Redes Neuronales

Ejercicio 1:

```python
import numpy as np # type: ignore
import pandas as pd # type: ignore
from sklearn.neural_network import MLPRegressor # type: ignore
from sklearn.model_selection import KFold # type: ignore
from sklearn.metrics import mean_squared_error, r2_score # type: ignore

# Cargar el archivo CSV
data = pd.read_csv('crime_data.csv')


x = data[['M', 'W', 'S', 'P']].values
y = data['MR'].values
reg = MLPRegressor(hidden_layer_sizes=(10, 10), max_iter=10000)
reg.fit(x, y)

kf = KFold(n_splits=5, shuffle=True)

cv_y_test = []
cv_y_pred = []

for train_index, test_index in kf.split(x, y):
    # Fase de entrenamiento
    x_train = x[train_index, :]
    y_train = y[train_index]

    reg_i = MLPRegressor(hidden_layer_sizes=(10, 10), max_iter=10000)
    reg_i.fit(x_train, y_train)

    # Fase de prueba
    x_test = x[test_index, :]
    y_test = y[test_index]
    y_pred = reg_i.predict(x_test)

    cv_y_test.append(y_test)
    cv_y_pred.append(y_pred)

# Calcular el error cuadrático medio y el coeficiente de determinación R2
y_test_concat = np.concatenate(cv_y_test)
y_pred_concat = np.concatenate(cv_y_pred)

print("MSE:", mean_squared_error(y_test_concat, y_pred_concat))
print("R2:", r2_score(y_test_concat, y_pred_concat))
```

```
MSE: 67.99105879995845
R2: 0.39624763172949773
```

¿Consideras que el modelo perceptrón multicapa es efectivo para modelar los datos del problema? ¿Por qué? No, tiene un error cuadrado medio de 67, el cual es excesivo, y el R2 nos da 0.396, bastante lejos de 1 lo cual nos da un nivel bajo de exactitud.

¿Qué modelo es mejor para los datos de criminalidad, el lineal o el perceptrón multicapa? ¿Por qué? Los datos parecen ser lineales, por lo que un perceptrón multicapa no es el indicado para este tipo de problemas.

Ejercicio 2:

```python
import numpy as np # type: ignore
import pandas as pd # type: ignore
from sklearn.neural_network import MLPRegressor # type: ignore
from sklearn.model_selection import GridSearchCV, KFold # type: ignore
from sklearn.metrics import mean_squared_error, r2_score # type: ignore

data = pd.read_csv('M_4.txt', delim_whitespace=True, header=None)

x = data.iloc[:, 2:].values
y = data.iloc[:, 0].values

param_grid = {
    'hidden_layer_sizes': [(20,) * 5],
    'activation': ['relu', 'tanh', 'logistic'],
    'solver': ['adam', 'sgd'],
    'alpha': [0.0001,0.0002,0.0003],
    'max_iter': [10000, 20000, 30000]
}

mlp = MLPRegressor()

kf = KFold(n_splits=5, shuffle=True, random_state=42)
grid_search = GridSearchCV(estimator=mlp, param_grid=param_grid, cv=kf, scoring='neg_mean_squared_error', n_jobs=-1)

grid_search.fit(x, y)

print("Mejores parámetros encontrados:")
best_params = grid_search.best_params_
print(best_params)

best_model = grid_search.best_estimator_

cv_results = grid_search.cv_results_
mean_test_scores = cv_results['mean_test_score']
std_test_scores = cv_results['std_test_score']

print("\nResultados de la búsqueda:")
for mean_score, std_score, params in zip(mean_test_scores, std_test_scores, cv_results['params']):
    print(f"Parámetros: {params} | MSE: {-mean_score:.4f} (+/- {std_score:.4f})")

y_pred = best_model.predict(x)
print("\nEvaluación del mejor modelo:")
print("MSE:", mean_squared_error(y, y_pred))
print("R2:", r2_score(y, y_pred))

print(f"Parámetros óptimos: {best_params}")
```

```
Resultados de la búsqueda:
Parámetros: {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'adam'} | MSE: 1.0010 (+/- 0.0871)
Parámetros: {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'sgd'} | MSE: 1.1508 (+/- 0.1614)
Parámetros: {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'adam'} | MSE: 0.9990 (+/- 0.0869)
Parámetros: {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'sgd'} | MSE: 1.0712 (+/- 0.0915)
Parámetros: {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'adam'} | MSE: 1.0606 (+/- 0.1667)
Parámetros: {'activation': 'relu', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'sgd'} | MSE: 1.1074 (+/- 0.2266)
Parámetros: {'activation': 'relu', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'adam'} | MSE: 1.0590 (+/- 0.1841)
Parámetros: {'activation': 'relu', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'sgd'} | MSE: 1.1447 (+/- 0.0874)
Parámetros: {'activation': 'relu', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'adam'} | MSE: 1.1183 (+/- 0.1908)
Parámetros: {'activation': 'relu', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'sgd'} | MSE: 0.9804 (+/- 0.0198)
Parámetros: {'activation': 'relu', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'adam'} | MSE: 1.0574 (+/- 0.1965)
Parámetros: {'activation': 'relu', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'sgd'} | MSE: 1.0918 (+/- 0.0815)
Parámetros: {'activation': 'relu', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'adam'} | MSE: 1.1144 (+/- 0.2271)
Parámetros: {'activation': 'relu', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'sgd'} | MSE: 1.0944 (+/- 0.0937)
Parámetros: {'activation': 'relu', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'adam'} | MSE: 1.0166 (+/- 0.0737)
Parámetros: {'activation': 'relu', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'sgd'} | MSE: 1.1019 (+/- 0.1660)
Parámetros: {'activation': 'relu', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'adam'} | MSE: 0.9657 (+/- 0.1487)
Parámetros: {'activation': 'relu', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'sgd'} | MSE: 1.1477 (+/- 0.0808)
Parámetros: {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'adam'} | MSE: 0.9694 (+/- 0.1818)
Parámetros: {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'sgd'} | MSE: 1.0232 (+/- 0.2747)
Parámetros: {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'adam'} | MSE: 1.1851 (+/- 0.1569)
Parámetros: {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'sgd'} | MSE: 1.2923 (+/- 0.1498)
Parámetros: {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'adam'} | MSE: 1.1276 (+/- 0.1982)
Parámetros: {'activation': 'tanh', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'sgd'} | MSE: 1.0865 (+/- 0.2370)
Parámetros: {'activation': 'tanh', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'adam'} | MSE: 1.0989 (+/- 0.1207)
Parámetros: {'activation': 'tanh', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'sgd'} | MSE: 1.1149 (+/- 0.2855)
Parámetros: {'activation': 'tanh', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'adam'} | MSE: 1.1487 (+/- 0.2563)
Parámetros: {'activation': 'tanh', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'sgd'} | MSE: 1.1143 (+/- 0.2257)
Parámetros: {'activation': 'tanh', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'adam'} | MSE: 1.2365 (+/- 0.3275)
Parámetros: {'activation': 'tanh', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'sgd'} | MSE: 1.0647 (+/- 0.1823)
Parámetros: {'activation': 'tanh', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'adam'} | MSE: 0.9550 (+/- 0.1781)
Parámetros: {'activation': 'tanh', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'sgd'} | MSE: 1.1227 (+/- 0.1078)
Parámetros: {'activation': 'tanh', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'adam'} | MSE: 0.9943 (+/- 0.1527)
Parámetros: {'activation': 'tanh', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'sgd'} | MSE: 1.1822 (+/- 0.2453)
Parámetros: {'activation': 'tanh', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'adam'} | MSE: 0.9776 (+/- 0.1127)
Parámetros: {'activation': 'tanh', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'sgd'} | MSE: 1.1725 (+/- 0.2782)
Parámetros: {'activation': 'logistic', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'adam'} | MSE: 1.6896 (+/- 1.1228)
Parámetros: {'activation': 'logistic', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'sgd'} | MSE: 4.0035 (+/- 0.2005)
Parámetros: {'activation': 'logistic', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'adam'} | MSE: 1.7273 (+/- 1.2166)
Parámetros: {'activation': 'logistic', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'sgd'} | MSE: 4.0073 (+/- 0.2033)
Parámetros: {'activation': 'logistic', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'adam'} | MSE: 2.2388 (+/- 1.4941)
Parámetros: {'activation': 'logistic', 'alpha': 0.0001, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'sgd'} | MSE: 4.0093 (+/- 0.2000)
Parámetros: {'activation': 'logistic', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'adam'} | MSE: 1.7046 (+/- 1.2056)
Parámetros: {'activation': 'logistic', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'sgd'} | MSE: 4.0083 (+/- 0.2033)
Parámetros: {'activation': 'logistic', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'adam'} | MSE: 1.0952 (+/- 0.1830)
Parámetros: {'activation': 'logistic', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'sgd'} | MSE: 4.0115 (+/- 0.2061)
Parámetros: {'activation': 'logistic', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'adam'} | MSE: 2.7880 (+/- 1.5592)
Parámetros: {'activation': 'logistic', 'alpha': 0.0002, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'sgd'} | MSE: 4.0027 (+/- 0.2006)
Parámetros: {'activation': 'logistic', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'adam'} | MSE: 2.1860 (+/- 1.5217)
Parámetros: {'activation': 'logistic', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'sgd'} | MSE: 4.0027 (+/- 0.1983)
Parámetros: {'activation': 'logistic', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 20000, 'solver': 'adam'} | MSE: 1.2119 (+/- 0.2458)
Parámetros: {'activation': 'logistic', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 30000, 'solver': 'sgd'} | MSE: 4.0098 (+/- 0.2051)

Evaluación del mejor modelo:
MSE: 0.34093139706635567
R2: 0.9147671507334111
Parámetros óptimos: {'activation': 'tanh', 'alpha': 0.0003, 'hidden_layer_sizes': (20, 20, 20, 20, 20), 'max_iter': 10000, 'solver': 'adam'}
```

¿Observas alguna mejora importante al optimizar el tamaño de la red? ¿Es el resultado que esperabas? Argumenta tu respuesta.

Dependiendo de los parámetros podemos observar MSEs que varían de 0.34 hasta 4, por lo que el encontrar los valores y parámetros óptimos nos da mucho mejores resultados.

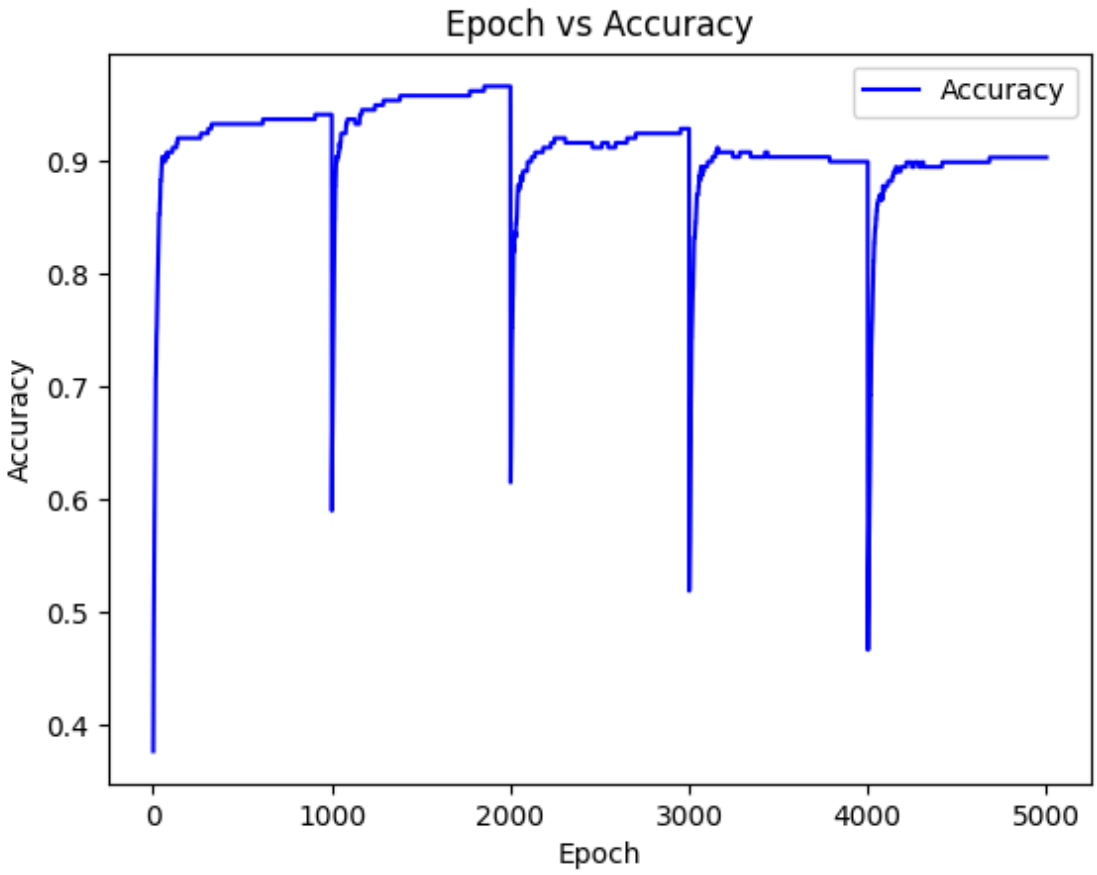
¿Qué inconvenientes hay al encontrar el tamaño óptimo de la red? ¿Por qué?

Al crecer la red neuronal estamos añadiendo tiempo computacional, el cual en una computadora de escritorio, incluso de gama alta, puede convertirse en un problema. No solo esto si no que al crecer más la red neuronal podría caer en un sobreajuste a los datos.

Ejercicio 3:

```python
import numpy as np  # type: ignore
import pandas as pd  # type: ignore
import torch  # type: ignore
import torch.nn as nn  # type: ignore
import torch.optim as optim  # type: ignore
from sklearn import datasets  # type: ignore
from sklearn.model_selection import StratifiedKFold  # type: ignore
from sklearn.metrics import accuracy_score  # type: ignore
import matplotlib.pyplot as plt  # type: ignore

data = pd.read_csv('P1_4.txt', delim_whitespace=True, header=None)

x = data.iloc[:, 2:].values
y = data.iloc[:, 0].values

unique_classes = np.unique(y)
class_map = {cls: idx for idx, cls in enumerate(unique_classes)}
y = np.array([class_map[cls] for cls in y])

x = torch.tensor(x, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.long)

class SingleNeuronPerceptron(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(SingleNeuronPerceptron, self).__init__()
        self.fc = nn.Linear(input_dim, output_dim)

    def forward(self, x):
        return self.fc(x)

loss_function = nn.CrossEntropyLoss()

num_epochs = 1000
learning_rate = 0.01
input_dim = x.shape[1]
output_dim = len(np.unique(y.numpy()))

accuracy_list = []
```

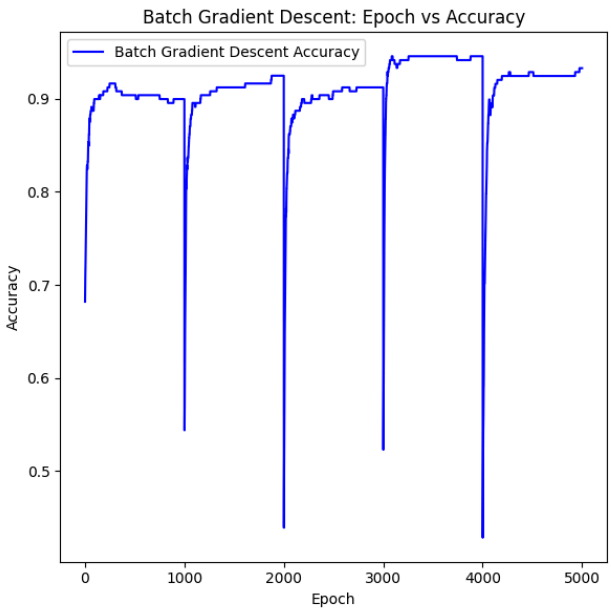
```python
kf = StratifiedKFold(n_splits=5, shuffle=True)

for fold, (train_index, test_index) in enumerate(kf.split(x, y), 1):
    x_train = x[train_index, :]
    y_train = y[train_index]
    x_test = x[test_index, :]
    y_test = y[test_index]

    model = SingleNeuronPerceptron(input_dim, output_dim)
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    # Training phase
    for epoch in range(num_epochs):
        model.train()
        outputs = model(x_train)
        loss = loss_function(outputs, y_train)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Calculate accuracy
        model.eval()
        with torch.no_grad():
            test_outputs = model(x_test)
            _, y_pred = torch.max(test_outputs.data, 1)
            accuracy = accuracy_score(y_test.numpy(), y_pred.numpy())
            accuracy_list.append(accuracy)

        # Print results
        if (epoch+1) % 10 == 0:
            print(f"Fold {fold}, Epoch {epoch+1}, Loss: {loss.item()}, Accuracy: {accuracy}")

epochs = range(1, len(accuracy_list) + 1)
plt.plot(epochs, accuracy_list, 'b-', label='Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Epoch vs Accuracy')
plt.legend()
plt.show()
```

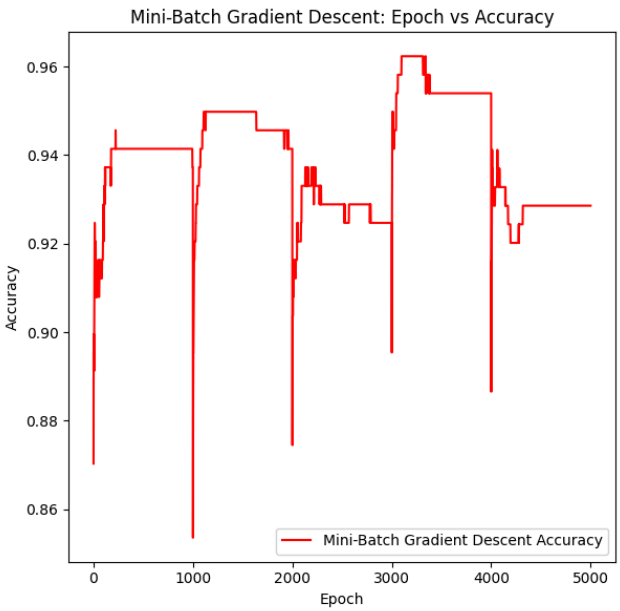



```python
import numpy as np # type: ignore
import pandas as pd # type: ignore
import torch # type: ignore
import torch.nn as nn # type: ignore
import torch.optim as optim # type: ignore
from sklearn.model_selection import StratifiedKFold # type: ignore
from sklearn.metrics import accuracy_score # type: ignore
import matplotlib.pyplot as plt # type: ignore
from torch.utils.data import DataLoader, TensorDataset # type: ignore

# Load data
data = pd.read_csv('P1_4.txt', delim_whitespace=True, header=None)

x = data.iloc[:, 2:].values
y = data.iloc[:, 0].values

unique_classes = np.unique(y)
class_map = {cls: idx for idx, cls in enumerate(unique_classes)}
y = np.array([class_map[cls] for cls in y])

x = torch.tensor(x, dtype=torch.float32)
y = torch.tensor(y, dtype=torch.long)

class SingleNeuronPerceptron(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(SingleNeuronPerceptron, self).__init__()
        self.fc = nn.Linear(input_dim, output_dim)

    def forward(self, x):
        return self.fc(x)

loss_function = nn.CrossEntropyLoss()

num_epochs = 1000
learning_rate = 0.01
batch_size = 32
input_dim = x.shape[1]
output_dim = len(unique_classes)

accuracy_list_batch = []
accuracy_list_mini_batch = []

```

```
kf = StratifiedKFold(n_splits=5, shuffle=True)

for fold, (train_index, test_index) in enumerate(kf.split(x, y), 1):
    # Separate training and test data
    x_train = x[train_index, :]
    y_train = y[train_index]
    x_test = x[test_index, :]
    y_test = y[test_index]

    train_dataset = TensorDataset(x_train, y_train)
    train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)

    # Batch Gradient Descent
    model = SingleNeuronPerceptron(input_dim, output_dim)
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    # Training phase
    for epoch in range(num_epochs):
        model.train()
        outputs = model(x_train)
        loss = loss_function(outputs, y_train)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Calculate accuracy
        model.eval()
        with torch.no_grad():
            test_outputs = model(x_test)
            _, y_pred = torch.max(test_outputs.data, 1)
            accuracy = accuracy_score(y_test.numpy(), y_pred.numpy())
            accuracy_list_batch.append(accuracy)

        # Print results
        if (epoch+1) % 10 == 0:
            print(f"Fold {fold}, Batch Gradient Epoch {epoch+1}, Loss: {loss.item()}, Accuracy: {accuracy}")

    # Mini-Batch Gradient Descent
    model = SingleNeuronPerceptron(input_dim, output_dim)
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)
```

```
    # Training phase
    for epoch in range(num_epochs):
        model.train()
        for batch_x, batch_y in train_loader:
            outputs = model(batch_x)
            loss = loss_function(outputs, batch_y)

            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        # Calculate accuracy
        model.eval()
        with torch.no_grad():
            test_outputs = model(x_test)
            _, y_pred = torch.max(test_outputs.data, 1)
            accuracy = accuracy_score(y_test.numpy(), y_pred.numpy())
            accuracy_list_mini_batch.append(accuracy)

        # Print results
        if (epoch+1) % 10 == 0:
            print(f"Fold {fold}, Mini-Batch Gradient Epoch {epoch+1}, Loss: {loss.item()}, Accuracy: {accuracy}")

epochs_batch = range(1, len(accuracy_list_batch) + 1)
epochs_mini_batch = range(1, len(accuracy_list_mini_batch) + 1)

plt.figure(figsize=(12, 6))

plt.subplot(1, 2, 1)
plt.plot(epochs_batch, accuracy_list_batch, 'b-', label='Batch Gradient Descent Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Batch Gradient Descent: Epoch vs Accuracy')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(epochs_mini_batch, accuracy_list_mini_batch, 'r-', label='Mini-Batch Gradient Descent Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.title('Mini-Batch Gradient Descent: Epoch vs Accuracy')
plt.legend()

plt.tight_layout()
plt.show()
```





¿El modelo de una neurona es suficiente para modelar el conjunto de datos de este problema?

Sí, después de varios epochs casi todas las precisiones eran de arriba de 90, llegando incluso hasta 0.96, por lo que los modelos hicieron un muy buen trabajo de predecir los resultados de nuestro dataset.

Link del codigo:
https://drive.google.com/file/d/1FwpCMIyvPV2p_DaThzclk1I8vDKWgoKa/view?usp=sharing