Reporte de Tarea 4

Reconocimiento de patrones en imágenes

Nombre
Laura Montaner

Profesor
Miguel Carrasco

30 de Noviembre de 2025

Índice

1. Introducción

El reconocimiento de patrones en imágenes es una disciplina clave dentro del procesamiento digital y la visión computacional, cuya eficacia depende en gran medida de la correcta extracción, representación y selección de características. En este trabajo se desarrolla un pipeline completo orientado a la clasificación de texturas mediante descriptores derivados de la Matriz de Co-ocurrencia(GLCM). Para ello, se construyó un dataset, generando un vector de 24 características por muestra. Posteriormente, se aplicaron procedimientos de normalización y técnicas de reducción de dimensionalidad basadas en el algoritmo Sequential Forward Selection (SFS) optimizado con el Índice de Fisher, con el objetivo de reducir redundancia. Finalmente, se implementaron y evaluaron tres clasificadores supervisados:Naive Bayes, Decision Tree Classifier y K-Nearest Neighbors. Se utilizó métricas F-Score y validación cruzada estratificada (K-Fold) para obtener estimaciones del rendimiento de los modelos. El presente informe documenta este proceso, así como el análisis comparativo del comportamiento de los modelos bajo distintas configuraciones del dataset.

2. Procesamiento

2.1. Construcción del dataset

Se implementó el código **crear_bd.py** para transformar la data no estructurada (imágenes) en un conjunto de datos estructurado. Iterando sobre las imágenes, se separaron los canales de color (R, G, B y Escala de Grises) y se aplicó una cuantización de niveles de gris (0-10) usando la función MinMaxScaler. Posteriormente, calculó la Matriz de Co-ocurrencia usando la función graycomatrix, para extraer 6 descriptores de textura (contraste, energía, ASM, homogeneidad, correlación y disimilitud) por cada canal. Esto generó un vector de características de 24 dimensiones por imagen el cual se almacenó en un archivo CSV llamado **texturas_features_24.csv**

2.2. Normalización

El código **normalizar_bd.py** tomó el dataset generado anteriormente (texturas_features_24.csv) y aplicó una estandarización para eliminar sesgos de escala entre las distintas características. Utilizó la técnica StandardScaler (Z-score), la cual transformó cada columna numérica para que tuviera una media de 0 y una desviación estándar de 1. Esto aseguró que ninguna característica dominara sobre las demás, preparando los datos para algoritmos

sensibles a la escala (como KNN). El resultado se almacenó en el archivo **texturas_scaled_24.csv**

2.3. Selección de características

Se implementó el código **fisher_sfs.py** para reducir la dimensionalidad e identificar el subconjunto de características más significativo. Para ello se utilizó el algoritmo de búsqueda Sequential Forward Selection (SFS), el cual comienza vacío y agrega iterativamente la característica que maximiza el Índice de Fisher. El resultado fue un dataset filtrado que seleccionó 17 columnas manteniendo solo la información más relevante para la clasificación, descartando ruido y redundancia. Las 17 columnas seleccionadas se almacenaron en el archivo **texturas_seleccionadas_optimas.csv**

3. Implementación de los clasificadores

Se implementó los modelos de clasificación Naive Bayes, Árboles de desición y KNN probando los tres set de datos generados anteriormente(texturas_features_24.csv, texturas_scaled_24.csv y texturas_seleccionadas_optimas.csv).
Para todos los modelos se destinó 70 % de los datos para entrenamiento y 30 % para testing. Estos fueron escogidos de forma distribuida sobre las clases, es decir, se escogieron 15 muestras de cada clase para evaluar. Para la implementación de los algoritmos se utilizó funciones de la librería **Scikit-learn**, siguiendo el flujo: iniciación, entrenamiento y predicción. A continuación se detalla la creación de los códigos `bayes.py`, `arbol.py` y `knn.py`. Los tres modelos reciben un dataset y retornan la matriz de confusión, métricas por clase (precisión, exhaustividad y F1-score) y F-score global.

3.1. Naive Bayes

El algoritmo Naive Bayes es un modelo probabilístico basado en el Teorema de Bayes. Su objetivo es encontrar la clase C_k que maximice la probabilidad posterior dada una entrada de características X, en este caso, los descriptores de textura.
La formula fundamental es:

$$P(C_k \mid X) = \frac{P(X \mid C_k)\, P(C_k)}{P(X)} \tag{1}$$

Donde:

- $P(C_k \mid X)$: Probabilidad posterior.

- $P(C_k)$: Probabilidad a priori.
- $P(X \mid C_k)$: Verosimilitud (probabilidad de observar los datos X si la clase fuera C_k).
- $P(X)$: Evidencia o probabilidad marginal de los datos.

El algoritmo asume que todas las características son independientes entre sí dado la clase.
En el código `bayes.py` se utilizó `GaussianNB`, el cual asume que la verosimilitud de las características sigue una distribución normal.

```
clf_nb = GaussianNB()
```

Luego, usando `.fit`, el modelo aprende cómo se distribuyen las características para cada clase. Internamente calcula qué tan frecuente es cada textura dentro del conjunto de entrenamiento. También calcula la media y la varianza de cada característica por cada clase.

```
clf_nb.fit(X_train, y_train)
```

Finalmente, el modelo predice a que clase pertenece cada nueva imagen. La función `.predict` toma internamente los valores de cada característica y calcula la probabilidad de pertenecer a cada una de las 10 clases usando la fórmula Gaussiana con las medias y varianzas aprendidas en el paso anterior. Multiplica la probabilidad a priori de la clase y asigna la etiqueta que tenga la probabilidad resultante más alta.

```
 y_pred = clf_nb.predict(X_test)
```

El resultado de una sola ejecución del código, entregando cómo parámetro de entrada el dataset que contenía las 17 características más significativas (texturas_seleccionadas_optimas.csv), se obtuvo la matriz de confusión de la figura 1.

3.2. Árboles de decisión

Se creó el código `arbol.py` que implementa un Árbol de Decisión, método de aprendizaje supervisado no paramétrico que modela las decisiones como una estructura de árbol jerárquica.
En primer lugar, se define el modelo `DecisionTreeClassifier`. Se le da el parámetro `criterion='gini'`, que indica que el árbol elegirá las divisiones usando el índice Gini, una medida de impureza que decide qué características separan mejor las clases.

```
clf_tree = DecisionTreeClassifier(criterion='gini',
   random_state=42)
```

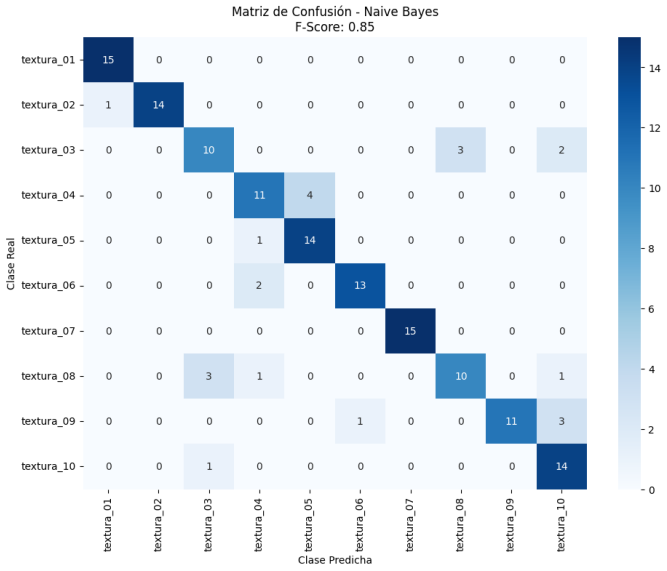

Figura 1: Matriz de confusión generada a partir del algoritmo Bayes

Utilizando el método `.fit` se construye el árbol buscando qué características permiten separar mejor las clases, creando nodos y ramas hasta definir reglas que permitan clasificar las texturas.

```
clf_tree.fit(X_train, y_train)
```

Finalmente, con el árbol ya entrenado, se predice la clase de cada muestra en el conjunto de prueba. El resultado es la lista de clases que el árbol asigna a cada muestra.

```
y_pred = clf_tree.predict(X_test)
```

El resultado de una sola ejecución del código, entregando cómo parámetro de entrada el dataset que contenía las 17 características más significativas (texturas_seleccionadas_optimas.csv), se obtuvo la matriz de confusión de la figura 2.

3.3. KNN

Para implementar el clasificador KNN (K-Nearest Neighbors o K Vecinos más Cercanos) se utilizó la clase `KNeighborsClassifier` de la librería **scikit-learn**. Este modelo es un *lazy learner*, es decir, no aprende parámetros durante la fase de entrenamiento, sino que almacena los datos y realiza las predicciones directamente comparando las distancias entre las muestras.

```
K_VECINOS = 5
```

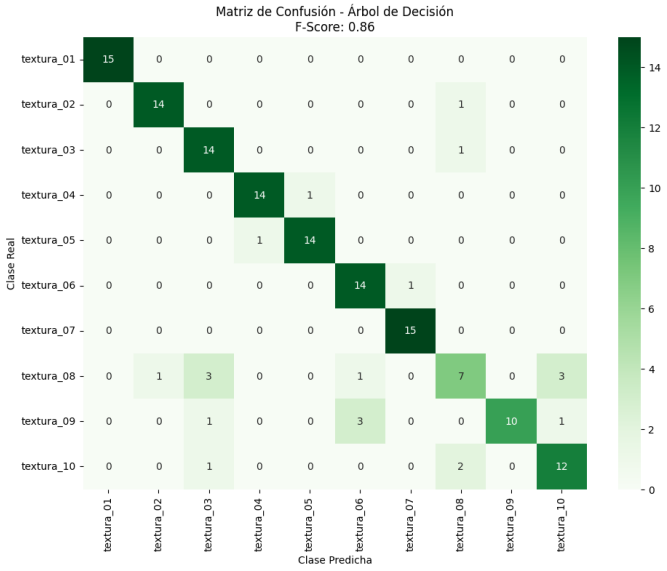

Figura 2: Matriz de confusión generada a partir del algoritmo Árbol de decisión

```
clf_knn = KNeighborsClassifier(n_neighbors=K_VECINOS,
    metric='euclidean')
```

Se seleccionó $K = 5$, lo que significa que cada punto de prueba será clasificado según la mayoría de sus cinco vecinos más cercanos en el conjunto de entrenamiento, utilizando la distancia euclidiana.
El método `.fit` en KNN no realiza cálculos estadísticos ni construye un modelo; simplemente guarda las muestras de entrenamiento para poder comparar distancias cuando se haga la predicción.

```
clf_knn.fit(X_train, y_train)
```

La predicción se realiza con la función `.predict`. Para cada muestra de prueba, el algoritmo calcula la distancia a todas las muestras de entrenamiento, selecciona los K vecinos más cercanos y asigna la clase que más se repite entre ellos.

```
y_pred = clf_knn.predict(X_test)
```

El resultado de una sola ejecución del código, entregando cómo parámetro de entrada el dataset que contenía las 17 características más significativas (texturas_seleccionadas_optimas.csv), se obtuvo la matriz de confusión de la figura 3.

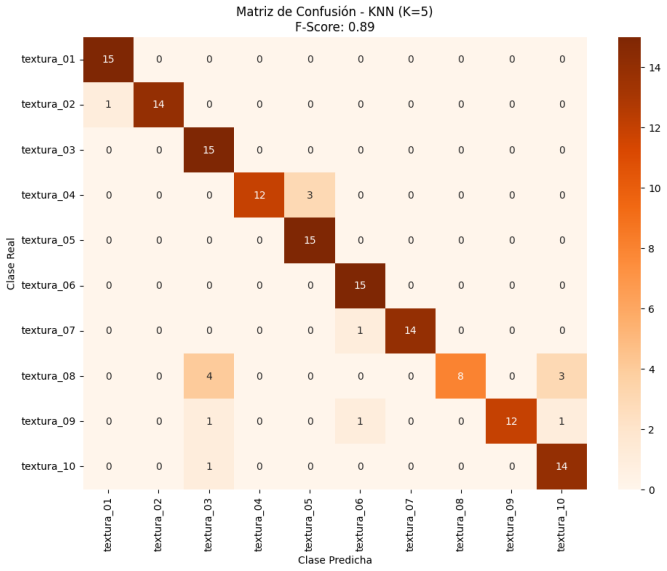

Figura 3: Matriz de confusión generada a partir del algoritmo KNN(K=5)

3.4. K-fold cross validation

Dado que una única ejecución de los algoritmos puede inducir sesgos por la forma en que se seleccionan los datos, se implementó K-Fold Cross Validation. Esta técnica permite evaluar el modelo varias veces en diferentes grupos de datos, con el objetivo de obtener una estimación del rendimiento estadísticamente más significativa y fiable que la proporcionada por una única ejecución.
Se implementó el código `kfold.py` con $k = 10$ iteraciones para cada algoritmo.

```
N_FOLDS = 10
#...
kfold = StratifiedKFold(n_splits=N_FOLDS, shuffle=True,
   random_state=SEED)
```

Cada modelo se entrenó con 9 folds y se probó con 1 fold, repitiendo esto 10 veces para cubrir todos los folds.

```
cv_results = cross_val_score(modelo, X, y, cv=kfold,
   scoring='f1_weighted')
```

Se calculó el F-Score ponderado en cada fold. Finalmente, se calculó la media y la desviación estándar del F-Score para cada modelo y se imprimió los resultados para comparar el rendimiento de los modelos.

```
msg = f"{nombre:<20} | {cv_results.mean():.4f} | {
   cv_results.std():.4f}"
print(msg)
```

Para una mejor visualización, se creó un boxplot que muestra la distribución de los F-Scores por modelo.

4. Análisis de resultados

Respecto a las matrices de confusión obtenidas(Fig. 1,2 y 3), en los tres casos se obtuvo un buen rendimiento, siendo KNN el que obtuvo mejor F-Score(0.89). Sin embargo, basarse en una sola ejecución del algoritmo podría ser impreciso para estimar su desempeño general. Por lo tanto, se utilizó la técnica de validación cruzada antes descrita. A partir de esto se obtuvo los diagramas de caja de que se muestran en las Figuras 4, 5 y 6.

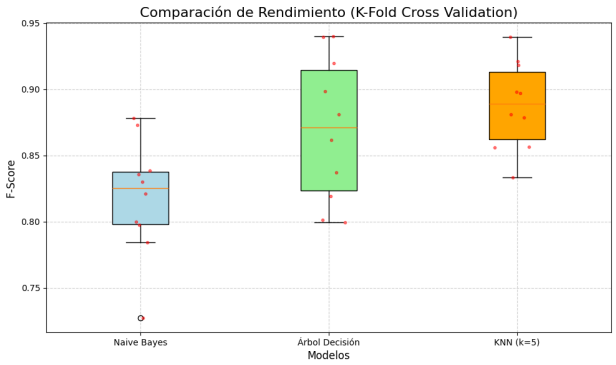

Figura 4: Diagrama de caja obtenido a partir de implementar el código K-Fold Cross Validation, utilizando como dataset el archivo texturas_features_24.csv

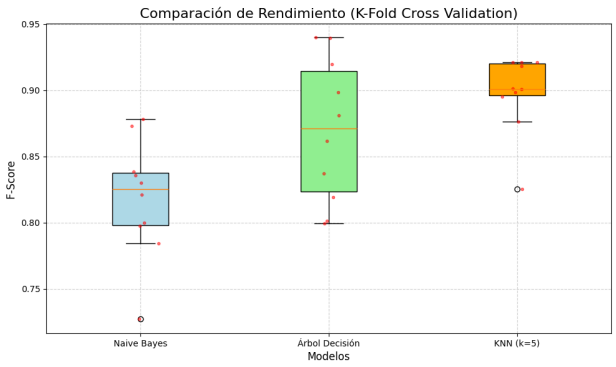

Figura 5: Diagrama de caja obtenido a partir de implementar el código K-Fold Cross Validation, utilizando como dataset el archivo texturas_scaled_24.csv

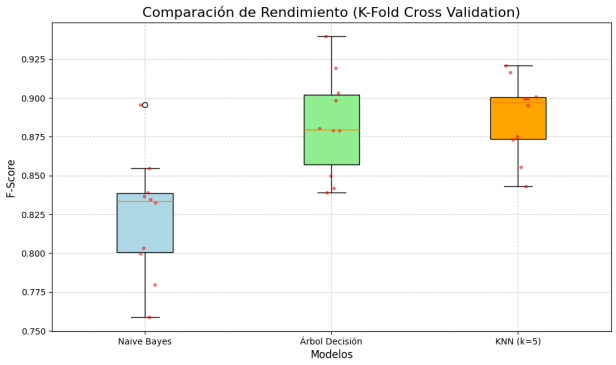

Figura 6: Diagrama de caja obtenido a partir de implementar el código K-Fold Cross Validation, utilizando como dataset el archivo texturas_seleccionadas_optimas.csv

En los tres escenarios, el algoritmo KNN demostró ser el modelo con mejor rendimiento, presentando el F-Score más alto y situándose en la parte superior de los gráficos. Por otro lado, el algoritmo Naive Bayes mostró un menor rendimiento en todos los casos.
Al observar el gráfico 6, que utilizó las 17 características más significativas, se observa una reducción en la dispersión de los resultados (caja más compacta) en comparación con el uso de las 24 características originales. Esto indica que la eliminación de características redundantes mediante SFS aumentó la estabilidad del modelo sin sacrificar su capacidad predictiva.
Al comparar el dataset escalado (Fig. 4) con su versión normalizada (Fig. 5), el rendimiento se mantiene similar. Sin embargo, KNN presenta una leve diferencia, donde la normalización favoreció una menor varianza, compactando los resultados en el diagrama de caja.

En los tres diagramas, el algoritmo Árbol de Decisión posee alta varianza. Esto confirma la naturaleza teórica del algoritmo, que es sensible a pequeñas variaciones en los datos de entrada, y sugiere que el preprocesamiento tiene poco efecto en mitigar su varianza estructural.
La comparación de los tres escenarios permite concluir que el dataset de 17 características seleccionadas y normalizadas (Fig. 6) representa la configuración óptima. Aunque el dataset de 24 características (Fig. 4) alcanza picos de rendimiento superiores, el dataset reducido ofrece mayor estabilidad y eficiencia computacional, ya que reduce la cantidad de datos utilizados sin comprometer significativamente los resultados en comparación con el uso de todas las características.

5. Conclusión

Los resultados obtenidos permitieron caracterizar el impacto del preprocesamiento y de la selección de características en el rendimiento de modelos de clasificación aplicados al reconocimiento de texturas. El clasificador KNN presentó el mayor desempeño global en términos de F-Score promedio, especialmente al utilizar el conjunto reducido de 17 características seleccionadas mediante SFS. El algoritmo Naive Bayes exhibió un rendimiento inferior, lo cual era esperable dado que la independencia condicional asumida por el modelo no se cumplía estrictamente en este caso. Por su parte, el Árbol de decisión mostró una alta varianza entre iteraciones, confirmando su sensibilidad estructural frente a pequeñas perturbaciones en los datos de entrada. La comparación entre los tres escenarios evaluados demostró que la selección de características además de reducir costos computacionales, también mejoró la estabilidad de los clasificadores sin pérdidas significativas de desempeño. El conjunto reducido y normalizado fue la configuración óptima para este caso. En términos generales, los resultados corroboraron la importancia del preprocesamiento, estandarización y selección de características como factores clave para optimizar sistemas de reconocimiento de patrones en imágenes basados en descriptores estadísticos.