepath)
```

Executar o script acima gerará 13 arquivos de resultados que serão visualizados na seção a seguir.

### 3.4.2 Visualizando resultados

Para visualizar os resultados de experimentos, é necessário executar a aplicação web disponível no pacote da *framework*.

#### Instalando os requisitos

Os requisitos estão no arquivo *requirements.txt*. Caso o pip esteja instalado basta executar:

```
pip install -r requirements.txt
```

#### Usando o visualizador

Por padrão, o visualizador de resultados, espera que exista o diretório *instance/* e dentro dele dois subdiretórios: *data/* e *experiments/*.

O diretório *data/* possui as imagens dos datasets, onde cada diretório é um dataset diferente. O diretório *experiments* deve conter arquivos *.json* que são os resultados a serem visualizados.

Um exemplo de estrutura de diretórios esperado é:

```
data/
|-- data01/
| |-- 01/
| | |-- 001-0007-138.jpg
| | |-- 001-0019-012.jpg
| | |-- 001-0122-240.jpg
| |-- 02/
|   |-- 004-0250-394.jpg
|   |-- 004-0411-028.jpg
|   |-- 004-0411-056.jpg
|-- data01-tr/
|-- 01/
| |-- 001-0007-138.jpg
| |-- 001-0019-012.jpg
|-- 001-0122-240.jpg
|-- 02/
    |-- A46-0406-034.jpg
    |-- A52-0005-266.jpg
    |- A52-0005-810.jpg

experiments/
|-- result-SVM-000.json
|-- result-SVM-001.json
|-- result-SVM-002.json
|-- result-SVM-003.json
|-- result-SVM-004.json
|-- result-SVM-005.json
|-- result-SVM-006.json
|-- result-SVM-007.json
|-- result-SVM-008.json
|-- result-SVM-009.json
|-- result-SVM-010.json
|-- result-SVM-011.json
```

```

|-- result-SVM-012.json
|-- result-SVM-013.json

```

Assim sendo, para iniciar o visualizador, execute:

```

python sena/runserver.py
# Output esperado:
# * Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
# * Restarting with stat

```

E agora os resultados já podem ser visualizados acessando no browser: <http://127.0.0.1:5000/>

O visualizador pode ser visto na Figura 10 e dispõe das seguintes funcionalidades:

- Detalhes de Experimentos

- Panorama do experimento: Métricas e Tabela de Confusão como média das classes. (Figura 11)
- Panorama de Classes: Métricas e Tabela de Confusão comparando classe a classe. (Figura 12)
- Detalhes das Classes: Métricas, Tabela de Confusão e Resultado da Identificação de cada imagem na classe. (Figura 13)

- Análise de Experimentos

- Tabela Experimentos x Métricas. (Figura 14)
- Gráficos Experimentos x Métricas. (Figura 15)

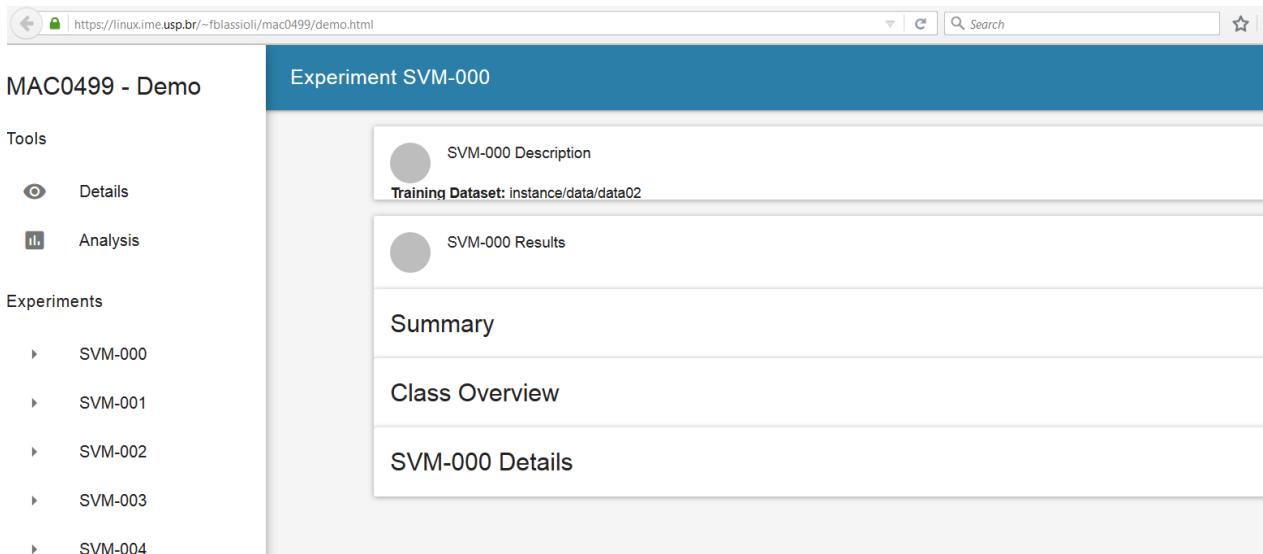


Figura 10: Visualizador de resultados acessado via browser *firefox*

Experiment SVM-009

SVM-009 Description

**Training Dataset:** instance/data/data02  
**Test Dataset:** instance/data/data02-tr

<b>kernel_type</b>	2	<b>gamma</b>	1	<b>max_axis</b>	128	<b>lo</b>	3
<b>C</b>	10000000	<b>svm_type</b>	100	<b>r</b>	12	<b>scales</b>	1
				<b>f</b>	0.5		

SVM-009 Results

Summary

Confusion Matrix			Statistics				
<b>Chuteira</b>	19	19	2	<b>ACC</b>	69.7 %	<b>FDR</b>	33.4 %
<b>Bota</b>	10	29	1	<b>F1</b>	63.8 %	<b>PPV</b>	66.6 %
<b>Chinelo</b>	4	8	28	<b>FNR</b>	36.7 %	<b>FPR</b>	24.4 %
Table of Confusion			<b>TNR</b>	75.6 %	<b>NPV</b>	73.6 %	
76 True positives	44 False negatives		<b>MCC</b>	39.5 %	<b>TPR</b>	63.3 %	
44 False negatives	120 True negatives						

Class Overview

Figura 11: Métricas e Tabela de Confusão como média das classes.

Experiment SVM-009

	<b>Chinelo</b>	4	8	28	<b>FNR</b>	36.7 %	<b>FPR</b>	24.4 %
Table of Confusion					<b>TNR</b>	75.6 %	<b>NPV</b>	73.6 %
76 True positives	44 False negatives		<b>MCC</b>	39.5 %	<b>TPR</b>	63.3 %		
44 False negatives	120 True negatives							

Class Overview

Overview				Table of Confusion				
	<b>Error</b>	<b>Precision</b>	<b>Recall</b>	<b>TP</b>	<b>FP</b>	<b>FN</b>	<b>TN</b>	
<b>Bota</b>	27.5 %	51.8 %	72.5 %	<b>Bota</b>	29	27	11	40
<b>Chuteira</b>	52.5 %	57.6 %	47.5 %	<b>Chuteira</b>	19	14	21	40
<b>Chinelo</b>	30.0 %	90.3 %	70.0 %	<b>Chinelo</b>	28	3	12	40

Statistics										
	<b>ACC</b>	<b>F1</b>	<b>FNR</b>	<b>TNR</b>	<b>MCC</b>	<b>FDR</b>	<b>PPV</b>	<b>FPR</b>	<b>NPV</b>	<b>TPR</b>
<b>Bota</b>	64.5 %	60.4 %	27.5 %	59.7 %	31.2 %	48.2 %	51.8 %	40.3 %	78.4 %	72.5 %
<b>Chuteira</b>	62.8 %	52.1 %	52.5 %	74.1 %	22.3 %	42.4 %	57.6 %	25.9 %	65.6 %	47.5 %
<b>Chinelo</b>	81.9 %	78.9 %	30.0 %	93.0 %	65.1 %	9.7 %	90.3 %	7.0 %	76.9 %	70.0 %

SVM-009 Details

Figura 12: Métricas e Tabela de Confusão comparando classe a classe.

Experiment SVM-009					
	Bota	Chuteira	Chinelo		
Overview			Statistics		
Error	27.5 %		<b>ACC</b>	64.5 %	<b>FDR</b> 48.2 %
Precision	51.8 %		<b>F1</b>	60.4 %	<b>PPV</b> 51.8 %
Recall	72.5 %		<b>FNR</b>	27.5 %	<b>FPR</b> 40.3 %
Table of Confusion			<b>TNR</b>	59.7 %	<b>NPV</b> 78.4 %
29 True positives	27 False positives		<b>MCC</b>	31.2 %	<b>TPR</b> 72.5 %
11 False negatives	40 True negatives				
Correct ( 29 )			Incorrect ( 11 )		
					
					

Figura 13: Métricas, Tabela de Confusão e Resultado da Identificação de cada imagem na classe.

Experiment Analysis

	Statistics									
	ACC	F1	FNR	TNR	MCC	FDR	PPV	FPR	NPV	TPR
SVM-000	50.0 %	16.7 %	66.7 %	77.8 %	11.1 %	22.2 %	11.1 %	22.2 %	66.7 %	33.3 %
SVM-001	48.8 %	15.7 %	69.2 %	75.5 %	2.2 %	56.1 %	10.5 %	24.5 %	64.3 %	30.8 %
SVM-002	59.0 %	45.6 %	51.7 %	70.6 %	21.4 %	43.3 %	56.7 %	29.4 %	68.2 %	48.3 %
SVM-003	76.6 %	73.5 %	26.7 %	79.6 %	53.1 %	25.7 %	74.3 %	20.4 %	79.0 %	73.3 %
SVM-004	79.2 %	76.8 %	23.3 %	81.5 %	58.2 %	22.8 %	77.2 %	18.5 %	81.1 %	76.7 %
SVM-005	79.8 %	77.6 %	22.5 %	81.9 %	59.5 %	22.1 %	77.9 %	18.1 %	81.7 %	77.5 %
SVM-006	81.3 %	79.3 %	20.8 %	83.4 %	62.8 %	19.8 %	80.2 %	16.6 %	82.8 %	79.2 %
SVM-007	71.8 %	66.8 %	33.3 %	76.1 %	43.1 %	31.9 %	68.1 %	23.9 %	75.3 %	66.7 %
SVM-008	70.7 %	65.0 %	35.0 %	76.3 %	42.0 %	32.2 %	67.8 %	23.7 %	74.9 %	65.0 %
SVM-009	69.7 %	63.8 %	36.7 %	75.6 %	39.5 %	33.4 %	66.6 %	24.4 %	73.6 %	63.3 %
SVM-010	70.2 %	64.8 %	35.8 %	75.1 %	39.7 %	33.7 %	66.3 %	24.9 %	73.7 %	64.2 %
SVM-011	78.4 %	75.5 %	24.2 %	80.7 %	56.8 %	24.0 %	76.0 %	19.3 %	81.1 %	75.8 %
SVM-012	54.2 %	28.7 %	60.0 %	78.6 %	23.0 %	21.4 %	45.2 %	21.4 %	68.5 %	40.0 %
SVM-013	81.4 %	79.5 %	20.8 %	84.1 %	63.8 %	18.6 %	81.4 %	15.9 %	82.9 %	79.2 %

Figura 14: Tabela Experimentos x Métricas.



Figura 15: Gráficos Experimentos x Métricas.

## 3.5 Extendendo a Framework

A framework foi pensada de modo que fosse fácil adicionar novos algoritmos que pudessem ser usados nos experimentos. Tais algoritmos podem ser tanto classificadores, como extratores de características.

Das seções anteriores sabemos que um extrator de características é qualquer objeto que disponibilize o método *compute* e um classificador é qualquer objeto que disponibilize *train* e *predict*. Nas seções seguintes iremos adicionar os descritores GRABED e LBP à framework.

### 3.5.1 Adicionando o descritor GRABED

O GRABED (GRAnulometry Based-descriptor applied in map of EDges)é um descritor que aplica granulometria morfológica ao mapa de bordas de uma imagem e mede a orientação destas e também sua extensão ao longo de uma orientação. O descritor consiste numa sequência de Padrões de Espectro, os quais são computados a partir de famílias de Elementos Estruturantes (EEs), onde cada família é composta por EEs lineares com mesma orientação e tamanhos crescentes.[24]

Assim as etapas para computar o descritor a partir de uma imagem são:

1. Obter o mapa de bordas da imagem.
2. Obter os conjuntos de famílias de EEs.
3. Para cada família calcular o conjunto padrões de espectro daquela família.

#### 3.5.1.1 Obtendo o mapa de bordas da imagem

Existem vários algoritmos para obtenção do mapa de bordas de uma imagem. Alguns deles como limiarização pelo método de Otsu e o detector de bordas Canny estão disponíveis na biblioteca OpenCV.

E podem ser utilizados de modo muito simples, fazendo chamada à função *Canny* do pacote *cv2*:

```
import cv2

img_path = 'lena.jpg'
img_original = cv2.imread(img_path)
img = cv2.imread(img_original, cv2.CV_LOAD_IMAGE_GRAYSCALE)
img = cv2.Canny(img_original,50,100)
cv2.imshow('Original image', img_original)
cv2.imshow('image', img)
```

O resultado pode ser visto na Figura 16.

#### 3.5.1.2 Obter os conjuntos de famílias de EEs

##### Calculando um EE linear usando o algoritmo de Bresenham

Infelizmente, as rotinas de construção de elementos estruturantes do OpenCV são bastante limitadas, então é necessário construirmos nós mesmos os elementos estruturante lineares.



Figura 16: Usando o algoritmo Canny disponível no OpenCV

Foi utilizado o algoritmo de Bresenham para calcular a linha a ser desenhada na matriz de entrada:

```
def line(x0,y0,x1,y1):
    dx = x1 - x0
    dy = y1 - y0
    eps = 0

    y = y0
    for x in range(x0, x1):
        yield x,y
        eps += dy
        if (eps << 1) >= dx:
            y += 1
            eps -= dx
```

Para construirmos o EE linear, calculamos a partir do centro da matriz quadrada de dimensão ímpar uma reta com ângulo  $d$  e depois disso usamos da simetria da operação para construir o restante da matriz.

```
def strel(l,d):
    print l,d
    def _get_rect_shape(l):
        if l % 2 == 0:
            l+=1
        return l,l
    w,h = _get_rect_shape(l)

    d = float(d)
    m = math.tan(math.radians(d))
    if d <= 45.0:
        x1 = w
        y1 = m*x1
        _line = line
    elif d <= 90.0:
```

```

y1 = h
x1 = y1 / m
_line = line2
elif d <= 180.0:
    output = strel(l, d-90.0)
return np.rot90(output)

m = np.zeros((w,h))
x1,y1 = int(math.ceil(x1)), int(math.ceil(y1))
for x,y in _line(0,0,x1,y1):
    m[x][y] = 1
m = np.rot90(m)
output = np.zeros(m.shape, dtype=np.uint8)
output[0:w/2+1, h/2:h] = m[w/2:w+1, 0:h/2+1]
output[w/2:w+1, 0:h/2+1] = np.rot90(output[0:w/2+1, h/2:h],2)

return output

```

A assinatura da função acima foi feita com base na função de mesmo nome escrita em MATLAB `strel('line', LEN, DEG)`. E seu uso pode ser visto abaixo:

<code>strel( 10 , 15 )</code>	<code>strel( 10 , 45 )</code>
<code>[[0 0 0 0 0 0 0 0 0 0]</code>	<code>[[0 0 0 0 0 0 0 0 0 1]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 0 0 0 1 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 0 0 0 1 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 0 0 0 1 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 0 0 0 1 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 0 0 0 1 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 0 0 0 1 0 0 0 0 0]</code>
<code>strel( 10 , 30 )</code>	<code>strel( 10 , 90 )</code>
<code>[[0 0 0 0 0 0 0 0 0 0]</code>	<code>[[0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 1]</code>	<code>[0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 1 0]</code>	<code>[0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 1 0 0]</code>	<code>[0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 1 0 0 0]</code>	<code>[0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 1 0 0 0 0]</code>	<code>[0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 1 0 0 0 0 0]</code>	<code>[0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 1 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 1 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[1 1 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 1 0 0 0 0 0]</code>
<code>[0 0 0 0 0 0 0 0 0 0]</code>	<code>[0 0 0 0 0 1 0 0 0 0 0]</code>

```

Input: Max_Axis, L0, R
Output: lista_EEs
tamanhos ← Ø;
l ← L0;
while l ≤ Max_Axis do
| tamanhos ← tamanhos + l;
| l ← l * 2;
end
lista_angulos ← conjunto de R ângulos entre 0° e 180°;
lista_EEs ← Ø;
for angulo ∈ lista_angulos do
| for tamanho ∈ tamanhos do
| | EE ← criar EE em forma de linha com tamanho tamanho e orientação angulo;
| | lista_EEs ← lista_EEs + EE;
| end
end
return lista_EEs

```

Figura 17: Algoritmo: Gerar EEs [24]

### Gerando a família de EEs lineares

Sendo possível obter EEs lineares com a função *strel*, podemos agora calcular as famílias de EEs de acordo com o algoritmo presente na Figura 17.

**gen\_ees:** gera uma lista de tuplas da forma (*tamanho*, *ângulo*, *matriz-do\_ee*).

```

def gen_ees(max_axis, 10, r):
    from collections import namedtuple
    EERow = namedtuple('EERow', ['length', 'angle', 'matrix'])

    sizes = []
    l = 10
    while l <= max_axis:
        sizes.append(l)
        l = l*2
    angles = np.linspace(0,180,r)

    return [ EERow(l,a,strel(l,a)) for a in angles for l in sizes ]

```

**get\_ee\_families:** gera uma lista de tuplas da forma (*ângulo*, *lista\_de\_ees*) ordenada por ângulo. Cada EE na *lista\_de\_ees* é da forma (*tamanho*, *matriz*) e a lista está ordenada por tamanho.

```

def get_ee_families(max_axis, 10, r):
    from collections import defaultdict
    from operator import itemgetter

    EEs = gen_ees(max_axis, 10, r)
    # Group by angle
    d = defaultdict(list)
    for ee in EEs:
        d[ee.angle].append((ee.length,ee.matrix))
    EE_families = sorted(d.items(), key=itemgetter(0))
    for a, ee_group in EE_families:

```

```

# sort by length
ee_group.sort(key=itemgetter(0))
return EE_families

```

### 3.5.1.3 Obter Padrões de Espectro de cada familia

Obter os padrões de espectro de uma família de EEs é bastante simples, basta efetuar uma sequência de aberturas morfológicas e calcular a área da imagem resultante.

Em outras palavras, é recebida uma lista de EEs em ordem crescente de tamanho e com um mesmo ângulo de orientação. Os EEs são aplicados em uma sequência de aberturas e a área da imagem resultante destas aberturas e o percentual de redução em relação a área inicial são calculados, sendo o percentual armazenado no vetor de retorno *ps*.

```

def grabed(img, scales=1, f=0.5, max_axis=128, lo=3, r=12, border_extractor=default_border_extractor):
    ps = []
    for s in xrange(1,scales+1):
        ir = cv2.resize(img, (0,0), fx=f, fy=f)
        borders = border_extractor(ir)
        initial_area = np.count_nonzero(borders)
        prev_area = initial_area

        initial_area = float(initial_area)
        for a,ee_family in get_ee_families(max_axis, lo, r):
            acc = 0.0
            for l, ee in ee_family:
                io = cv2.morphologyEx(borders, cv2.MORPH_OPEN, ee)
                tmp_area = np.count_nonzero(io)
                ps.append( (prev_area-tmp_area)/initial_area )
                acc = acc + (prev_area-tmp_area)/initial_area

            prev_area = tmp_area
    return ps

```

O script abaixo faz uso do descritor acima e os resultados podem ser visualizados na Figura 18.

```

from senjo.algorithms.grabed import grabed
import cv2

img_path = 'bota.jpg'
img = cv2.imread(img_path, cv2.CV_LOAD_IMAGE_GRAYSCALE)
grabed(img, output_file='bota_grabed.png')

```

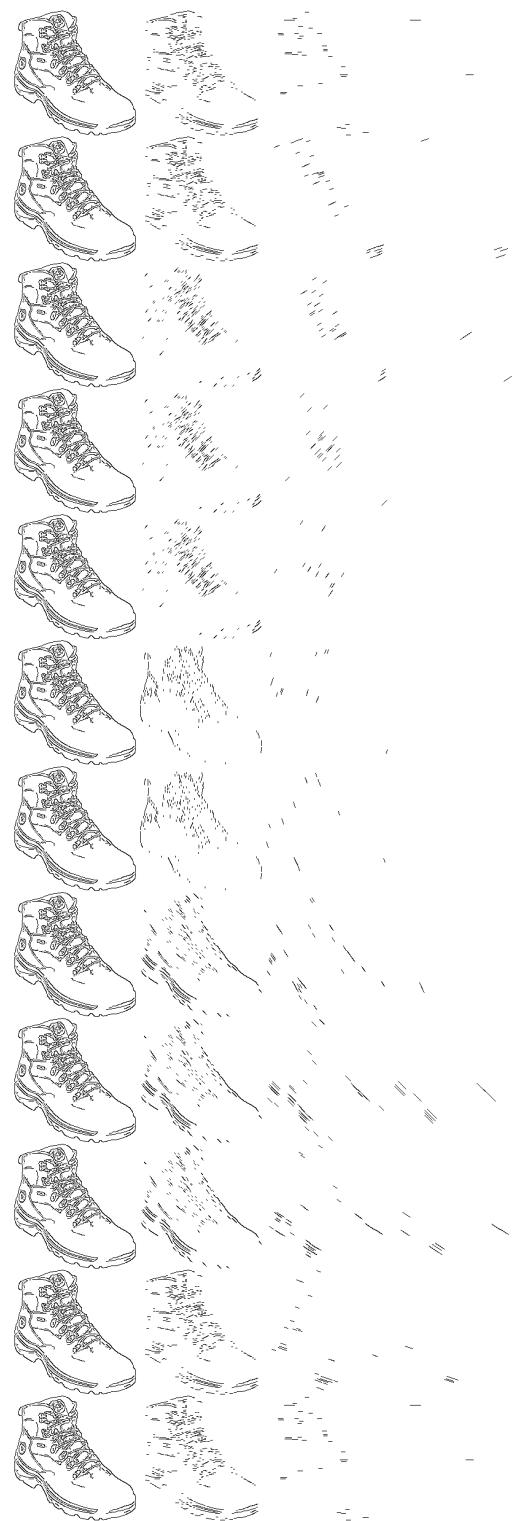


Figura 18: Teste do algoritmo GRABED

### 3.5.1.4 Adicionando à framework

Agora que temos a implementação do algoritmo, precisamos criar um *DescriptorExtractor*(seção 3.2.3) que faça uso das rotinas criadas.

Para tanto basta herdar da classe *senjo.algorithms.BaseDescriptorExtractor* e implementar o método *\_compute* fazendo uso da função desenvolvida nas seções anteriores.

Adicionando em *senjo/algorithms/\_\_init\_\_.py* o seguinte código:

```
import cv2
from grabed import grabed

def default_border_extractor(img):
    img = cv2.Canny(img,50,100)
    return img

class GRABED(BaseDescriptorExtractor):
    def __init__(self, scales=1, f=0.5, max_axis=128, l0=3, r=12,
                 border_extractor=default_border_extractor, output_file=None):
        self.scales = scales
        self.f = f
        self.max_axis = max_axis
        self.l0 = l0
        self.r = r
        self.border_extractor = border_extractor
        self.output_file = output_file

    def _compute(self, img):
        desc = grabed(img, self.scales, self.f, self.max_axis, self.l0, self.r, se
```

Torna possível utilizar o GRABED num experimento seguinte modo:

```
from senjo.algorithms import GRABED
class GRABEDExperiment(Experiment):
    @property
    def descriptor_extractor(self):
        return GRABED()
```

### 3.5.2 Adicionando o descriptor LBP

#### 3.5.2.1 Definição

Segundo [23], a ideia do LBP é utilizar o valor do pixel central  $f(h,c)$  como limiar para comparar os pixels da vizinhança. Assim é verificado se cada pixel dos 8 vizinhos é maior ou igual ao pixel central. Este resultado consiste de 8 comparações verdadeiro ou falso. Um código é criado a partir destas comparações. Dá-se um peso a cada uma destas comparações de modo que este código de 8 comparações gere um valor entre 0 e 255.

A função de pesos a ser utilizada é da por:

$$g_{LBP}(f_c) = \sum_{p=0}^7 (f_p \geq f_c) 2^p$$

onde:  $a \geq b = \begin{cases} 1, & \text{se } a \geq b \\ 0, & \text{caso contrario} \end{cases}$

$f_c$	o pixel central
$f_p$	o pixel da vizinhança conforme figura abaixo

#### 3.5.2.2 Adicionando à framework

Adicionar o LBP à framework é tão simples quanto herdar a classe *BaseDescriptorExtractor* e adicionar ao arquivo *senjo/algorithms/\_\_init\_\_.py* o seguinte:

```
class LBP(BaseDescriptorExtractor):
    def lbp(self, f):
        import numpy as np
        H,W = f.shape
        result = np.zeros((H-2,W-2), dtype='uint8')
        w = np.array([[1,2,4],[8,0,16],[32,64,128]], dtype='uint8')
        for dh,dw in [(0,0),(0,1),(0,2),(1,0),(1,2),(2,0),(2,1),(2,2)]:
            i = dh-2 if dh != 2 else None
            j = dw-2 if dw != 2 else None
            result += ((f[1:-1,1:-1] <= f[dh:i,dw:j])*w[dh,dw])
        return result

    def _compute(self, img):
        return np.float32(self.lbp(img))
```

## 4 Resultados e Trabalhos Futuros

Foi desenvolvida uma *framework* que permite realizar e visualizar experimentos de classificação de objetos utilizando algoritmos disponíveis na biblioteca *OpenCV*. É fácil acrescentar novos algoritmos à *framework*, como demonstrado na seção 3.5.

No entanto há várias adições e melhorias que poderiam ser feitas para tornar a aplicação uma ferramenta mais poderosa que auxilie na construção de sistemas reconhecedores de objetos, como por exemplo:

- Integrar mais fortemente o mecanismo de experimentos e o visualizador. Isto é, transformar o visualizador de resultados numa aplicação capaz de realizar experimentos fazendo uso da framework.
- Adicionar mecanismos de *cross-validation* e suporte a outras bibliotecas populares como *VLFeat* e *CCV* à framework.
- Adicionar à aplicação web um manipulador de datasets que permita processar e analisar o efeito de diversos algoritmos em imagens, assim como softwares como o *ImageJ* [38] (veja a figura 19)

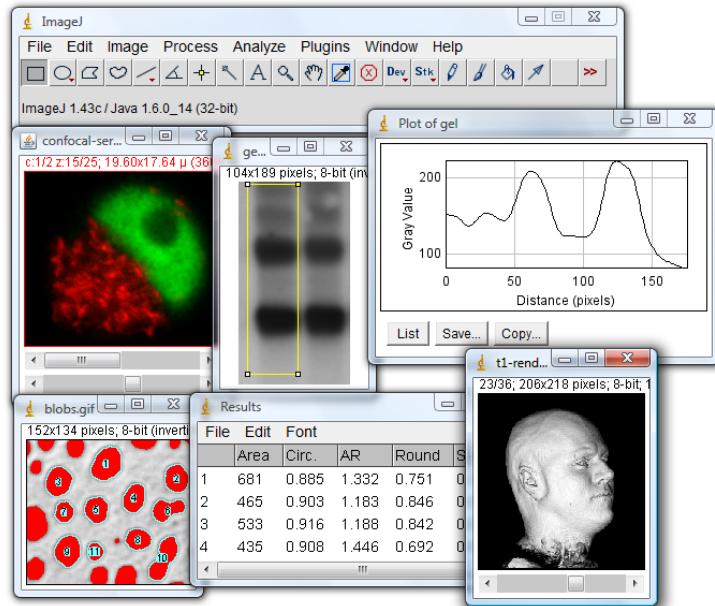


Figura 19: Funcionalidades do *ImageJ*

## Referências

- [1] M. Treiber. *An introduction to object recognition: Selected algorithms for a wide variety of applications.* Springer, 2010.
- [2] Flow powered by amazon: Identify tens of millions of products, including books and dvds., 2015.
- [3] Nvidia announces pricing and a launch date for drive px self-driving car platform at gtc 2015, 2015.
- [4] Here's how deep learning will accelerate self-driving cars, 2015.
- [5] Opencv: Open source computer vision, 2015.
- [6] Ccv: A modern computer vision library, 2015.
- [7] Vlfeat: open source library implements popular computer vision algorithms specializing in image understanding and local features extraction and matching, 2015.
- [8] Torch: Scientific computing framework with wide support for machine learning algorithms, 2015.
- [9] Fair open sources deep-learning modules for torch, 2015.
- [10] Ian T. Jolliffe. *Principal Component Analysis.* Springer, 2002.
- [11] Erkki Oja Aapo Hyvärinen, Juha Karhunen. *Independent Component Analysis.* John Wiley & Sons, 2001.
- [12] H. Sebastian Seung Daniel D. Lee. Learning the parts of objects by non-negative matrix factorization. *Nature*, 1999.
- [13] Mike Stephens Chris Harris. A combined corner and edge detector. 1988.
- [14] Cordelia Schmid Krystian Mikolajczyk. Indexing based on scale invariant interest points. *IEEE International Conference on Computer Vision*, 1:525–531, 2001.
- [15] David Lowe. Distinctive image features from scale-invariant keypoints. *Intern. Journal of Computer Vision*, 60:91–110, 2004.
- [16] Cordelia Schmid Krystian Mikolajczyk. An affine invariant interest point detector. *European Conference on Computer Vision*, 1:128–142, 2002.
- [17] U. Martin Tomas Pajdla Jiri Matas, Ondrei Chum. Robust wide baseline stereo from maximally stable extremal regions. *Machine Vision Conference*, I:384–393, 2002.
- [18] Tony Lindeberg. Feature detection with automatic scale selection. *International Journal of Computer Vision*, 30(2):77–116, 1998.
- [19] David Lowe. Object recognition from local scale-invariant features. *IEEE International Conference on Computer Vision*, pages 1150–1157, 1999.

- [20] Tinne Tuytelaars Luc Van Gool Herbert Bay, Andreas Ess. Surf: Speeded up robust features. *CVIU*, 110(3):346–359, 2008.
- [21] Michael Jones Paul Viola. Rapid object detection using a boosted cascade of simple features. 2001.
- [22] D. Harwood T. Ojala, M. Pietikainen. Performance evaluation of texture measures with classification based on kullback discrimination of distributions. 1994.
- [23] Fee0146 - laboratório de programação python/numpy de processamento de imagens e reconhecimento de padrões - 4a. edição, 2015.
- [24] Roberto Hirata Jr Arnaldo Câmara Lara. A granulometry based descriptor for object categorization. *Mathematical Morphology and Its Applications to Signal and Image Processing*, pages 413–424, 2013.
- [25] David G. Stork Richard O. Duda, Peter E. Hart. *Pattern Classification*. John Wiley & Sons, 2000.
- [26] Robert E. Shapire Yoav Freund. A decision-theoretic generalization of online learning and an application to boosting. *Journal of Computer and System Sciences*, 55:119–139, 1997.
- [27] V. Vapnik C. Cortes. Support-vector networks. 1995.
- [28] Richard Szeliski. *Computer Vision: Algorithms and Applications*. 2011.
- [29] Michael G. Madden Tom Howley. An evolutionary approach to automatic kernel construction. 2006.
- [30] Andrew Zisserman Josef Sivic. Video google: A text retrieval approach to object matching in videos. *ICCV '03 Proceedings of the Ninth IEEE International Conference on Computer Vision*, page 1470, 2003.
- [31] Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. 1998.
- [32] J. Dana G. Cula. Compact representation of bidirectional texture functions. 2001.
- [33] Jitendra Malik Thomas Leung. Representing and recognizing the visual appearance of materials using three-dimensional textons. 2001.
- [34] Chih-Fong Tsai.
- [35] Scikit-learn: Machine learning in python, 2015.
- [36] Flask is a microframework for python based on werkzeug, jinja 2 and good intentions, 2015.
- [37] Reactjs: A javascript library for building user interfaces, 2015.
- [38] Imagej: Image processing and analysis in java, 2015.