

CARACTERISTICAS FUNDAMENTALES: ALGORITMO GENETICO VS PCA

Plata Salinas Eidan Owen

December 6, 2023

Introducción

La elección de las características fundamentales de un dataset es un aspecto crítico en la investigación y análisis de datos en la actualidad. La calidad de los resultados y las conclusiones obtenidas de cualquier estudio depende en gran medida de la selección adecuada de las variables o características que se incluyen en el conjunto de datos. Esta etapa inicial de la investigación de datos es fundamental, ya que determina en gran medida la eficacia de los algoritmos de análisis y las predicciones que se pueden obtener posteriormente [1].

La importancia de la selección de características radica en varios aspectos clave. En primer lugar, una selección cuidadosa de características puede ayudar a reducir la dimensionalidad de un conjunto de datos, lo que a su vez mejora la eficiencia computacional y evita problemas como la maldición de la dimensionalidad. Además, al elegir las características adecuadas, se pueden eliminar variables irrelevantes o ruidosas que podrían introducir sesgos o distorsiones en el análisis. Esto conduce a modelos más precisos y resultados más confiables [2].

Existen numerosos métodos y enfoques para la selección de características, que van desde técnicas estadísticas hasta algoritmos de aprendizaje automático. Algunos de los métodos más comunes incluyen el análisis de componentes principales (PCA), la eliminación de características con baja varianza, la selección basada en la importancia de características, y enfoques de búsqueda exhaustiva o heurística. La elección del método adecuado depende en gran medida del tipo de datos, el objetivo del estudio y las limitaciones computacionales [3].

La elección de las características fundamentales de un dataset es un proceso crucial en la investigación de datos. Requiere un enfoque estratégico y un profundo conocimiento del dominio del problema. La selección cuidadosa de características puede mejorar la calidad de los resultados, reducir la complejidad computacional y aumentar la interpretabilidad de los modelos. Sin embargo, es importante recordar que la selección de características es solo una parte del proceso de análisis de datos, y su eficacia depende de la atención continua a otros aspectos como la limpieza de datos, la ingeniería de características y la validación de modelos. En última instancia, una elección inteligente de características allana el camino hacia la obtención de conocimientos valiosos a partir de los datos [4].

El Análisis de Componentes Principales (PCA) es un poderoso algoritmo de reducción de dimensionalidad ampliamente utilizado en el campo del análisis de datos y la estadística. Su objetivo principal es transformar un conjunto de datos original en un nuevo conjunto de datos, donde las variables originales se reemplazan por un conjunto más pequeño de variables llamadas componentes principales. Estos componentes principales son combinaciones lineales de las variables originales y están diseñados para capturar la mayor variabilidad posible en los datos. PCA es especialmente útil cuando se trabaja con conjuntos de datos de alta dimensionalidad, ya que puede ayudar a reducir la com-

plejidad y a eliminar la multicolinealidad entre las variables. Además de la reducción de dimensionalidad, PCA también se utiliza en aplicaciones como la visualización de datos, la compresión de imágenes y la eliminación de ruido en señales, lo que lo convierte en una herramienta versátil en el análisis de datos y la minería de datos [5].

Los Algoritmos Genéticos (AG) son una familia de técnicas de optimización que pueden utilizarse para abordar la selección de características de manera más directa. Los AG trabajan con un conjunto inicial de características, evalúan diferentes subconjuntos de características en función de un criterio de aptitud y evolucionan hacia subconjuntos óptimos a través de generaciones de selección, cruzamiento y mutación. Esta capacidad para explorar exhaustivamente el espacio de características puede ser especialmente útil cuando se enfrenta a conjuntos de datos complejos y no lineales, y cuando se busca una selección de características más precisa y específica para una tarea en particular [6].

En última instancia, la elección entre PCA y Algoritmos Genéticos en la selección de características depende de las características del problema y de los objetivos del análisis. PCA es útil para reducir la dimensionalidad y mantener la estructura global de los datos, pero puede no ser adecuado cuando se requiere una selección de características más precisa. Los Algoritmos Genéticos, por otro lado, ofrecen un enfoque más flexible y personalizable para la selección de características, pero pueden requerir más recursos computacionales y ajuste de parámetros. En muchos casos, una combinación de ambos enfoques o el uso de otros métodos de selección de características específicos del dominio pueden ser la mejor estrategia. La elección adecuada debe basarse en una comprensión profunda del problema y las características del conjunto de datos en cuestión [7].

Desarrollo

FUNCION CARGAR DATOS

Utiliza `pd.read_csv` para leer el archivo CSV especificado en `ruta_csv`. La opción `header=None` indica que el archivo no tiene fila de encabezado. `X` es asignado a todas las columnas excepto la última, que se supone contienen las características (features) de los datos. es asignado a la última columna, que se supone es la variable objetivo (target). Retorna `X` (características) `Y` y (objetivo).

```
1 def cargar_datos(ruta_csv):
2     data = pd.read_csv(ruta_csv, header=None)
3     X = data.iloc[:, :-1].values
4     y = data.iloc[:, -1].values
5     return X, y
```

FUNCION K FOLD CROSS VALIDATION

Utiliza `StratifiedKFold` para dividir los datos en `k` conjuntos, manteniendo la proporción de las clases en cada partición. Itera sobre los conjuntos de entrenamiento y prueba generados, entrenando y evaluando el modelo en cada iteración. Clona el modelo para cada iteración usando `clone(modelo)` para asegurar que cada evaluación es independiente. Calcula y guarda la precisión (accuracy) del modelo en cada iteración. Retorna una lista de las puntuaciones de precisión obtenidas.

```
1 def k_fold_cross_validation(X, y, k, modelo):
2     skf = StratifiedKFold(n_splits=k, shuffle=True, random_state=42)
3     scores = []
4
5     for train_index, test_index in skf.split(X, y):
6         X_train, X_test = X[train_index], X[test_index]
7         y_train, y_test = y[train_index], y[test_index]
8
9         modelo_clonado = clone(modelo)
10        modelo_clonado.fit(X_train, y_train)
11
12        y_pred = modelo_clonado.predict(X_test)
13        score = accuracy_score(y_test, y_pred)
14        scores.append(score)
15
16    return scores
```

FUNCION GENERAR POBLACION INICIAL

Crea una matriz binaria aleatoria de tamaño `tam_poblacion` x `num_caracteristicas`. Cada fila de la matriz representa un “individuo” en la población, con cada columna representando la presencia o ausencia de una característica (1 o 0). Retorna esta matriz poblacional.

```
1 def generar_poblacion_inicial(num_caracteristicas, tam_poblacion=10):  
2     return np.random.randint(2, size=(tam_poblacion, num_caracteristicas)  
    )
```

FUNCION REPRODUCIR

Para cada nueva generación, selecciona al azar dos padres de la población existente. Genera dos hijos por cada par de padres, dividiendo y combinando sus "genes" (características) en un punto aleatorio. Repite este proceso hasta crear una nueva población de tamaño igual al doble de los individuos originales. Retorna la nueva población.

```
1 def reproducir(individuos):  
2     nuevos_individuos = []  
3     for _ in range(10):  
4         padres = np.random.choice(len(individuos), 2, replace=False)  
5         punto_cruce = np.random.randint(1, len(individuos[0]))  
6         hijo1 = np.concatenate([individuos[padres[0]][:punto_cruce],  
            individuos[padres[1]][punto_cruce:]])  
7         hijo2 = np.concatenate([individuos[padres[1]][:punto_cruce],  
            individuos[padres[0]][punto_cruce:]])  
8         nuevos_individuos.extend([hijo1, hijo2])  
9     return nuevos_individuos
```

FUNCION MUTAR

Itera sobre cada individuo y cada gen (característica) del individuo. Con una probabilidad prob_mutacion, cambia el gen (de 0 a 1 o de 1 a 0). Esto introduce variabilidad en la población, lo cual es clave en los algoritmos genéticos. Retorna la población mutada.

```
1 def mutar(individuos, prob_mutacion=0.1):  
2     for individuo in individuos:  
3         for i in range(len(individuo)):  
4             if np.random.rand() < prob_mutacion:  
5                 individuo[i] = 1 - individuo[i]  
6     return individuos
```

FUNCION EVALUAR INDIVIDUOS

Itera sobre cada individuo de la población. Si un individuo no tiene características seleccionadas (suma igual a 0), su puntuación es 0. Para los demás, selecciona las características correspondientes del conjunto de datos X basándose en los genes (1 o 0) del individuo. Crea un pipeline con escalado estándar y el modelo de clasificación proporcionado. Aplica validación cruzada k-fold al conjunto de datos filtrado y calcula la precisión media. Retorna una lista de puntuaciones, una para cada individuo.

```

1
2 def evaluar_individuos(X, y, individuos, k=5, modelo=LogisticRegression
   ()):
3     puntuaciones = []
4     for individuo in individuos:
5         if np.sum(individuo) == 0:
6             puntuaciones.append(0)
7             continue
8
9         caracteristicas_seleccionadas = X[:, individuo.astype(bool)]
10        pipeline = Pipeline([('scaler', StandardScaler()), ('classifier',
   modelo)])
11        scores = k_fold_cross_validation(caracteristicas_seleccionadas, y,
   k, pipeline)
12        puntuaciones.append(np.mean(scores))
13    return puntuaciones

```

FUNCION SELECCIONAR MEJORES

Ordena a los individuos según sus puntuaciones. Selecciona el número especificado de los mejores individuos (los que tienen las puntuaciones más altas). Retorna estos individuos seleccionados.

```

1
2 def seleccionar_mejores(individuos, puntuaciones, num_seleccionados=10)
   :
3     indices = np.argsort(puntuaciones)[-num_seleccionados:]
4     return [individuos[i] for i in indices]
5

```

FUNCION COMPARAR MODELOS

Entrena y evalúa el modelo proporcionado primero con todas las características (X completo) y luego solo con las características importantes. Utiliza validación cruzada k-fold en ambos casos. Imprime las precisiones medias para cada caso, proporcionando una comparación directa.

```

1
2 def comparar_modelos(X, y, caracteristicas_importantes, modelo=
   LogisticRegression()):
3     pipeline_completo = Pipeline([('scaler', StandardScaler()), ('
   classifier', modelo)])
4     scores_completo = k_fold_cross_validation(X, y, 5, pipeline_completo)
5
6     X_reducido = X[:, caracteristicas_importantes]
7     pipeline_reducido = Pipeline([('scaler', StandardScaler()), ('
   classifier', modelo)])
8     scores_reducido = k_fold_cross_validation(X_reducido, y, 5,
   pipeline_reducido)
9

```

```

10 print(f"Efectividad promedio usando todas las características: {np.
    mean(scores_completo):.2f}")
11 print(f"Efectividad promedio usando características seleccionadas: {
    np.mean(scores_reducido):.2f}")
12

```

FUNCION APLICAR PCA Y COMPARAR

Reduce la dimensionalidad de X a num_componentes utilizando PCA. Entrena y evalúa el modelo de clasificación proporcionado en el conjunto de datos transformado. Utiliza validación cruzada k-fold y calcula la precisión media. Imprime la precisión media y retorna las puntuaciones obtenidas.

```

1
2 def aplicar_pca_y_comparar(X, y, num_componentes, modelo=
    LogisticRegression()):
3     pca = PCA(n_components=num_componentes)
4     X_pca = pca.fit_transform(X)
5
6     pipeline_pca = Pipeline([('scaler', StandardScaler()), ('classifier',
    modelo)])
7     scores_pca = k_fold_cross_validation(X_pca, y, 5, pipeline_pca)
8
9     print(f"Efectividad promedio usando PCA con {num_componentes}
    componentes: {np.mean(scores_pca):.2f}")
10
11     return scores_pca
12

```

FUNCION MAIN

Carga los datos y genera una población inicial para el algoritmo genético. Ejecuta el algoritmo genético para varias generaciones, reproduciendo, mutando y seleccionando los mejores individuos. Al final, identifica las características importantes, compara los modelos con y sin estas características y aplica PCA para una comparación adicional. Finalmente, imprime los resultados relevantes.

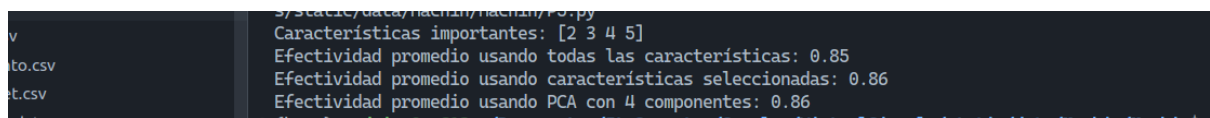
```

1
2 def main(ruta_csv, num_generaciones=10, num_componentes_pca=4):
3     X, y = cargar_datos(ruta_csv)
4     num_caracteristicas = X.shape[1]
5     poblacion = generar_poblacion_inicial(num_caracteristicas)
6
7     for _ in range(num_generaciones):
8         descendientes = reproducir(poblacion)
9         descendientes_mutados = mutar(descendientes)
10        copia_poblacion = mutar(poblacion.copy())
11        poblacion_total = np.vstack((poblacion, copia_poblacion,
    descendientes_mutados))
12        puntuaciones = evaluar_individuos(X, y, poblacion_total)

```

```
13     poblacion = seleccionar_mejores(poblacion_total, puntuaciones)
14
15     mejor_individuo = poblacion[0]
16     caracteristicas_importantes = np.where(mejor_individuo == 1)[0]
17     print("Características importantes:", caracteristicas_importantes)
18
19     comparar_modelos(X, y, caracteristicas_importantes)
20     scores_pca = aplicar_pca_y_comparar(X, y, num_componentes_pca)
21
```

Resultados



```

v          s/static/data/nachin/nachin/v3.py
to.csv    Características importantes: [2 3 4 5]
et.csv    Efectividad promedio usando todas las características: 0.85
          Efectividad promedio usando características seleccionadas: 0.86
          Efectividad promedio usando PCA con 4 componentes: 0.86
          Effectiveness using all characteristics: 0.85
          Effectiveness using selected characteristics: 0.86
          Effectiveness using PCA with 4 components: 0.86

```

Figure 1: Resultados de la ejecución

Referencias

[1] Diccionario. “Dataset: Qué es y principales características”. The Data Schools. Accedido el 22 de noviembre de 2023. [En línea]. Disponible: <https://thedata.schools.com/que-es/data-set/>

[2] Blog Tecnico. “Algoritmo PCA: De lo mucho a lo poco — LIS Data Solutions”. LIS Data Solutions. Accedido el 22 de noviembre de 2023. [En línea]. Disponible: <https://www.lisdatasolutions.com/es/blog/algoritmo-pca-de-lo-mucho-a-lo-poco/>

[3] Amazon Web Services. “Algoritmo de analisis de componente principal (PCA) - Amazon SageMaker”. Accedido el 22 de noviembre de 2023. [En línea]. Disponible: https://docs.aws.amazon.com/es_es/sagemaker/latest/dg/pca.html

[4] J. Alquicira. “Análisis de componentes principales (PCA)”. Conogasi. Accedido el 22 de noviembre de 2023. [En línea]. Disponible: <https://conogasi.org/articulos/analisis-de-componentes-principales-pca/>

[5] SITIOBIGDATA. “Interpretar dataset en machine learning - sitiobigdata.com”. sitiobigdata.com. Accedido el 22 de noviembre de 2023. [En línea]. Disponible: <https://sitiobigdata.com/dataset-en-machine-learning/>

[6] Microsoft. “DataSets - ADO.NET”. Microsoft Learn: Build skills that open doors in your career. Accedido el 22 de noviembre de 2023. [En línea]. Disponible: <https://learn.microsoft.com/es-es/dotnet/framework/data/adonet/ado-net-datasets>

[7] Conogasi. “Algoritmos genéticos”. Conogasi. Accedido el 22 de noviembre de 2023. [En línea]. Disponible: <https://conogasi.org/articulos/algoritmos-geneticos/>

1 Còdigo completo

```

1
2 import csv
3 import pandas as pd
4 from sklearn.model_selection import train_test_split
5 import numpy as np
6
7 class CascadeSVM:
8     def __init__(self, learning_rate=0.01, n_iterations=1000, lambda_param
        =0.1):
9         self.clf_dh = LinearSVM(learning_rate=learning_rate, n_iterations=
            n_iterations, lambda_param=lambda_param)
10        self.clf_sl = LinearSVM(learning_rate=learning_rate, n_iterations=
            n_iterations, lambda_param=lambda_param)
11
12    def print_weights(self):
13        print("Weights for DH Classifier:")
14        print("-----")
15        for idx, w in enumerate(self.clf_dh.weights):
16            print(f"Feature {idx+1}: {w:.5f}")
17        print(f"Bias (DH Classifier): {self.clf_dh.bias:.5f}\n")
18
19        print("Weights for SL Classifier:")
20        print("-----")
21        for idx, w in enumerate(self.clf_sl.weights):
22            print(f"Feature {idx+1}: {w:.5f}")
23        print(f"Bias (SL Classifier): {self.clf_sl.bias:.5f}")
24
25    def fit(self, X, y):
26        y_dh = np.where(y == 'DH', 1, -1)
27        self.clf_dh.fit(X, y_dh)
28
29        mask = y != 'DH'
30        X_sl = X[mask]
31        y_sl = np.where(y[mask] == 'SL', 1, -1)
32        self.clf_sl.fit(X_sl, y_sl)
33
34    def predict(self, X):
35        preds = []
36
37        dh_predictions = self.clf_dh.predict(X)
38        sl_predictions = self.clf_sl.predict(X)
39
40        for idx, (dh_pred, sl_pred) in enumerate(zip(dh_predictions,
            sl_predictions)):
41            if dh_pred == 1:
42                label = 'DH'
43            elif sl_pred == 1:
44                label = 'SL'
45            else:
46                label = 'NO'
47            preds.append(label)
48        print(f"Data {idx+1}: {X.iloc[idx].values} -> Predicted label: {label}"
            )
49
50    return np.array(preds)

```

```

51
52 class LinearSVM:
53     def __init__(self, learning_rate=0.001, n_iterations=1000, lambda_param
        =0.1):
54         self.learning_rate = learning_rate
55         self.n_iterations = n_iterations
56         self.lambda_param = lambda_param
57         self.weights = None
58         self.bias = None
59
60     def fit(self, X, y):
61         n_samples, n_features = X.shape
62         self.weights = np.zeros(n_features)
63         self.bias = 0
64
65         for _ in range(self.n_iterations):
66             for idx, x_i in enumerate(X.values):
67                 condition = y[idx] * (np.dot(x_i, self.weights) - self.bias) >= 1
68                 if condition:
69                     dw = 2 * self.lambda_param * self.weights
70                     db = 0
71                 else:
72                     dw = 2 * self.lambda_param * self.weights - np.dot(x_i, y[idx])
73                     db = y[idx]
74                 self.weights -= self.learning_rate * dw
75                 self.bias -= self.learning_rate * db
76
77     def predict(self, X):
78         linear_output = np.dot(X, self.weights) - self.bias
79         return np.sign(linear_output)
80
81     def dat_to_csv(dat_filename, csv_filename):
82         with open(dat_filename, 'r') as dat_file:
83             lines = dat_file.readlines()
84         with open(csv_filename, 'w', newline='') as csv_file:
85             writer = csv.writer(csv_file, delimiter=',')
86             for line in lines:
87                 data = line.split()[:]
88                 writer.writerow(data)
89
90     def csv_datos(nombre_archivo):
91         datos = []
92         with open(nombre_archivo, newline='') as archivo_csv:
93             lector_csv = csv.reader(archivo_csv, delimiter=',')
94             for fila in lector_csv:
95                 datos.append(','.join(fila))
96
97         csv_data = '\n'.join(datos)
98         return csv_data
99
100    def process_csv(data: str):
101        from io import StringIO
102        data_io = StringIO(data)
103        df = pd.read_csv(data_io, header=None)
104        X = df.iloc[:, :-1]
105        y = df.iloc[:, -1]
106        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
            =0.2, random_state=42)

```

```
107
108 return X_train, X_test, y_train, y_test
109
110 dat_filename = 'datata.dat'
111 csv_filename = 'dataset.csv'
112 dat_to_csv(dat_filename, csv_filename)
113 X_train, X_test, y_train, y_test = process_csv(csv_dados('dataset.csv')
114 )
115 clf_cascade = CascadeSVM(learning_rate=0.01, n_iterations=1000,
116     lambda_param=0.1)
117 clf_cascade.fit(X_train, y_train)
118 clf_cascade.print_weights()
119 y_pred = clf_cascade.predict(X_test)
120 accuracy = np.mean(y_pred == y_test)
121 print(f"Accuracy: {accuracy * 100:.2f}%")
```