OPTION #1: KNN Classifier with Iris Data

Scott Miner

Colorado State University – Global Campus

Abstract

This paper presents a k-Nearest Neighbors classifier (k-NN-C) implemented in Python, which achieves a mean accuracy of 96.67% on the Iris Dataset using 5-fold cross-validation, with the number of nearest neighbors set to 10. The Iris Dataset, comprising three classes and four attributes, serves as a foundation for understanding classification techniques. The k-NN-C model is a lazy learner that leverages Euclidean distance to identify similarities between observation pairs and makes predictions based on the k nearest neighbors. By using cross-validation, the classifier's performance is assessed on mutually exclusive data splits, ensuring a robust evaluation. The implemented Python program reads the Iris Dataset, preprocesses the data, and applies the k-NN-C algorithm to make predictions. Additionally, the program accepts user input for previously unseen iris plant features and generates class predictions based on the model. This work demonstrates the effectiveness of the k-NN-C model on a widely recognized dataset and lays the groundwork for future research in feature normalization and model optimization.

KNN.py

normalize_dataset

```python
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split
# k-nearest neighbors on the Iris Flowers Dataset
from random import seed
from random import randrange
from csv import reader
from math import sqrt
import re

# Load a CSV file
def load_csv(filename):
    dataset = list()
    with open(filename, 'r') as file:
        csv_reader = reader(file)
        for row in csv_reader:
            if not row:
                continue
            dataset.append(row)
    return dataset

# Convert string column to float
def str_column_to_float(dataset, column):
    for row in dataset:
        row[column] = float(row[column].strip())

# Convert string column to integer
def str_column_to_int(dataset, column):
    class_values = [row[column] for row in dataset]
    unique = set(class_values)
    lookup = dict()
    # create dictionary
    for i, value in enumerate(unique):
        lookup[value] = i
    # convert dataset column
    for row in dataset:
        row[column] = lookup[row[column]]
    return lookup

# Find the min and max values for each column
def dataset_minmax(dataset):
    minmax = list()
    for i in range(len(dataset[0])):
        col_values = [row[i] for row in dataset]
        value_min = min(col_values)
        value_max = max(col_values)
        minmax.append([value_min, value_max])
    return minmax

# Rescale dataset columns to the range 0-1
def normalize_dataset(dataset, minmax):
    for row in dataset:
        for i in range(len(row)):
            row[i] = (row[i] - minmax[i][0]) / (minmax[i][1] - minmax[i][0])

# Split a dataset into k folds
def cross_validation_split(dataset, n_folds):
    dataset_split = list()
    dataset_copy = list(dataset)
    fold_size = int(len(dataset) / n_folds)
    for _ in range(n_folds):
        fold = list()
        while len(fold) < fold_size:
            index = randrange(len(dataset_copy))
            fold.append(dataset_copy.pop(index))
        dataset_split.append(fold)
    return dataset_split
```

Figure 1. Python code to implement a *k*-NN-C from scratch (part 1)

KNN.py

evaluate_algorithm

```python
    return dataset_split

# Calculate accuracy percentage
def accuracy_metric(actual, predicted):
    correct = 0
    for i in range(len(actual)):
        if actual[i] == predicted[i]:
            correct += 1
    return correct / float(len(actual)) * 100.0

# Evaluate an algorithm using a cross validation split
def evaluate_algorithm(dataset, algorithm, n_folds, *args):
    folds = cross_validation_split(dataset, n_folds)
    scores = list()
    for fold in folds:
        train_set = list(folds)
        # create hold out set
        train_set.remove(fold)
        #combine train sets
        train_set = sum(train_set, [])
        # create test set on new hold
        test_set = list()
        for row in fold:
            row_copy = list(row)
            test_set.append(row_copy)
            # remove prediction from hold out set
            row_copy[-1] = None
        predicted = algorithm(train_set, test_set, *args)
        actual = [row[-1] for row in fold]
        accuracy = accuracy_metric(actual, predicted)
        scores.append(accuracy)
    return scores

# Calculate the Euclidean distance between two vectors
def euclidean_distance(row1, row2):
    distance = 0.0
    for i in range(len(row1) - 1):
        distance += (row1[i] - row2[i]) ** 2
    return sqrt(distance)

# Locate the most similar neighbors
def get_neighbors(train, test_row, num_neighbors):
    distances = list()
    for train_row in train:
        dist = euclidean_distance(test_row, train_row)
        distances.append((train_row, dist))
    distances.sort(key=lambda tup: tup[1])
    neighbors = list()
    for i in range(num_neighbors):
        neighbors.append(distances[i][0])
    return neighbors

# Make a prediction with neighbors
def predict_classification(train, test_row, num_neighbors):
    neighbors = get_neighbors(train, test_row, num_neighbors)
    output_values = [row[-1] for row in neighbors]
    prediction = max(set(output_values), key=output_values.count)
    return prediction

# kNN Algorithm
def k_nearest_neighbors(train, test, num_neighbors):
    predictions = list()
    for row in test:
        output = predict_classification(train, row, num_neighbors)
        predictions.append(output)
    return(predictions)
```

Figure 2. Python code to implement a *k*-NN-C from scratch (part 2)

```python

# Test the kNN on the Iris Flowers dataset
seed(1)
filename = 'data/iris.txt'
dataset = load_csv(filename)
for i in range(len(dataset[0]) - 1):
     str_column_to_float(dataset[1:], i)
# convert class column to integers
# versicolor : 0
# virginica: 1
# setosa: 2
lookup = str_column_to_int(dataset[1:], len(dataset[0]) - 1)

# evaluate algorithm
n_folds = 5
num_neighbors = 10
scores = evaluate_algorithm(dataset[1:], k_nearest_neighbors, n_folds, num_neighbors)
print(f'**************************************************************************************')
print(f'*')
print(f'*   K-Nearest Neighbor (KNN) algorithm with {num_neighbors} neighbors trained on a ')
print(f'*   dataset containing  {len(dataset)-1} rows and {len(dataset[1])-1} features, using {n_folds}-fold cross validation.')
print(f'*')
print(f'*   Users can adjust the n_folds and num_neighbors variables in the script.')
print(f'*')
print(f'**************************************************************************************')
print()
print(f'Accuracy per fold: {scores}')
print(f'Mean Accuracy: {sum(scores) / float(len(scores)):.3f}')

while True:
    try:
        sl, sw, pl, pw = [float(x) for x in re.split(r'\, | |\,', input('\nPlease input four floating point numbers, representing, respectively,\n\
sepal length, sepal width, petal length, and petal width \n(e.g., 5.1, 3.5, 1.4, 0.2).  The model will guess the flower category\n\
(i.e., setosa, versicolor, or virginica) based on your input (CTRL-C to Exit): '))]
        test_row = [sl, sw, pl, pw]
        test_row
        prediction = predict_classification(train=dataset[1:], test_row=test_row, num_neighbors=num_neighbors)
        for key, value in lookup.items():

               if prediction == value:
                   prediction = key
        print()
        print(f'Prediction: {prediction}')
    except ValueError:
        print('\nNote: wrong input format.')

```

Figure 3. Python code to implement a *k*-NN-C from scratch (part 3)

```
**************************************************************************************
*
*   K-Nearest Neighbor (KNN) algorithm with 10 neighbors trained on a
*   dataset containing  150 rows and 4 features, using 5-fold cross validation.
*
*   Users can adjust the n_folds and num_neighbors variables in the script.
*
**************************************************************************************

Accuracy per fold: [90.0, 100.0, 100.0, 93.33333333333333, 100.0]
Mean Accuracy: 96.667

Please input four floating point numbers, representing, respectively,
sepal length, sepal width, petal length, and petal width
(e.g., 5.1, 3.5, 1.4, 0.2).  The model will guess the flower category
(i.e., setosa, versicolor, or virginica) based on your input (CTRL-C to Exit): 4.9 3.0 1.5 0.1

Prediction: Iris-setosa

Please input four floating point numbers, representing, respectively,
sepal length, sepal width, petal length, and petal width
(e.g., 5.1, 3.5, 1.4, 0.2).  The model will guess the flower category
(i.e., setosa, versicolor, or virginica) based on your input (CTRL-C to Exit): 5.1, 1.9, 3.4, 1.2

Prediction: Iris-versicolor

Please input four floating point numbers, representing, respectively,
sepal length, sepal width, petal length, and petal width
(e.g., 5.1, 3.5, 1.4, 0.2).  The model will guess the flower category
(i.e., setosa, versicolor, or virginica) based on your input (CTRL-C to Exit): 7.8,3.6,6.5,1.8

Prediction: Iris-virginica

Please input four floating point numbers, representing, respectively,
sepal length, sepal width, petal length, and petal width
(e.g., 5.1, 3.5, 1.4, 0.2).  The model will guess the flower category
(i.e., setosa, versicolor, or virginica) based on your input (CTRL-C to Exit):
```

```
5.1,3.8,1.6,0.2,-Iris-setosa
5,2,3.5,1,Iris-versicolor
7.9,3.8,6.4,2,Iris-virginica
```

Figure 4. Program output and predictions based on user input similar to the training examples found on the right-hand side of the image

Table of Contents

OPTION #1: KNN Classifier with Iris Data

This paper presents a k-Nearest Neighbors classifier (k-NN-C) built from scratch in Python, achieving a notable mean accuracy of 96.67% on the renowned Iris Dataset using 5-fold cross-validation (CV). The model's number of nearest neighbors is set to 10. Fenner (2019) discusses the Iris Dataset, often referred to as Fisher's Iris Dataset, named after the eminent mid-20th-century statistician, Sir Ronald Fisher, who pioneered its use in one of the earliest academic papers on classification. The dataset comprises three distinct classes, with 50 instances per class, representing the types of iris plants: (a) Setosa, (b) Versicolor, and (c) Virginica. Four descriptive attributes measure these plants in centimeters: (a) sepal length, (b) sepal width, (c) petal length, and (d) petal width.

User Input and Model Predictions

Upon training the k-NN-C model on the Iris Dataset, the program invites users to input four floating-point numbers, separated by commas or spaces, representing the features of previously unencountered iris plants. Subsequently, the program generates the model's class prediction based on this user input, demonstrating its practical utility in real-world scenarios.

Introduction to the k-Nearest Neighbor Classifier (k-NN-C)

Fenner (2019) highlights k-NN-C as a relatively simple yet effective machine-learning model designed to make predictions using labeled datasets. At its core, k-NN-C assesses similarities between pairs of observations, selects a predefined number of the most similar instances, and combines these findings to generate a single output prediction. While the Euclidean distance is frequently employed to measure similarities among features, alternative metrics like Minkowski and Hamming distances can also be utilized. In k-NN-C, the variable 'k' denotes the number of nearest neighbors that the model relies on for its predictions. Typically, practitioners experiment with values such as 1, 3, 10, and 20 to find the optimal setting.

Understanding the k-Nearest Neighbor Classifier (k-NN-C) Algorithm and Its Unique Characteristics

Fenner (2019) emphasizes the unique characteristics of k-NN-C models in comparison to other machine learning approaches, particularly their reliance on the entirety of the training data when making predictions for new test cases. Consequently, removing any training observations could result in inaccurate predictions, as these records might have been crucial in determining the nearest neighbors for specific test cases. This property classifies k-NN-C as a lazy learner, which, unlike eager learners such as logistic regression, does not have a dedicated training phase. Instead, it retains all training data for use during the prediction phase, making this process more computationally intensive than that of eager learners (*KNN Algorithm - Finding Nearest Neighbors*, n.d.; *Why Is Nearest Neighbor a Lazy Algorithm?*, 2021).

Program and Model Architecture

Figures 1 – 3 illustrate the Python code used to create the k-NN-C model, with much of the code adapted from Brownlee (2019). The program starts by reading the Iris Dataset from the data/iris.txt file using the *load_csv()* function (lines 12 – 20, Figure 1). To ensure the dataset's features are in the correct format, *str_column_to_float()* (lines 28 – 38, Figure 1) converts the features from strings to floating-point numbers, while *str_column_to_int()* (lines 23 – 35, Figure 1) converts the string representation of each iris plant category to integers for easier processing by the k-NN-C model.

Cross-Validation Process

The *cross_validation_split()* method (lines 56 – 67, Figure 1) divides the dataset into five cross-validation (CV) splits. Barrow and Crone (2013) describe CV as a technique to estimate the expected accuracy of a predictive algorithm by averaging predictive errors across mutually exclusive subsamples of the data (p. 1). The function divides the dataset into k mutually exclusive groups of roughly equal size, as determined by the user-defined k value. One fold is reserved for the validation dataset, while the remaining folds form the training data. The model is trained on the training data and evaluated using the validation dataset (Brownlee, 2019).

Barrow and Crone (2013) explain that the CV process is repeated k times, with each fold serving as the validation dataset exactly once. The program records the accuracy scores for each of the k evaluation sessions and calculates the final measure of the model's predictive accuracy by averaging these scores across the k folds. The accuracy_metric() function (lines 70 – 75, Figure 2) calculates the model's accuracy for each fold using the formula: $\frac{\#\text{ correct predictions}}{\#\text{ total predictions}}$. The advantages of *k*-fold CV are that it uses all observations for both training and validation datasets, uses all training observations with equal weight, and uses each observation for validation exactly once.

Generating Predictions and Calculating Euclidean Distance

Since k-NN-C is a lazy learning algorithm, it doesn't have a specialized training phase but instead uses all its training data to generate predictions during the evaluation phase. The *evaluate_algorithm()* method (lines 78 – 98, Figure 2) calls *k_nearest_neighbors()*, which iterates over the test dataset and calls *predict_classification()* for each row of test data. *Predict_classification()* then calls *get_neighbors()*, which in turn calls the *euclidean_distance()* function (lines 101 – 105, Figure 2) to calculate the Euclidean distance between each training example and a given test example using the formula:

$d = \sqrt{(sl_{train} - sl_{test})^2 + (sw_{train} - sw_{test})^2 + (pl_{train} - pl_{test})^2 + (pw_{train} - pw_{test})^2}.$

The variables sl, sw, pl, and pw represent sepal length, sepal width, petal length, and petal width for each training and test example (Brownlee, 2019).

Model Evaluation, User Input, and Future Research

The k-NN-C model achieves a mean accuracy of 96.67% on the Iris Dataset using 5-fold CV, with the number of nearest neighbors set to 10 (Figure 4). The program then allows users to input four floating-point numbers representing the sepal length, sepal width, petal length, and petal width of previously unseen iris plants. For example, by inputting features similar to, but not identical with, the training examples shown in Figure 4, the model accurately predicts each iris plant category. The driver code for user input is shown in lines 134 – 177 of Figure 3. Users must input four valid numerical features, separated by spaces or commas, for the model to generate a valid prediction. If the input is incorrect, the program notifies the user and prompts for new input. Additional functions, like the *normalize_dataset()* function, can be used in future research to assess how normalizing the input features of the Iris Dataset affects the classifier's mean accuracy (Brownlee, 2019).

Conclusion

In conclusion, this paper presented an overview of a *k*-Nearest Neighbor classifier (k-NN-C) built from scratch in Python, achieving 96.67% accuracy on the Iris Dataset using 5-fold cross-validation (CV), with the model's number of neighbors set to 10. The paper provided an overview of the Iris Dataset, k-NN-C models, and CV. The classifier's inner workings were also explained, including the calculation of Euclidean distance for each training example per test example, sorting of results, and summation of frequencies of the *k* nearest neighbor classes to produce predictions. The program also accepts user input for iris plant features previously unseen by the model and generates predictions based on the training data.

References

Barrow, D. K., & Crone, S. F. (2013). Crogging (cross-validation aggregation) for forecasting—A novel algorithm of neural network ensembles on time series subsamples. *The 2013 International Joint Conference on Neural Networks (IJCNN)*, 1–8.

Brownlee, J. (2019, October 23). Develop k-Nearest Neighbors in Python from Scratch. *Machine Learning Mastery*. https://machinelearningmastery.com/tutorial-to-implement-k-nearest-neighbors-in-python-from-scratch/

Fenner, M. (2019). *Machine Learning in Python for Everyone*. Addison-Wesley.

KNN Algorithm—Finding Nearest Neighbors. (n.d.). Retrieved August 28, 2021, from https://www.tutorialspoint.com/machine_learning_with_python/machine_learning_with_python_knn_algorithm_finding_nearest_neighbors.htm

Why is Nearest Neighbor a Lazy Algorithm? (2021, August 25). Dr. Sebastian Raschka. https://sebastianraschka.com/faq/docs/lazy-knn.html