Instituto Tecnológico y de
Estudios Superiores de Monterrey

Evidencia 1:

Proyecto Minería de Datos

Escuela de Ingeniería y Ciencias
Modelación del Aprendizaje con Inteligencia Artificial (TC2034.301)

Miembros

Leonardo De Regil Cardenas	A00837118
Diego Armando Mijares Ledezma	A01722421
José Francisco Obregón Gaxiola	A00227502
Alexei Carrillo Acosta	A01285424

13 de junio de 2024

Docente: Dr. Eduardo Ramírez

e1-supervised-machine-learning-1

June 16, 2024

0.1 # Project: Supervised Machine Learning - Algorithm Exploration and Comparison

Original Code:

https://colab.research.google.com/drive/1ORbMohKEgPkOFgJHiIZKrwLYxROeo-HE?usp=sharing

Database utilized:

https://drive.google.com/file/d/15S4bKEGj5onu6SBIkvo9i6-5Xzcao0Dn/view?usp=sharing

0.2 # Data Description

The database we used for this project is called 'Heart Disease'. The database contains 14 columns with information on age, sex, type of chest pain, resting blood pressure, serum cholesterol, fasting blood sugar, resting electrocardiographic results, maximum heart rate achieved, exercise-induced angina, ST depression induced by exercise relative to rest, the slope of the peak exercise ST segment, thalassemia, and our categorical variable indicating the severity of heart disease.

1. **age**: Age in years
2. **sex**: Sex (1 = male; 0 = female)
3. **cp**: Chest pain type
   - Value 1: Typical angina
   - Value 2: Atypical angina
   - Value 3: Non-anginal pain
   - Value 4: Asymptomatic
4. **trestbps**: Resting blood pressure (in mm Hg on admission to the hospital)
5. **chol**: Serum cholesterol in mg/dl
6. **fbs**: Fasting blood sugar > 120 mg/dl (1 = true; 0 = false)
7. **restecg**: Resting electrocardiographic results
   - Value 0: Normal
   - Value 1: Having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of > 0.05 mV)
   - Value 2: Showing probable or definite left ventricular hypertrophy by Estes' criteria
8. **thalach**: Maximum heart rate achieved
9. **exang**: Exercise-induced angina (1 = yes; 0 = no)
10. **oldpeak**: ST depression induced by exercise relative to rest
11. **slope**: The slope of the peak exercise ST segment
    - Value 1: Upsloping
    - Value 2: Flat

- Value 3: Downsloping

12. **ca**: Number of major vessels (0-3) colored by fluoroscopy
13. **thal**: Thalassemia
    - Value 3: Normal
    - Value 6: Fixed defect
    - Value 7: Reversible defect
14. **num**: Diagnosis of heart disease (angiographic disease status)
    - Value 0: Absence of heart disease
    - Value 1, 2, 3, 4: Varying degrees of presence and severity

1 Libraries

[ ]:
```
# Import libraries
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, r2_score
from sklearn.utils import check_random_state, check_array
from sklearn.svm import LinearSVC, LinearSVR, SVC, SVR, _libsvm
from sklearn.preprocessing import LabelEncoder
import matplotlib.pyplot as plt
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, precision_score, recall_score,␣
 ↪f1_score, confusion_matrix, make_scorer
from sklearn.tree import DecisionTreeClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score, KFold
from sklearn.model_selection import train_test_split, cross_validate
```

[ ]:
```
# We previously replaced the null values with the median of the column.

# Import "heart_disease" Database.
#df_path = "/content/heart_disease.csv"
df_path = "/content/dataset.csv"

df_heart_disease = pd.read_csv(df_path)
```

2 First Look at the Database

[ ]:
```
# Here we show the first 5 lines of the database.
df_heart_disease.head()
```

[ ]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	num
0	63	1	1	145	233	1	2	150	0	2.3	3	0.0	6.0	0
1	67	1	4	160	286	0	2	108	1	1.5	2	3.0	3.0	2
2	67	1	4	120	229	0	2	129	1	2.6	2	2.0	7.0	1
3	37	1	3	130	250	0	0	187	0	3.5	3	0.0	3.0	0
4	41	0	2	130	204	0	2	172	0	1.4	1	0.0	3.0	0

3 General Information

[ ]:
```
# Basic stadistic information.
df_heart_disease.describe()
```

[ ]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	num
count	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000	303.000000
mean	54.438944	0.679868	3.158416	131.689769	246.693069	0.148515	0.990099	149.607261	0.326733	1.039604	1.600660	0.663366		
std	9.038662	0.467299	0.960126	17.599748	51.776918	0.356198	0.994971	22.875003	0.469794	1.161075	0.616226	0.934375		
min	29.000000	0.000000	1.000000	94.000000	126.000000	0.000000	0.000000	71.000000	0.000000	0.000000	1.000000	0.000000		
25%	48.000000	0.000000	3.000000	120.000000	211.000000	0.000000	0.000000	133.500000	0.000000	0.000000	1.000000	0.000000		
50%	56.000000	1.000000	3.000000	130.000000	241.000000	0.000000	1.000000	153.000000	0.000000	0.800000	2.000000	0.000000		
75%	61.000000	1.000000	4.000000	140.000000	275.000000	0.000000	2.000000	166.000000	1.000000	1.600000	2.000000	1.000000		
max	77.000000	1.000000	4.000000	200.000000	564.000000	1.000000	2.000000	202.000000	1.000000	6.200000	3.000000	3.000000		

```
mean     4.722772    0.937294
std      1.938383    1.228536
min      3.000000    0.000000
25%      3.000000    0.000000
50%      3.000000    0.000000
75%      7.000000    2.000000
max      7.000000    4.000000
```

[ ]:
```
# Database shape.
shape = df_heart_disease.shape
print(f"The database has the shape: {shape}")
```

```
The database has the shape: (303, 14)
```

3.1 # Problem Description

Machine learning is at the forefront of medical research, aiding in the identification of diseases and the diagnosis of conditions. According to a 2015 report published by Pharmaceutical Research and Manufacturers of America, more than 800 drugs and vaccines were being tested to treat cancer. In an interview with Bloomberg Technology, Jeff Tyner, a researcher at the Knight Institute, stated that while this is exciting, it also presents the challenge of finding ways to work with all the resulting data. "That's where the idea of biologists and doctors working with data scientists and computer scientists is very important," said Tyner.

In this document we try to diagnose the presence and severity of a heart disease, with the use of machine learning algorithms such as, decision tree, logistic regression, K-nearest neighbor, and support vector machine. We show the algorithm generated from scratch and compare it with the algorithm generated by scikit-learn.

3.2 # Model Implementation

3.3 Spliting the Data

[ ]:
```
# Here we split the data.
X, y = df_heart_disease.iloc[:, :-1], df_heart_disease.iloc[:, -1]
y = pd.Series(LabelEncoder().fit_transform(y))
X_train, X_test, y_train, y_test = train_test_split(X.values, y.values,␣
 ↪test_size=0.3, random_state=0)
```

3.4 ## Decision Tree

A Decision Tree is a supervised learning algorithm used in machine learning for classification. This model is represented by roots and leafs, and the paths from the roots to the leafs represents classification rules. And each leaf node represents a class label.

One of the advantages of Decision Trees is that they are intuitive and easy to interpret. However, they can be prone to overfitting if not properly configured, for example if they are allowed to grow too much.

[ ]:
```
class Node():
    def __init__(self, feature_index=None, threshold=None, left=None,␣
 ↪right=None, info_gain=None, value=None):
        ''' Constructor to initialize a node'''

        # For decision node
        self.feature_index = feature_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.info_gain = info_gain

        # for leaf node
        self.value = value


class MyDecisionTreeClassifier():
    def __init__(self, min_samples_split=2, max_depth=2):
        ''' Constructor to initialize the decision tree'''

        self.root = None  # root node of the tree
        self.min_samples_split = min_samples_split    # minimum number of␣
 ↪samples required to split an internal node
        self.max_depth = max_depth    # maximum depth of the tree

    def build_tree(self, dataset, curr_depth=0):
        ''' Recursive function to build the tree '''

        X, Y = dataset[:,:-1], dataset[:,-1]  # split the dataset into features␣
 ↪and labels
        num_samples, num_features = np.shape(X)  # get the number of samples␣
 ↪and features
        num_labels = len(np.unique(Y))  # get the number of labels#

        # Check if the current node can be split further
        if num_samples>=self.min_samples_split and curr_depth<=self.max_depth:
            # Find the best split
            best_split = self.get_best_split(dataset, num_samples, num_features)

            # If a valid split is found
            if best_split["info_gain"]>0:
                # Build the left and right subtrees recursively
                left_subtree = self.build_tree(best_split["dataset_left"],␣
 ↪curr_depth+1)
                right_subtree = self.build_tree(best_split["dataset_right"],␣
 ↪curr_depth+1)
                # Return the current node with left and right subtrees
```

```
            return Node(best_split["feature_index"],␣
↪best_split["threshold"],
                        left_subtree, right_subtree,␣
↪best_split["info_gain"])

        # If splitting is not possible, create a leaf node
        leaf_value = self.calculate_leaf_value(Y)

        return Node(value=leaf_value)

    def get_best_split(self, dataset, num_samples, num_features):
        ''' Function to find the best split for the dataset '''


        best_split = {} # Dictionary to store the best split
        max_info_gain = -float("inf")  # Initialize maximum information gain

        # Iterate over all features
        for feature_index in range(num_features):
            feature_values = dataset[:, feature_index]  # Get all values of the␣
↪feature
            possible_thresholds = np.unique(feature_values) # Get unique␣
↪threshold values

            # Iterate over all possible thresholds
            for threshold in possible_thresholds:
                # Split the dataset based on the threshold
                dataset_left, dataset_right = self.split(dataset,␣
↪feature_index, threshold)
                # Check if both splits are non-empty
                if len(dataset_left)>0 and len(dataset_right)>0:
                    y, left_y, right_y = dataset[:, -1], dataset_left[:, -1],␣
↪dataset_right[:, -1]
                    # Calculate information gain
                    curr_info_gain = self.information_gain(y, left_y, right_y,␣
↪"gini")

                    # Update the best split if current information gain is␣
↪greater
                    if curr_info_gain>max_info_gain:
                        best_split["feature_index"] = feature_index
                        best_split["threshold"] = threshold
                        best_split["dataset_left"] = dataset_left
                        best_split["dataset_right"] = dataset_right
                        best_split["info_gain"] = curr_info_gain
                        max_info_gain = curr_info_gain
```

```
        return best_split

    def split(self, dataset, feature_index, threshold):
        ''' Function to split the data '''

        dataset_left = np.array([row for row in dataset if␣
↪row[feature_index]<=threshold])
        dataset_right = np.array([row for row in dataset if␣
↪row[feature_index]>threshold])
        return dataset_left, dataset_right

    def information_gain(self, parent, l_child, r_child, mode="entropy"):
        ''' Function to split the dataset based on a feature and threshold '''

        weight_l = len(l_child) / len(parent) # Left child
        weight_r = len(r_child) / len(parent)# Right child
        if mode=="gini":
          # Calculate information gain using Gini index
            gain = self.gini_index(parent) - (weight_l*self.gini_index(l_child)␣
↪+ weight_r*self.gini_index(r_child))
        else:
          # Calculate information gain using entropy
            gain = self.entropy(parent) - (weight_l*self.entropy(l_child) +␣
↪weight_r*self.entropy(r_child))
        return gain

    def entropy(self, y):
        ''' Function to compute entropy '''

        class_labels = np.unique(y)   # Get unique class labels
        entropy = 0

        # Calculate entropy
        for cls in class_labels:
            p_cls = len(y[y == cls]) / len(y) # Proportion of class label
            entropy += -p_cls * np.log2(p_cls)
        return entropy

    def gini_index(self, y):
        ''' Function to compute gini index '''

        class_labels = np.unique(y) # Get unique class labels
        gini = 0

        # Calculate Gini index
```

```
        for cls in class_labels:
            p_cls = len(y[y == cls]) / len(y)
            gini += p_cls**2
        return 1 - gini

    def calculate_leaf_value(self, Y):
        ''' Function to compute leaf node '''

        Y = list(Y)
        return max(Y, key=Y.count) # Return the most common class label

    def print_tree(self, tree=None, indent=" "):
        ''' Function to print the tree '''

        if not tree:
            tree = self.root # Start from the root node if no tree is provided

        if tree.value is not None:
            print(tree.value)

        else:
            print("X_"+str(tree.feature_index), "<=", tree.threshold, "?", tree.
↪info_gain)
            print("%sleft:" % (indent), end="")
            self.print_tree(tree.left, indent + indent) # Print left subtree
            print("%sright:" % (indent), end="")
            self.print_tree(tree.right, indent + indent)  # Print right subtree

    def fit(self, X, Y):
        ''' Function to train the tree '''

        dataset = np.concatenate((X, Y), axis=1)  # Concatenate features and␣
↪labels
        self.root = self.build_tree(dataset)  # Build the tree

    def predict(self, X):
        ''' Function to predict new dataset '''

        preditions = [self.make_prediction(x, self.root) for x in X]  # Predict␣
↪for each sample
        return preditions

    def make_prediction(self, x, tree):
        ''' Function to predict a single data point '''

        if tree.value!=None: return tree.value  # Return value if leaf node
        feature_val = x[tree.feature_index]
```

```
        if feature_val<=tree.threshold:
            return self.make_prediction(x, tree.left)
        else:
            return self.make_prediction(x, tree.right)
```

[ ]:
```
# Fill the NA values with the median of the column
for columna in df_heart_disease.columns:
    mediana = df_heart_disease[columna].median()
    df_heart_disease[columna].fillna(mediana, inplace=True)
```

[ ]:
```
# Splits the dataset in train and test, also Y is reshaped
X = df_heart_disease.iloc[:, :-1].values
Y = df_heart_disease.iloc[:, -1].values.reshape(-1,1)
from sklearn.model_selection import train_test_split
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=.2,␣
 ↪random_state=41)
```

[ ]:
```
classifier = MyDecisionTreeClassifier(min_samples_split=3, max_depth=3)
classifier.fit(X_train,Y_train)
classifier.print_tree()
```

```
X_12 <= 3.0 ? 0.08453547529659733
 left:X_11 <= 0.0 ? 0.052931165315667106
  left:X_0 <= 57.0 ? 0.02229400148528221
    left:X_9 <= 3.0 ? 0.02932012648809519
        left:0.0
        right:1.0
    right:X_7 <= 71.0 ? 0.05565003779289479
        left:2.0
        right:0.0
  right:X_2 <= 3.0 ? 0.0807426192041577
    left:X_4 <= 236.0 ? 0.07369614512471656
        left:0.0
        right:0.0
    right:X_9 <= 2.0 ? 0.12918871252204578
        left:1.0
        right:3.0
 right:X_9 <= 0.8 ? 0.0631670972928784
  left:X_4 <= 240.0 ? 0.09022973640275589
    left:X_3 <= 135.0 ? 0.10583333333333333
        left:0.0
        right:1.0
    right:X_2 <= 3.0 ? 0.07804032766225577
        left:0.0
        right:1.0
  right:X_7 <= 132.0 ? 0.04064612664902878
    left:X_3 <= 144.0 ? 0.06625043357613591
```

```
        left:3.0
        right:4.0
    right:X_7 <= 147.0 ? 0.10449928693171939
        left:2.0
        right:0.0
```

[ ]:
```
y_pred = classifier.predict(X_test)

accuracy = accuracy_score(Y_test, y_pred)
precision = precision_score(Y_test, y_pred, average='weighted')
f1 = f1_score(Y_test, y_pred, average='weighted')
conf_matrix = confusion_matrix(Y_test, y_pred)

print("Accuracy:", accuracy)
print("Precision:", precision)
print("F1:", f1)
print("Confusion Matrix:")
print(conf_matrix)
```

```
Accuracy: 0.6229508196721312
Precision: 0.57472359893252
F1: 0.5967476463229968
Confusion Matrix:
[[35  3  2  0  0]
 [ 6  0  0  4  1]
 [ 2  1  0  0  0]
 [ 0  1  3  2  0]
 [ 0  0  0  0  1]]
```

3.5 SKLearn Decision Tree

[ ]:
```
lassifier = DecisionTreeClassifier()
lassifier.fit(X_train, Y_train)
predictions = lassifier.predict(X_test)
```

[ ]:
```
accuracy = accuracy_score(Y_test, predictions)
precision = precision_score(Y_test, predictions, average='weighted')
f1 = f1_score(Y_test, predictions, average='weighted')
conf_matrix = confusion_matrix(Y_test, predictions)


print("Accuracy:", accuracy)
print("Precision:", precision)
print("F1:", f1)
print("Confusion Matrix:")
print(conf_matrix)
```

```
Accuracy: 0.5245901639344263
```

```
Precision: 0.6284729284464874
F1: 0.5646848302932347
Confusion Matrix:
[[26  7  6  1  0]
 [ 4  3  0  3  1]
 [ 1  2  0  0  0]
 [ 0  1  1  2  2]
 [ 0  0  0  0  1]]
```

[ ]:
```python
scoring = {
    'accuracy': 'accuracy',
    'precision': make_scorer(precision_score, average='weighted'),
    'recall': make_scorer(recall_score, average='weighted'),
    'f1': make_scorer(f1_score, average='weighted')
}
cv_results1 = cross_validate(lassifier, X_train, Y_train, cv=5, scoring=scoring)

# Print cross-validation results
print("Cross-Validation Results:")
print("Accuracy: ", np.mean(cv_results1['test_accuracy']))
print("Precision:", np.mean(cv_results1['test_precision']))
print("Recall:   ", np.mean(cv_results1['test_recall']))
print("F1-score: ", np.mean(cv_results1['test_f1']))

# Train the model on the full training set
lassifier.fit(X_train, Y_train)

# Make predictions on the test set
predictions1 = lassifier.predict(X_test)

# Calculate evaluation metrics on the test set
accuracy = accuracy_score(Y_test, predictions1)
precision = precision_score(Y_test, predictions1, average='weighted')
recall = recall_score(Y_test, predictions1, average='weighted')
f1 = f1_score(Y_test, predictions1, average='weighted')
conf_matrix1 = confusion_matrix(Y_test, predictions1)

# Print evaluation metrics on the test set
print("\nTest Set Evaluation Metrics:")
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)
print("Confusion Matrix:")
print(conf_matrix1)
```

```
Cross-Validation Results:
Accuracy:  0.40867346938775506
```

```
Precision: 0.4444803928686964
Recall:    0.40867346938775506
F1-score:  0.41654457542299594

Test Set Evaluation Metrics:
Accuracy: 0.5573770491803278
Precision: 0.6077733741668168
Recall: 0.5573770491803278
F1-score: 0.5762295081967211
Confusion Matrix:
[[30  5  5  0  0]
 [ 4  3  1  2  1]
 [ 1  2  0  0  0]
 [ 0  3  1  0  2]
 [ 0  0  0  0  1]]
```

3.6 #### Decision Tree Comparison and Conclusion

Result Comparison

DT accuracy: 0.6230
sk DT accuracy: 0.5245

DT F1-score: 0.5967
sk DT F1-score: 0.5646

Conclusion
The Decision Tree from scratch outperforms the Decision Tree from sklearn in therms of accuracy and F1-scorde. Higher accuracy means that the model makes fewer mistakes. In this case, the DT from scratch has an accuracy of 0.6230, which is higher than the sklearn DT accuracy of 0.5245. In terms of F1-score a higher F1-score indicates better performance. The DT from scratch has 0.5967, which is higher but no much than DT from sklearn, with an F1-score of 0.5646. In conclusion, the DT from scratch seems to perform better than the sklearn DT model.And this is more evident if we do cross validation, which the results show us that it radically decreases the accuracy with 4087 and an F1-score of 0.4165

3.7 ## Logistic Regression

Logistic Regression is a supervised machine learning algorithm used for binary classification tasks. It models the probability of a binary outcome using a logistic function (sigmoid function). The algorithm estimates the relationship between a dependent binary variable and one or more independent variables by fitting a logistic curve to the data. It outputs probabilities that can be converted to binary decisions, making it useful for tasks like spam detection, medical diagnosis, and credit scoring.

[ ]:
```python
# Logistic Regression Implementation
class MyLogisticRegression:
    def __init__(self, alpha=0.01, num_iters=1000):
        self.alpha = alpha
```

```
        self.num_iters = num_iters
        self.theta = None

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def compute_cost(self, X, y):
        m = len(y)
        h = self.sigmoid(X.dot(self.theta))
        cost = -(1/m) * np.sum(y * np.log(h) + (1 - y) * np.log(1 - h))
        return cost

    def gradient_descent(self, X, y):
        m = len(y)
        self.theta = np.zeros(X.shape[1])
        cost_history = []

        for _ in range(self.num_iters):
            h = self.sigmoid(X.dot(self.theta))
            gradient = (1/m) * X.T.dot(h - y)
            self.theta = self.theta - self.alpha * gradient
            cost = self.compute_cost(X, y)
            cost_history.append(cost)

        return self.theta, cost_history

    def fit(self, X, y):
        X_bias = np.c_[np.ones((X.shape[0], 1)), X]
        self.theta, self.cost_history = self.gradient_descent(X_bias, y)

    def predict_prob(self, X):
        X_bias = np.c_[np.ones((X.shape[0], 1)), X]
        return self.sigmoid(X_bias.dot(self.theta))

    def predict(self, X):
        prob = self.predict_prob(X)
        return [1 if p >= 0.5 else 0 for p in prob]
```

[ ]:
```
# Feature Scaling
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

[ ]:
```
model = MyLogisticRegression(alpha=0.01, num_iters=1000)
model.fit(X_train, y_train)
y_pred = model.predict(X_test)
```

```
<ipython-input-23-97eae066d7e6>:14: RuntimeWarning: divide by zero encountered
in log
  cost = -(1/m) * np.sum(y * np.log(h) + (1 - y) * np.log(1 - h))
```

[ ]:
```
# Fit the model and make predictions on the  dataset ...
# Calculate Precission
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Show Classification Report
print("Classification Report:")
print(classification_report(y_test, y_pred))

# Show Confusion Matrix
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

```
Accuracy: 48.35%
Classification Report:
              precision    recall  f1-score   support

           0       0.77      0.70      0.73        47
           1       0.23      0.58      0.33        19
           2       0.00      0.00      0.00        12
           3       0.00      0.00      0.00         9
           4       0.00      0.00      0.00         4

    accuracy                           0.48        91
   macro avg       0.20      0.26      0.21        91
weighted avg       0.44      0.48      0.45        91

Confusion Matrix:
[[33 14  0  0  0]
 [ 8 11  0  0  0]
 [ 1 11  0  0  0]
 [ 1  8  0  0  0]
 [ 0  4  0  0  0]]
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

SkLearn Logistic Regression

[ ]:
```
# Create an instance of the Logistic Regression model
log_reg = LogisticRegression(random_state=42)

# Train the model
log_reg.fit(X_train, y_train)
```

[ ]:
```
LogisticRegression(random_state=42)
```

[ ]:
```
# Make predictions on the test set
y_pred = log_reg.predict(X_test)

# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy * 100:.2f}%")

# Display the classification report
print("Classification Report:")
print(classification_report(y_test, y_pred))

# Display the confusion matrix
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred))
```

```
Accuracy: 58.24%
Classification Report:
              precision    recall  f1-score   support

           0       0.72      0.91      0.80        47
           1       0.36      0.21      0.27        19
           2       0.50      0.08      0.14        12
           3       0.28      0.56      0.37         9
           4       0.00      0.00      0.00         4

    accuracy                           0.58        91
   macro avg       0.37      0.35      0.32        91
weighted avg       0.54      0.58      0.53        91

Confusion Matrix:
[[43  2  1  1  0]
 [11  4  0  4  0]
```

```
 [ 3  2  1  6  0]
 [ 2  2  0  5  0]
 [ 1  1  0  2  0]]
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision and F-score are ill-defined and being set to
0.0 in labels with no predicted samples. Use `zero_division` parameter to
control this behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

3.8 ## Logistic Regression Comparison and Conclusion

Result Comparison

DT accuracy: 0.4835
sk DT accuracy: 0.5824

Conclusion
In this comparison, the logistic regression model implemented using scikit-learn achieved better performance with an accuracy of 58.24% compared to the from-scratch implementation, which achieved an accuracy of 48.35%. In summary, the scikit-learn logistic regression model would be the recommended choice for predicting heart disease due to its better performance and ease of use.

3.9 ## K-Nearest Neighbor

It is a versatile supervised learning algorithm used for classification and regression tasks. It classifies data points based on the majority class among their nearest neighbors (for classification) or predicts values based on the average of nearest neighbors (for regression). It doesn't require explicit training, instead, it memorizes the entire training dataset.

[ ]:
```python
# K-Nearest Neighbors Implementation
class MyKNN:
    def __init__(self, k=5):
        self.k = k
        self.X_train = None
        self.y_train = None

    def euclidean_distance(self, x1, x2):
        return np.sqrt(np.sum((x1 - x2)**2))
```

```
    def fit(self, X_train, y_train):
        self.X_train = StandardScaler().fit_transform(X_train)
        self.y_train = y_train

    def predict(self, X_test):
        X_test_scaled = StandardScaler().fit_transform(X_test)
        predictions = []
        for x_test in X_test_scaled:
            distances = []
            for i, x_train in enumerate(self.X_train):
                distance = self.euclidean_distance(x_test, x_train)
                distances.append((i, distance))
            distances.sort(key=lambda x: x[1])
            neighbors = [i for i, _ in distances[:self.k]]
            labels = [self.y_train[i] for i in neighbors]
            prediction = max(set(labels), key=labels.count)
            predictions.append(prediction)
        return predictions
```

[ ]:
```
# Create an instance of MyKNN
knn_model = MyKNN()
```

[ ]:
```
# Train the model
knn_model.fit(X_train, y_train)
```

[ ]:
```
# Make predictions on the test set
predictions = knn_model.predict(X_test)
```

[ ]:
```
# Calculate the accuracy of the model
accuracy = accuracy_score(y_test, predictions)
print("Accuracy:", accuracy)

# Calculate the confusion matrix of the model
conf_matrix = confusion_matrix(y_test, predictions)
print("Confusion Matrix:")
print(conf_matrix)

# Calculate precision, recall, and F1-score with average='weighted'
precision = precision_score(y_test, predictions, average='weighted')
recall = recall_score(y_test, predictions, average='weighted')
f1 = f1_score(y_test, predictions, average='weighted')

print("Precision (weighted):", precision)
print("Recall (weighted):", recall)
print("F1-score (weighted):", f1)
```

```
Accuracy: 0.6153846153846154
Confusion Matrix:
[[45  2  0  0  0]
 [11  6  2  0  0]
 [ 3  4  3  2  0]
 [ 1  4  2  2  0]
 [ 1  1  1  1  0]]
Precision (weighted): 0.5437144340712325
Recall (weighted): 0.6153846153846154
F1-score (weighted): 0.567817896389325

/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels
with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

3.10 SKlearn K-Nearest Neighbor

[ ]:
```python
# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

[ ]:
```python
# Create an instance of KNeighborsClassifier - (scikit learn)
knn_model = KNeighborsClassifier(n_neighbors=5)

# Train the model
knn_model.fit(X_train_scaled, y_train)

# Make predictions on the test set
predictions = knn_model.predict(X_test_scaled)
```

[ ]:
```python
# Calculate evaluation metrics
accuracy = accuracy_score(y_test, predictions)
precision = precision_score(y_test, predictions, average='weighted')
recall = recall_score(y_test, predictions, average='weighted')
f1 = f1_score(y_test, predictions, average='weighted')
conf_matrix = confusion_matrix(y_test, predictions)

# Print evaluation metrics
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)
print("Confusion Matrix:")
print(conf_matrix)
```

```
Accuracy: 0.6373626373626373
Precision: 0.5591204430490144
Recall: 0.6373626373626373
F1-score: 0.5834473219088603
Confusion Matrix:
[[46  1  0  0  0]
 [11  7  0  1  0]
 [ 4  3  4  1  0]
 [ 1  4  3  1  0]
 [ 1  1  1  1  0]]

/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels
with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

Cross Validation KNN

[ ]:
```
# Perform cross-validation
# Define scoring metrics
scoring = {
    'accuracy': 'accuracy',
    'precision': make_scorer(precision_score, average='weighted'),
    'recall': make_scorer(recall_score, average='weighted'),
    'f1': make_scorer(f1_score, average='weighted')
}
cv_results = cross_validate(knn_model, X_train_scaled, y_train, cv=5,␣
 ↪scoring=scoring)

# Print cross-validation results
print("Cross-Validation Results:")
print("Accuracy: ", np.mean(cv_results['test_accuracy']))
print("Precision:", np.mean(cv_results['test_precision']))
print("Recall:   ", np.mean(cv_results['test_recall']))
print("F1-score: ", np.mean(cv_results['test_f1']))

# Train the model on the full training set
knn_model.fit(X_train_scaled, y_train)

# Make predictions on the test set
predictions = knn_model.predict(X_test_scaled)

# Calculate evaluation metrics on the test set
accuracy = accuracy_score(y_test, predictions)
precision = precision_score(y_test, predictions, average='weighted')
recall = recall_score(y_test, predictions, average='weighted')
f1 = f1_score(y_test, predictions, average='weighted')
```

```python
conf_matrix = confusion_matrix(y_test, predictions)

# Print evaluation metrics on the test set
print("\nTest Set Evaluation Metrics:")
print("Accuracy:", accuracy)
print("Precision:", precision)
print("Recall:", recall)
print("F1-score:", f1)
print("Confusion Matrix:")
print(conf_matrix)
```

```
Cross-Validation Results:
Accuracy:  0.5895902547065337
Precision: 0.5272558721430298
Recall:    0.5895902547065337
F1-score:  0.5411714643911797

Test Set Evaluation Metrics:
Accuracy: 0.6373626373626373
Precision: 0.5591204430490144
Recall: 0.6373626373626373
F1-score: 0.5834473219088603
Confusion Matrix:
[[46  1  0  0  0]
 [11  7  0  1  0]
 [ 4  3  4  1  0]
 [ 1  4  3  1  0]
 [ 1  1  1  1  0]]
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels
with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels
with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels
with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels
with no predicted samples. Use `zero_division` parameter to control this
behavior.
```

```
  _warn_prf(average, modifier, msg_start, len(result))
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344:
UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels
with no predicted samples. Use `zero_division` parameter to control this
behavior.
  _warn_prf(average, modifier, msg_start, len(result))
```

3.11 ## K-Nearest Neighbor Comparison and Conclusion

Result Comparison

DT accuracy: 0.6153
sk DT accuracy: 0.6373

Conclusion
As we can see the scikit-learn implementation of K-Nearest Neighbor outperforms the from-scratch implementation, showing higher precision, recall, and F1-score. Therefore, for this classification task, using the scikit-learn implementation would be preferred due to its better performance and reliability.

3.12 ## Support Vector Machine

The main idea of SVM (Support Vector Machine) is to find the hyperplane or line that best separates the classes, and the samples closest to the separating hyperplane are called support vectors. The best class separation would be the one that maximizes the distance between the support vectors and the separating hyperplane. This distance is called the margin.

[ ]:
```python
class LinearSVM:
    def __init__(self, regression=False, C=1.0, eps=0, learning_rate=0.001,␣
 ↪max_iter=1000,
                 random_state=0):
        self.regression = regression
        self.C = C
        self.eps = eps
        self.learning_rate = learning_rate
        self.max_iter = max_iter
        self.random_state = random_state

    def fit(self, X, y):
        if self.regression:
            self.bias, self.weights = self._find_weights(X, y)
        else:
            classes = np.unique(y)
            n_classes = len(classes)
            _, n_features = X.shape

            self.bias = np.zeros(n_classes)
            self.weights = np.zeros((n_classes, n_features))
```

```
            np.random.seed(self.random_state)

            for i, cls in enumerate(classes):
                y_binary = np.where(y == cls, 1, -1)
                self.bias[i], self.weights[i] = self._find_weights(X, y_binary)

    def _find_weights(self, X, y):
        n_samples, n_features = X.shape
        bias = 0
        weights = np.zeros(n_features) if self.regression else np.random.
↪randn(n_features)

        for _ in range(self.max_iter):
            for i in range(n_samples):
                y_pred = X[i] @ weights + bias
                margin = y[i] - y_pred if self.regression else y[i] * y_pred
                condition = np.abs(margin) > self.eps if self.regression else␣
↪margin < 1

                if condition:
                    if self.regression:
                        db = -self.C * (margin - self.eps)
                        dw = -self.C * (margin - self.eps) * X[i]
                    else:
                        db = -self.C * y[i]
                        dw = -self.C * y[i] * X[i]

                    bias -= self.learning_rate * db
                    weights -= self.learning_rate * dw

        return bias, weights

    def predict(self, X):
        scores = X @ self.weights.T + self.bias

        return scores if self.regression else np.argmax(scores, axis=1)
```

[ ]:
```
class GaussianSVM:
    # Initialization method for the GaussianSVM class
    def __init__(self, regression=False, C=1.0, kernel='rbf', degree=3,␣
↪solver='auto',
                 gamma='scale', epsilon=0.1, coef0=0.0, shrinking=True,␣
↪probability=False,
                 tol=0.001, cache_size=200, max_iter=-1, random_state=None):
        # Store whether the SVM is for regression or classification
        self.regression = regression
        # Regularization parameter
```

```
        self.C = C
        # Kernel type (e.g., 'rbf', 'linear')
        self.kernel = kernel
        # Degree for polynomial kernel
        self.degree = degree
        # Solver type (e.g., 'auto', 'c_svc')
        self.solver = solver
        # Kernel coefficient
        self.gamma = gamma
        # Epsilon margin for regression
        self.epsilon = epsilon
        # Independent term in kernel function
        self.coef0 = coef0
        # Whether to use the shrinking heuristic
        self.shrinking = shrinking
        # Whether to enable probability estimates
        self.probability = probability
        # Tolerance for stopping criterion
        self.tol = tol
        # Cache size in MB
        self.cache_size = cache_size
        # Maximum number of iterations (-1 for no limit)
        self.max_iter = max_iter
        # Seed for random number generation to ensure reproducibility
        self.random_state = random_state

    # Method to fit the model to the data
    def fit(self, X, y):
        # Ensure the input data is of type float64
        X = X.astype(np.float64)
        y = y.astype(np.float64)

        # Get a random state based on the given random seed
        rnd = check_random_state(self.random_state)
        seed = rnd.randint(np.iinfo('i').max)

        # Set gamma value based on input parameter
        if self.gamma == 'scale':
            # Gamma is 1 / (number of features * variance of X)
            self.gamma = 1.0 / (X.shape[1] * X.var()) if X.var() != 0 else 1.0
        elif self.gamma == 'auto':
            # Gamma is 1 / number of features
            self.gamma = 1.0 / X.shape[1]
        else:
            # Use the provided gamma value
            self.gamma = self.gamma
```

```
        # Automatically select the solver if set to 'auto'
        if self.solver == 'auto':
            # Use 'epsilon_svr' for regression, 'c_svc' for classification
            self.solver = 'epsilon_svr' if self.regression else 'c_svc'

        # List of supported solver implementations
        libsvm_impl = ['c_svc', 'nu_svc', 'one_class', 'epsilon_svr', 'nu_svr']
        # Get the index of the solver in the implementation list
        self.solver = libsvm_impl.index(self.solver)

        # Call the internal _libsvm fit function to train the model
        (self.support_, self.support_vectors_, self._n_support, self.
↪dual_coef_, self.intercept_,
         self._probA, self._probB, self.fit_status_, self._num_iter
        ) = _libsvm.fit(X, y, C=self.C, svm_type=self.solver, kernel=self.
↪kernel, gamma=self.gamma,
                        degree=self.degree, epsilon=self.epsilon, coef0=self.
↪coef0, tol=self.tol,
                        shrinking=self.shrinking, probability=self.probability,
                        cache_size=self.cache_size, max_iter=self.max_iter,␣
↪random_seed=seed
                        )

    # Method to make predictions using the trained model
    def predict(self, X_test):
        # Ensure the input data is of type float64
        X_test = X_test.astype(np.float64)
        # Call the internal _libsvm predict function to get predictions
        prediction = _libsvm.predict(X_test, self.support_, self.
↪support_vectors_, self._n_support,
                                     self.dual_coef_, self.intercept_, self.
↪_probA, self._probB,
                                     svm_type=self.solver, kernel=self.kernel,␣
↪degree=self.degree,
                                     coef0=self.coef0, gamma=self.gamma,␣
↪cache_size=self.cache_size
                                     )

        # Return prediction directly for regression, convert to int for␣
↪classification
        return prediction if self.regression else prediction.astype(int)
```

[ ]:
```
# Linear SVC

linear_svc = LinearSVM(random_state=0)
linear_svc.fit(X_train, y_train)
```

```
linear_svc_pred_res = linear_svc.predict(X_test)
linear_svc_accuracy = accuracy_score(y_test, linear_svc_pred_res)

print(f'LinearSVC accuracy: {linear_svc_accuracy:}')
print(linear_svc_pred_res)
```

[ ]:
```
# Linear SVC - (scikit-learn)

sk_linear_svc = LinearSVC(loss='squared_hinge', max_iter=10000, random_state=0)
sk_linear_svc.fit(X_train, y_train)
sk_linear_svc_pred_res = sk_linear_svc.predict(X_test)
sk_linear_svc_accuracy = accuracy_score(y_test, sk_linear_svc_pred_res)

print(f'sk LinearSVC accuracy: {sk_linear_svc_accuracy:}')
print(sk_linear_svc_pred_res)
```

[ ]:
```
# GaussianSVC

svc = GaussianSVM(random_state=0, gamma='auto')
svc.fit(X_train, y_train)
svc_pred_res = svc.predict(X_test)
svc_accuracy = accuracy_score(y_test, svc_pred_res)

print(f'SVC accuracy: {svc_accuracy:}')
print(svc_pred_res)
```

[ ]:
```
# GaussianSVC - (scikit-learn)

sk_svc = GaussianSVM(random_state=0, gamma='auto')
sk_svc.fit(X_train, y_train)
sk_svc_pred_res = sk_svc.predict(X_test)
sk_svc_accuracy = accuracy_score(y_test, sk_svc_pred_res)

print(f'sk SVC accuracy: {sk_svc_accuracy:}')
print(sk_svc_pred_res)
```

3.13 ## Support Vector Machine Comparison and Conclusion

Result Comparison
LinearSVC accuracy: 0.275
sk LinearSVC accuracy: 0.286

SVC accuracy: 0.516
sk SVC accuracy: 0.516

Conclusion
A Support Vector Machine algorithm is essentially a quadratic programming problem, meaning it is an optimization problem. In the code, only the linear and Gaussian kernels are used, but we can

notice that with the Gaussian kernel, the results are 'better'; however, all the values delivered by the Gaussian kernel are purely zeros, and the accuracy result it provides (around 0.5) is assumed to be because half of the original data are 0. The linear kernel has better results because, unlike the Gaussian kernel, it gives different results than 0, but it has lower accuracy performance.

The results with the scikit-learn library are very similar as we can see, and the phenomenon of zero results with the Gaussian kernel remains, and the accuracy level with the linear kernel is maintained.

3.14 # Conclusion

As you can see, each of the models shows different performance, both for learning and from scratch. As for the models with the best performance in accuracy from sklearn, it was K-Nearest Neighbor with a value of 0.637 and a cross-validation of 0.5896. While the model from scratch with the best performance was Decision Tree, which had an accuracy value of 0.623. This can be explained as possibly caused by overfitting or model optimization. In general, the model that obtained the best performance was K-Nearest Neighbor, since although it did not have the highest value in the from scratch model it was very close to that result, and comparing its F1-score values, the best model was K-Nearest Neighbor with a value of 0.5678 in terms of models from scratch. And the same model but from sklearn was the best in its area, with a value in its F1-score of 0.5834

3.15 # References

Egazakharenko. (2024, May 12). Support Vector Machines (SVM) from scratch . Kaggle. https://www.kaggle.com/code/egazakharenko/support-vector-machines-svm-from-scratch/notebook

Normalized Nerd. (2021, January 21). Decision Tree Classification in Python (from scratch!). Recovered on June 14, 2024. Recovered from https://youtu.be/sgQAhG5Q7iY?si=GYp66yb51L9CIT4Q

IBM. What is a decision tree?. Recovered on June 14, 2024. Recovered from https://www.ibm.com/topics/decision-trees