Detección de placas de vehículos en imágenes usando operaciones morfologicas y los algoritmos SVM y KNN

Universidad Nacional de Ingeniería
Facultad de Ciencias
Maestría en Ciencia de la Computación

Daniel Jimenez
Ingeniero Electricista

***Resumen*—En el presente trabajo se desarrolla un modelo de detección de placas utilizando operaciones morfologicas y el entrenamiento de modelos SVM y KNN. El enfoque utilizado consiste en dos pasos: 1. Detectar las posiciones de las placas usando operaciones morfologicas (función sobel, erosion, dilatación, cierre, apertura), 2. entrenar un modelo SVM y un modelo KNN que permita ingresar vectores generados a partir de las imágenes con su respectivo etiquetado.**

Keywords: Función Sobel, Función Erosión, Función Dilatación, Función Cierre, Función Apertura.

I. Introducción

El presente Informe es un complemento al Informe 2 del curso de Analisis de Imagenes, el cual permite automatizar las operaciones morfologicas expuestas en el Informe 2 y generalizar la deteccion de las placas con el algoritmo K-Nearest Neighbor (KNN). Asimismo, todos los codigos fueron escritos en python y no se utilizaron librerias para las operaciones morfologicas.

II. Marco Teórico

A. Conversión de escala de grises

En el presente proyecto se tiene imágenes en formato de escala de grises por lo que se adquiere una sola matriz con el tamaño de la resolución de imagen, y una densidad de 8 bits.

B. Convolución de imagen

Según [4], la convolución puede ser descrito forma sencilla, como una multiplicación de funciones. Si f y g son funciones en t, entonces la convolución de f y g sobre un intervalo infinito es una integral dada por:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{1}$$

C. Dilatación y erosión

Proceso matemático morfológico en las imágenes en espacio de color binario mediante el cual se busca dilatar el tamaño de un objeto en la imagen.

En el presente proyecto se realiza dicho proceso para eliminar objetos que no pertenezcan al área de la placa vehicular.

```python
def binarizar_imagen( imagen_grayscale ):

    imagen_binarizada=[]

    for i in range(np.shape(imagen_grayscale)[0]):
        n=0
        column_binarizada=[]
        for j in range(np.shape(imagen_grayscale)[1]):
            m=imagen_grayscale[i][j]
            if m>=128:
                bit=1
            else:
                bit=0
            column_binarizada.append(bit)

        imagen_binarizada.append(column_binarizada)

    return imagen_binarizada
```

Fig. 1: codificacion binarizacion

D. Clasificación con Support Vector Machine (SVM)

La performance del SVM, depende del tamaño del kernel Originalmente fue desarrollado como método de clasificación binaria actualmente muy usado para clasificación múltiple. Este método se basa en el Maximal Margin Clasifier (clasificación lineal) y de hiperplano, mediante el cual permite realizar clasificación como si fuera lineal al aumentar las dimensiones para trazar la clasificación:

El SVM tiene un hiper-parámetro importante a considerar para su aplicación en clasificación múltiple el cual es el factor "C" de costo o penalidad de la medida de distancia del margen de las observaciones.

Además, se utiliza un kernel para el aumento de dimensionalidad utilizando los datos de las observaciones para clasificaciones no lineales, por ejemplo: Lineal, polinómico, Gaussian. Finalmente, la estrategia elegida para la clasificación en dimensión multiple las cuales pueden ser: One-versus-one, One-versus-all, Directed Acyclic Graph SVM(DAGSVM).

En el presente proyecto se utilizó el SVM de la siguiente

```python
#codigo SOBEL
def convolucion_imagen(imagen:list, mascara:list,)->list:

    alto=len(imagen)
    ancho=len(imagen[0])

    tamano_mascara=len(mascara)
    div=int(tamano_mascara/2)
    imagen_nueva=imagen.copy()

    for i in range(alto):
        for j in range(ancho):
            #lista suma de cada componente
            suma_colores=[0.0, 0.0, 0.0]
            suma_coef_mascara=0.0
            x=0
            for fila in range(i-div, i +div+1):
                y=0
                for columna in range(j-div,j+div+1):
                    if fila>=0 and fila <alto and columna >=0 and columna< ancho:
                        suma_colores[0]+=(mascara[x][y]*imagen[fila][columna][0])#R
                        suma_colores[1]+=(mascara[x][y]*imagen[fila][columna][1])#G
                        suma_colores[2]+=(mascara[x][y]*imagen[fila][columna][2])#B
                        suma_coef_mascara += mascara[x][y]
                    y+=1
                x+=1
            if suma_coef_mascara!=0:
                nuevo_r=suma_colores[0]/suma_coef_mascara
                nuevo_g=suma_colores[1]/suma_coef_mascara
                nuevo_b=suma_colores[2]/suma_coef_mascara
            else:
                nuevo_r=suma_colores[0]
                nuevo_g=suma_colores[1]
                nuevo_b=suma_colores[2]

            nuevo_pixel=(nuevo_r,nuevo_g,nuevo_b)
            imagen_nueva[i][j]=nuevo_pixel

    return imagen nueva
```

Fig. 2: codificacion binarizacion

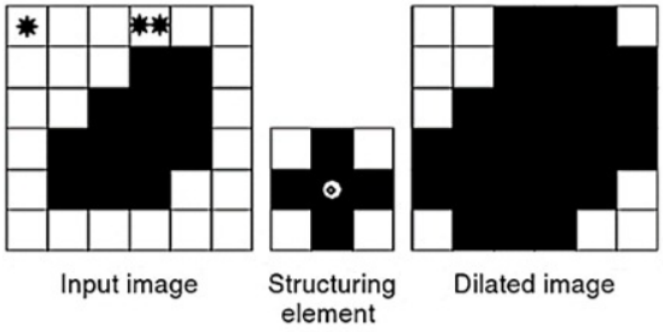


Figure a. Dilation

Fig. 3: Proceso de dilatación

```python
def dilate_image(image):
    k = 3
    kernel = np.zeros((k, k))
    # kernel = np.array([[1, 0 ,1] ,[0, 0 ,0], [1, 0, 1]])
    constant = (k-1)//2
    bordered_image = add_temp_border_to_image(image)
    h,w = bordered_image.shape
    bordered_dilated_image = np.ones((h, w), dtype=np.uint8)
    for i in range(constant, h - constant):
      for j in range(constant, w - constant):
        sub_image= bordered_image[i-constant:i+constant+1, j-constant:j+constant+1]
        bordered_dilated_image[i,j]= 0 if over_images(sub_image, kernel) else 1
    dilated_image = remove_temp_bordered_from_image(bordered_dilated_image)
    return dilated_image
```

Fig. 4: codigo proceso de dilatación

manera:

- $C = 1$, por defecto
- $Gamma = 0.001$
- $Kernel$: 'rbf', radial basis function kernel tipo exponencial como sigue:

$$K(x, x') = e^{-\gamma||x-x'||^2} \tag{2}$$

$$\gamma = \frac{1}{n_{features} * \sigma^2} \tag{3}$$

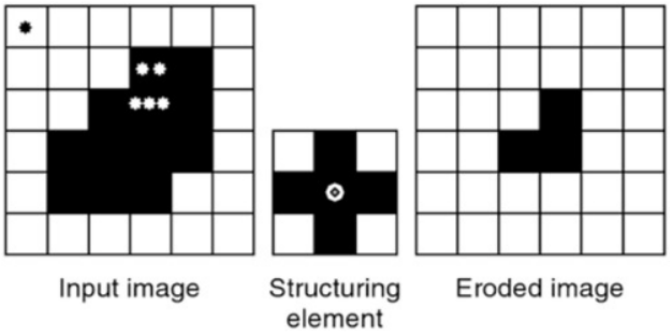


Fig. 5: Proceso de erosion

```python
def erode_image(image):
  h, w = image.shape
  k = 3
  kernel = np.zeros((k,k))
  # kernel = np.array([[1, 0 ,1] ,[0, 0 ,0], [1, 0, 1]])
  eroded_image = np.ones((h, w), dtype=np.uint8)
  constant = (k-1)//2
  for i in range(constant, h - constant):
    for j in range(constant, w - constant):
      sub_image= image[i-constant:i+constant+1, j-constant:j+constant+1]
      eroded_image[i,j]= 0 if fit_images(sub_image, kernel) else 1
  return eroded_image
```

Fig. 6: codigo proceso de erosion

Con un conjunto de entrenamiento de 10% de 44 imágenes, se obtiene los resultados para la clasificación mostradas en la figura 17.

E. Clasificación con K-Nearest Neighbor (KNN)

KNN intenta predecir la clase correcta para los datos de prueba calculando la distancia entre los datos de prueba y todos los puntos de entrenamiento. Luego seleccione el número K de puntos que está más cerca de los datos de prueba. El algoritmo KNN calcula la probabilidad que los datos de prueba pertenezcan a las clases de datos de entrenamiento 'K' y se seleccione la clase que contiene la probabilidad más alta.

El funcionamiento de K-NN se puede explicar sobre la base del siguiente algoritmo:

1) Seleccione el número K de los vecinos
2) Calcule la distancia euclidiana de K número de vecinos
3) tome los K vecinos más cercanos según la distancia euclidiana calculada.
4) Asigne los nuevos puntos de datos a esa categoría para la cual el número de vecinos es máximo.
5) Fin

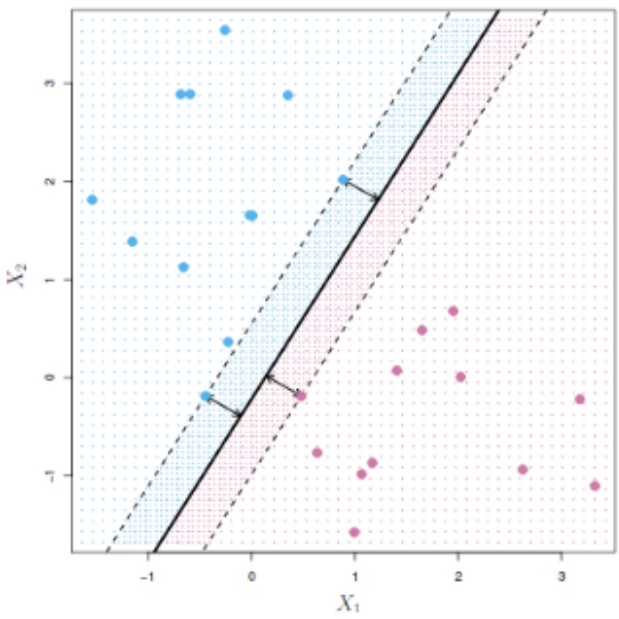


Fig. 7: Clasificación por medio de Maximal Margin Clasifier entre dos clases

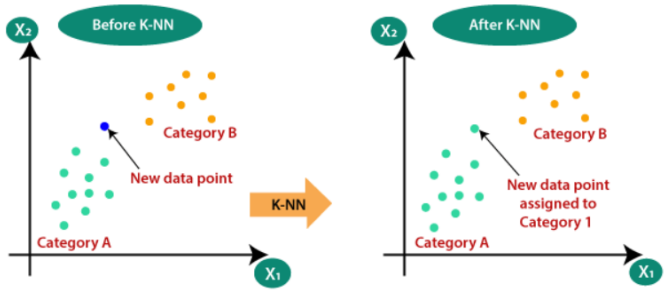


Fig. 8: algoritmo KNN

III. Metodología de solución

A. Primera etapa: Pre-procesamiento de imágenes para obtener máscaras de las placas

El objetivo de esta etapa es la de procesar la imagen original para detectar la imagen de las placas vehiculares como un objeto de imagen.

Esto se realiza mediante la aplicación de operaciones morfológicas de imagen como siguen:

1) Detección de bordes, se utiliza la convolución de imagen con el kernel de Sobel para detección de bordes de figuras perpendiculares y horizontales, ya que la figura de la placa es un objeto rectangular.
2) Uso de la función *imfill*, esta función permite rellenar de valores blancos o "1" los objetos que tengan un perímetro completo cerrado.
   Esto permite crear máscaras de objetos con áreas y perímetros definidos.

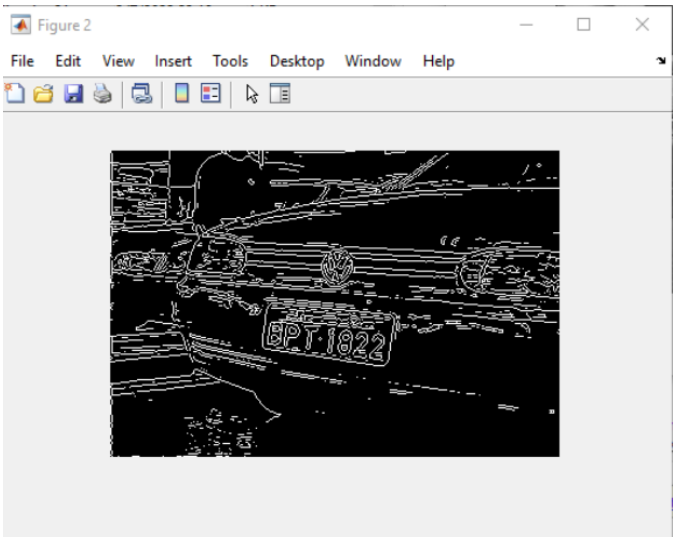


Fig. 9: Imagen con detección de bordes con kernel Besel(Matlab) en Python el kernel Canny da resultados similares

```
BW = logical([0   0   0   0   0   0   0   0;
              0   1   1   1   1   1   0   0;
              0   1   0   0   0   1   0   0;
              0   1   0   0   0   1   0   0;
              0   1   0   0   0   1   0   0;
              0   1   1   1   1   0   0   0;
              0   0   0   0   0   0   0   0;
              0   0   0   0   0   0   0   0]);
```

Fig. 10: Matriz imagen binaria con un objeto hueco de unos

```
ans =
     0     0     0     0     0     0     0     0
     0     1     1     1     1     1     0     0
     0     1     1     1     1     1     0     0
     0     1     1     1     1     1     0     0
     0     1     1     1     1     1     0     0
     0     1     1     1     1     0     0     0
     0     0     0     0     0     0     0     0
     0     0     0     0     0     0     0     0
```

Fig. 11: Imagen rellenada por detección de continuidad de vecinos próximos

3) Uso de erosión y dilatación
   Se aplica operaciones morfológicas para erosionar (disminuir los objetos de color blanco o ceros) de esta manera podemos eliminar líneas que serían ruido en nuestra detección del objeto placa.

   Se aplica operaciones morfológicas para dilatar (aumentar los objetos de color blanco o ceros) de esta manera podemos recuperar el tamaño original del objeto el cual perdió área por el proceso de erosión.

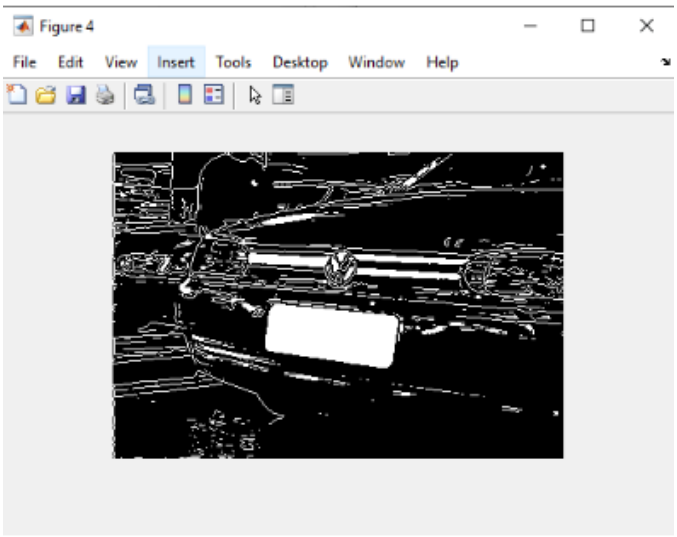


Fig. 12: Imagen de placa con “imfill” aplicado.

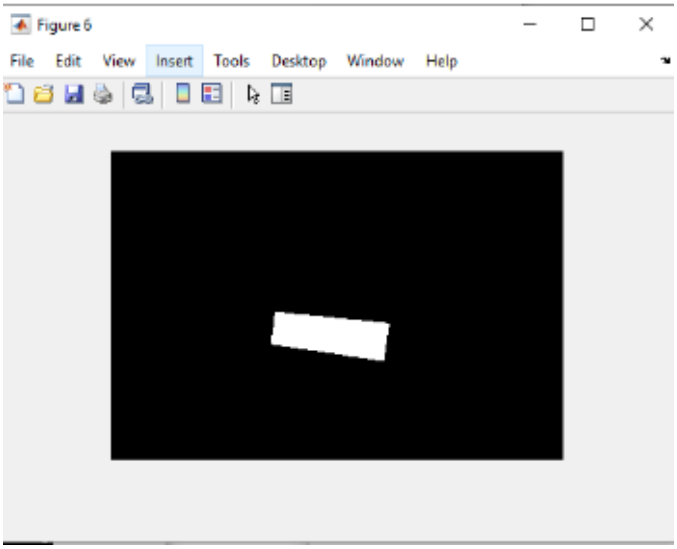


Fig. 13: Se aplica erosión para eliminar las líneas delgadas.

B. Segundo etapa: generación de sub-imágenes y entrenamiento del modelo

Obtenidas las mascaras útiles del primer paso (43 de 200), se realiza el siguiente tratamiento.

1) Leer el conjunto de imágenes que serán procesables para el entrenamiento del modelo: este conjunto de imágenes corresponde a los original de los cuales se derivan las mascaras correctamente procesadas.
2) Binarizar las imágenes de origen: esto debido a la necesidad de encontrar las coordenadas de las mascaras para definir las dimensiones de las ventanas de rastreo que permita incluir en sub-imágenes a todas las imágenes que contienen placas. La binarización se realiza con un umbral de 50 (de 255)
3) Encontrar coordenadas: del paso 3 se realiza un barrido de todas las píxeles de las imágenes con mascaras útiles.

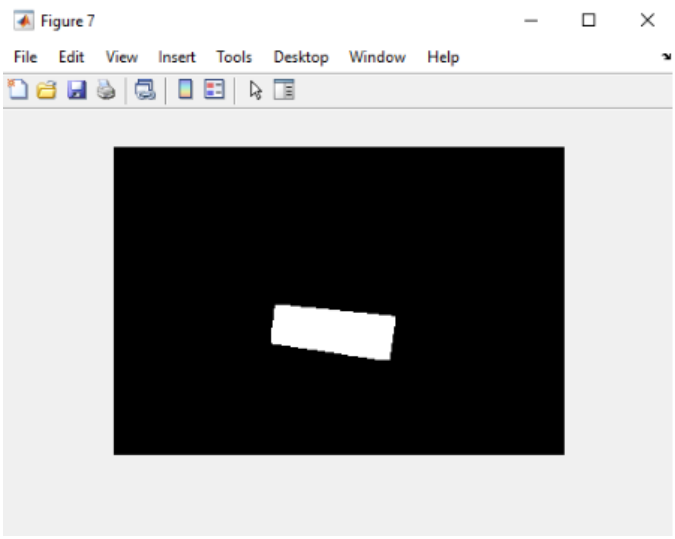
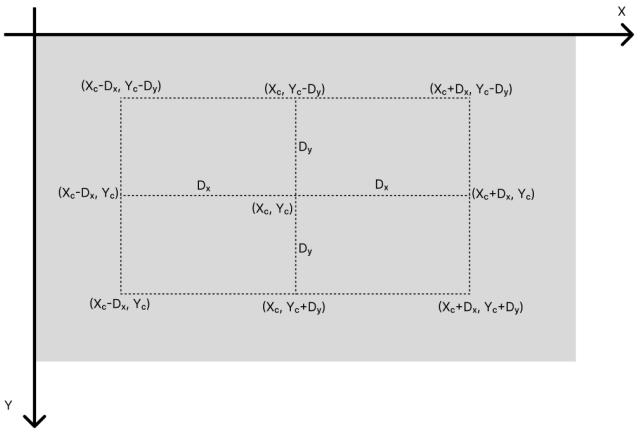
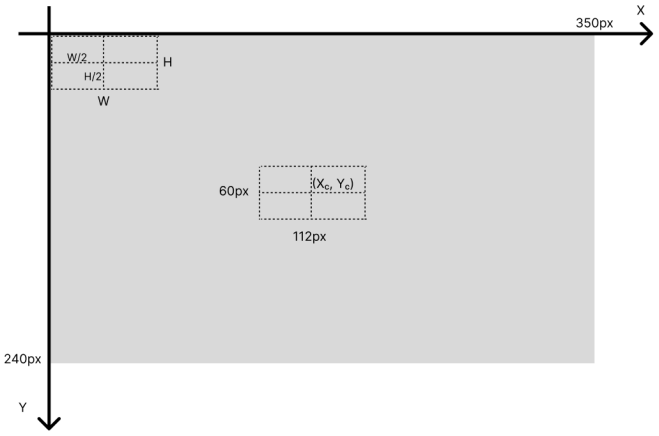


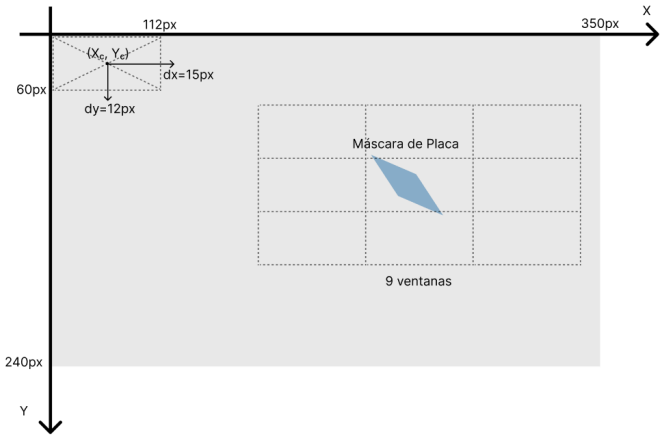
Fig. 14: Se aplica dilatación para recuperar área del objeto placa que sufrió erosión anteriormente.

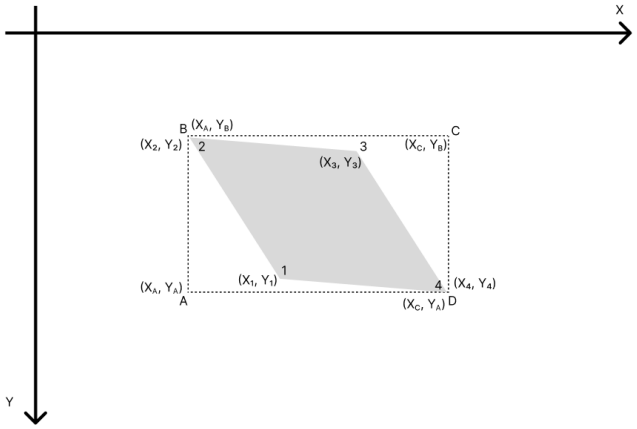
El proceso consiste en guardar las posiciones donde hay píxeles de valor 1 (luego de binarizada la imagen), y luego entrar los x, y máximos y mínimos, para determinar el tamaño de las ventanas de barrido estándar. Bajo este enfoque, se encontró que la dimensión de la ventana de barrido estándar debe ser de 60×112 píxeles. Asimismo, con el fin de aumentar la probabilidad que en una (o varias) de las ventanas de barrido estándar se encuentre la placa, se determino que la ventana se mueva 12 píxeles en el eje x, y 15 píxeles en el eje y, generando un total de 225 sub-imágenes en las cuales se encontrara la placa.



4) Encontrar centro de los rectángulos que contienen las mascaras: teniendo en cuenta las coordenadas de los centros de las placas y las coordenadas de todas las ventanas de barrido estándar, se generan las 225 sub-imágenes por imagen, haciendo un total de 43×225 sub-imágenes para el conjunto de entrenamiento.







5) Encontrar las sub-imágenes que contienen placas: teniendo los centros de las mascaras y todas las sub-imágenes, se busca la ventana de barrido estándar cuyo centro este los mas cercano posible a las coordenadas de los centros de las mascaras. Una vez hallada, se toma como ventanas estándar que contiene la placa, a las 8 ventanas cuyos centros rodeen la ventana estándar inicialmente halladas, haciendo un total de 9 ventanas estándar que contienen placas.
6) etiquetado: encontradas las sub-imágenes que contienen las placas (9 en total), se procede a etiquetarlas, placa será 1 y no placa será 0.
7) Elección del SVM: se selecciona como modelo al SVM por ser un modelo especializado en la clasificación binaria, lo cual corresponde al enfoque utilizado (placa o no placa). La data que ingresa al modelo corresponde a 90% de train y 10% de test.

IV. Resultados

A. Resultados del entrenamiento del modelo

Los resultados se muestran en las figuras 15, 17 y 19.

```
Classification report for classifier SVC(gamma=0.001):
              precision    recall  f1-score   support

           0       0.98      0.99      0.99       923
           1       0.85      0.62      0.72        45

    accuracy                           0.98       968
   macro avg       0.92      0.81      0.85       968
weighted avg       0.98      0.98      0.98       968
```

Fig. 15: Precisión

```
Confusion matrix:
[[918   5]
 [ 17  28]]
```

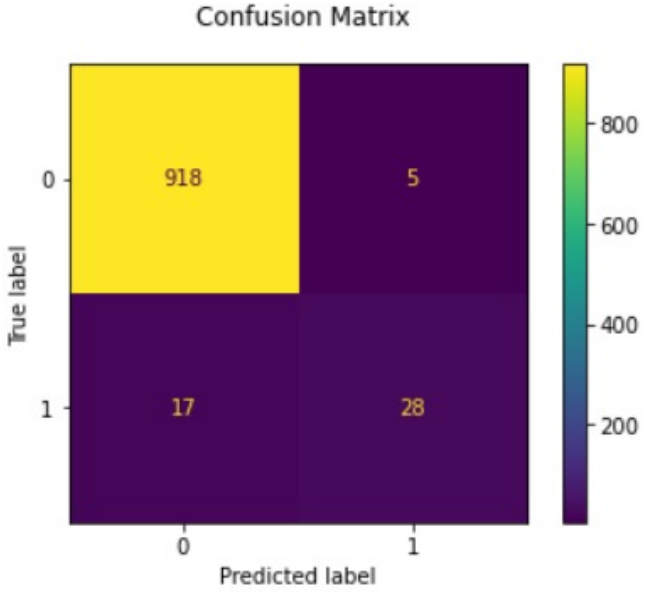


Fig. 16: Matriz de Confusión SVM

B. Resultados de las pruebas con el modelo entrenado

Entrenado el modelo SVM, se procede a evaluar su precisión con un conjunto de pruebas, las cuales corresponden a

```
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(test_data_predicted, y_test)
print(cm)
```

```
[[917  23]
 [  6  22]]
```

Fig. 17: Matriz de Confusión KNN

```
loaded_model = joblib.load(filename)
result = loaded_model.score(X_test, y_test)
print(result)
```

```
0.9772727272727273
```

Fig. 18: Precisión del Modelo SVM

```
test_data_predicted = knn.predict(X_test)
print(accuracy_score(test_data_predicted, y_test))
```

```
0.9700413223140496
```

Fig. 19: Precisión del Modelo KNN

imágenes que no han sido incluidas en la fase de entrenamiento del modelo. A continuación se muestran los siguientes resultados:

```
prueba_resultado=mostrar_resultados_de_pruebas_modelo('./placas-2/orig_0030.png')
```

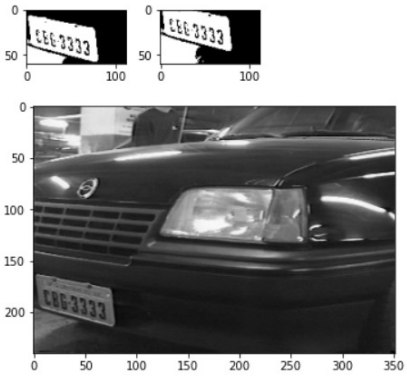


Fig. 20: prueba SVM con imagen de test 30

```
prueba_resultado=mostrar_resultados_de_pruebas_modelo_1('./placas-2/orig_0030.png')
```

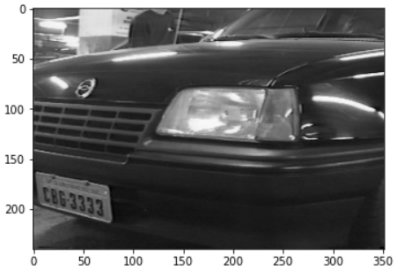


Fig. 21: prueba KNN con imagen de test 30

Puede notarse que el modelo identifica la(s) posición(es) donde se encuentran las placas, pudiéndose dar el caso que encuentre de 1 a mas posiciones; sin embargo, hay casos en que el modelo no puede identificar la posición de la placa en toda la imagen:

```
prueba_resultado=mostrar_resultados_de_pruebas_modelo('./placas-2/orig_0009.png')
```

Fig. 22: prueba SVM con imagen de test 9

```
prueba_resultado=mostrar_resultados_de_pruebas_modelo_1('./placas-2/orig_0009.png')
```

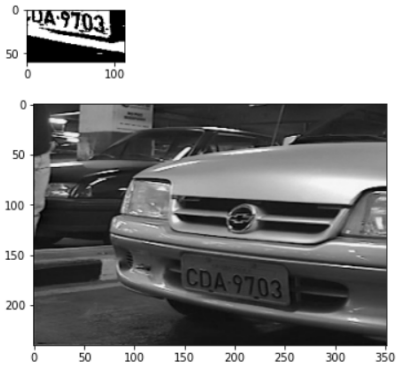


Fig. 23: prueba KNN con imagen de test 9

```
prueba_resultado=mostrar_resultados_de_pruebas_modelo('./placas-2/orig_0013.png')
```

Fig. 24: prueba SVM con imagen de test 13

Estos corresponden a casos aislados, en mas del 90% de los casos de prueba se observan resultados de detección satisfactorios.

Para el modelo KNN tenemos los siguientes resultados

V. Conclusiones

1) Con las operaciones morfologicas utilizadas, se obtiene el numero suficiente de mascaras de las imagenes de

```
prueba_resultado=mostrar_resultados_de_pruebas_modelo_1('./placas-2/orig_0013.png')
```

Fig. 25: prueba KNN con imagen de test 13

entrenamiento (43 de 200 imagenes) para poder entrenar los modelos KNN y SVM, los cuales alcanzan accuracy de mas del 95%

2) los modelos SVM y KNN pueden ser utilizados en simultaneo y se complementan, tal como se muestra en las figuras 20 y 21 donde el modelo SVM detecta la placa y no asi el de KNN. Asimismo, en la figura 24 y 25 se muestra que el modeolo KNN detecta la placa y el modelo SVM no lo detecta.
3) Utilizando las técnicas de procesamiento de imágenes, se puede obtener mascaras útiles para usarlas en el entrenamiento del modelo de detección. En este caso, se obtuvieron 43 mascaras a partir de imágenes originales, de 200 posibles (21.5%).
4) Se determino que las dimensiones de la ventana de barrido para obtener sub-imágenes debe ser de 60×112 píxeles para imágenes originales de 240×352 píxeles. Dicha ventana recorre la imagen original, cambiando su centro cada 12 pixeles en el eje x y 15 pixeles en el eje y, haciendo un total de 225 sub-imágenes. Dentro de este conjunto de sub-imágenes, se determinan 9 que contienen la placa, evaluando la posición del centro de las ventanas y la posición de la mascara que contiene la placa.
5) Debido a que se trata de una clasificación binaria (placa o no placa), se selecciona como modelo de clasificación al modelo SVM. Utilizando las sub-imágenes generadas, se divide todo el conjunto en 90% para train y 10% para test, obteniéndose el 97% de precisión.
6) Utilizando imágenes de prueba, se obtiene que en mas del 90% de veces el modelo predice con precisión.

REFERENCES

[1] Lee, E. R., Kim, P. K., & Kim, H. J. (1994, November). *Automatic recognition of a car license plate using color image processing. In Proceedings of 1st international conference on image processing* (Vol. 2, pp. 301-305). IEEE.

[2] Naito, T., Tsukada, T., Yamada, K., Kozuka, K., & Yamamoto, S. (2000). *Robust license-plate recognition method for passing vehicles under outside environment. IEEE transactions on vehicular technology*, 49(6), 2309-2319.

[3] Lin, C. H., & Li, Y. (2019, August). *A license plate recognition system for severe tilt angles using mask R-CNN*. In 2019 International Conference on Advanced Mechatronic Systems (ICAMechS) (pp. 229-234). IEEE.

[4] Sung Kim, *Applications of Convolutions in Image Processing with Matlab*, August 20, 2013.

[5] Mustafa Abdullah and Mohsin Abdulazeez - 2021 - *Machine Learning Applications based on SVM Classif.pdf*