OPTION #1: KNN Classifier with Iris Data

Scott Miner

Colorado State University – Global Campus

Abstract

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
```

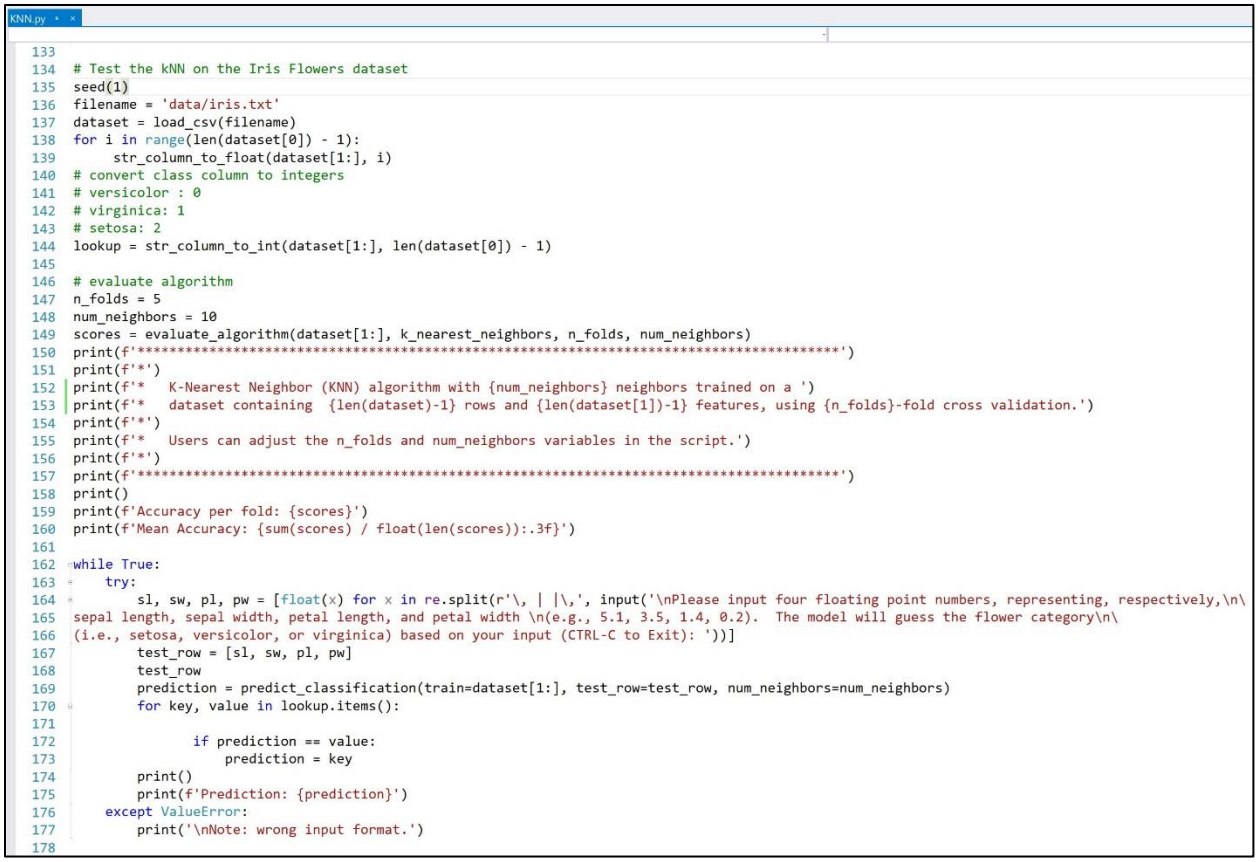
Figure 1. Python code to implement a *k*-NN-C from scratch (part 1)

KNN.py

evaluate_algorithm

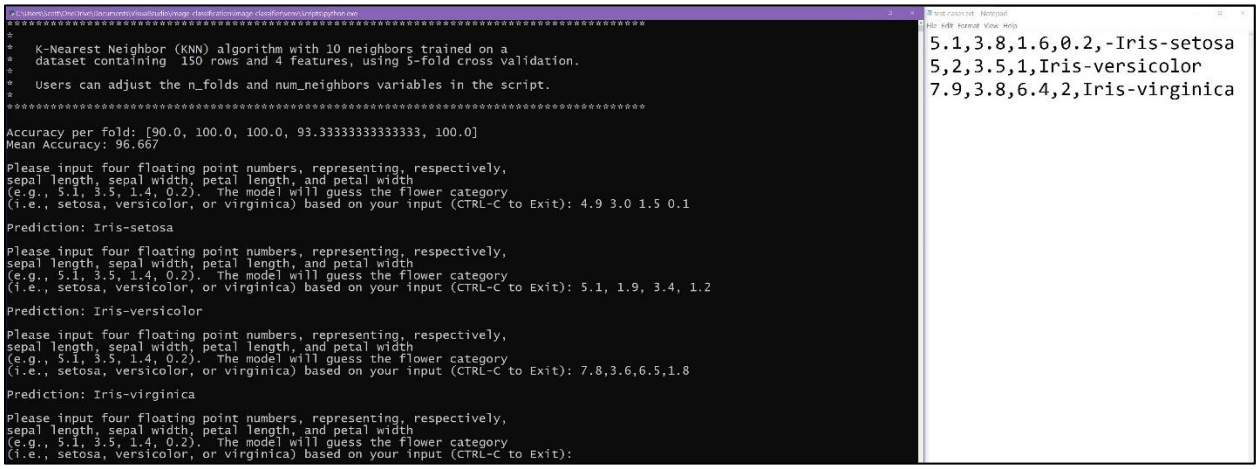
```
67      return dataset_split
68
69  # Calculate accuracy percentage
70  def accuracy_metric(actual, predicted):
71      correct = 0
72      for i in range(len(actual)):
73          if actual[i] == predicted[i]:
74              correct += 1
75      return correct / float(len(actual)) * 100.0
76
77  # Evaluate an algorithm using a cross validation split
78  def evaluate_algorithm(dataset, algorithm, n_folds, *args):
79      folds = cross_validation_split(dataset, n_folds)
80      scores = list()
81      for fold in folds:
82          train_set = list(folds)
83          # create hold out set
84          train_set.remove(fold)
85          #combine train sets
86          train_set = sum(train_set, [])
87          # create test set on new hold
88          test_set = list()
89          for row in fold:
90              row_copy = list(row)
91              test_set.append(row_copy)
92              # remove prediction from hold out set
93              row_copy[-1] = None
94          predicted = algorithm(train_set, test_set, *args)
95          actual = [row[-1] for row in fold]
96          accuracy = accuracy_metric(actual, predicted)
97          scores.append(accuracy)
98      return scores
99
100 # Calculate the Euclidean distance between two vectors
101 def euclidean_distance(row1, row2):
102     distance = 0.0
103     for i in range(len(row1) - 1):
104         distance += (row1[i] - row2[i]) ** 2
105     return sqrt(distance)
106
107 # Locate the most similar neighbors
108 def get_neighbors(train, test_row, num_neighbors):
109     distances = list()
110     for train_row in train:
111         dist = euclidean_distance(test_row, train_row)
112         distances.append((train_row, dist))
113     distances.sort(key=lambda tup: tup[1])
114     neighbors = list()
115     for i in range(num_neighbors):
116         neighbors.append(distances[i][0])
117     return neighbors
118
119 # Make a prediction with neighbors
120 def predict_classification(train, test_row, num_neighbors):
121     neighbors = get_neighbors(train, test_row, num_neighbors)
122     output_values = [row[-1] for row in neighbors]
123     prediction = max(set(output_values), key=output_values.count)
124     return prediction
125
126 # kNN Algorithm
127 def k_nearest_neighbors(train, test, num_neighbors):
128     predictions = list()
129     for row in test:
130         output = predict_classification(train, row, num_neighbors)
131         predictions.append(output)
132     return(predictions)
```

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



```
KNN.py

# 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

OPTION #1: KNN Classifier with Iris Data

This paper describes a *k*-Nearest Neighbors classifier (*k*-NN-C) constructed from scratch in Python that obtains a mean accuracy of 96.67% on the Iris Dataset using 5-fold cross-validation (CV), with the model's number of nearest neighbors set to 10. Fenner (2019) describes the Iris Dataset, sometimes referred to as Fisher's Iris Dataset, named after the mid-20th-century statistician, Sir Ronald Fisher, who used it in one of the first published academic papers on classification. The dataset contains three classes and 50 instances of each class. The classes correspond to the type of iris plant in the dataset: (a) Setosa, (b) Versicolor, and (c) Virginica. Four attributes in the dataset describe these plants: (a) sepal length in cm, (b) sepal width in cm, (c) petal length in cm, and (d) petal width in cm. After the program trains the *k*-NN-C model on the Iris Dataset, it prompts the user to input four floating-point numbers, separated by commas or spaces, which represent the features of new iris plants not found in the model's training data. The program then outputs the model's class prediction based on this user input.

k-Nearest Neighbor Classifier (*k*-NN-C) Overview

Fenner (2019) describes *k*-NN-C as one of the simpler machine learning models used to make predictions from labeled datasets. Briefly, *k*-NN-C models describe similarities between observation pairs, choose a specified number of the most similar examples, and combine these results to produce a single output prediction. The Euclidean distance is often used to describe similarities amongst features, though other metrics can be used, including the Minkowski and Hamming distances. The *k* in *k*-NN-C refers to the number of nearest neighbors on which the model bases its predictions. Common settings include 1, 3, 10, and 20, with the best setting often obtained through experimentation.

Fenner (2019) writes that *k*-NN-C models differ from other machine learning methods because they require all training data to make predictions on new test cases. For instance, removing any training observations will lead to the classifier making incorrect predictions because any of these training records may have been the nearest neighbor to a given test case. Such makes *k*-NN-C a lazy learner, meaning that it does not have a specialized training phase, which eager learners like logistic regression do, but instead stores all its training data to use in its prediction phase, subsequently making this phase more expensive 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 show the Python code to create the *k*-NN-C model. Much of this code is adapted from the example given by Brownlee (2019). The program begins by reading in the dataset, stored in the *data/iris.txt* file using the *load_csv()* function, as shown in lines 12 – 20 of Figure 1. Because reading in the file stores the dataset as a series of strings rather than floating-point numbers, the method *str_column_to_float()*, shown in lines 28 – 38 of Figure 1, converts the dataset features to decimal numbers. Similarly, in lines 23 – 35 of Figure 1, the method *str_column_to_int()* converts the string representation of each iris plant category to an integer so the *k*-NN-C model can better comprehend the data.

Cross-Validation Splits

Next, the program uses the *cross_validation_split()* method, shown in lines 56 – 67 of Figure 1, to divide the dataset into five CV splits. Barrow and Crone (2013) describe CV as a widely employed technique "to estimate the expected accuracy of a predictive algorithm by averaging predictive errors across mutually exclusive subsamples of … data" (p. 1). The function

divides the dataset into *k* mutually exclusive groups of approximately equal size, depending on the users' setting for *k.* One fold is retained for the validation dataset, while the remaining folds comprise the training data. The model is fitted on the training data and evaluated using the validation dataset (Brownlee, 2019).

Barrow and Crone (2013) write that the CV process is repeated *k* times, with each of the folds serving as the validation dataset exactly once. The program records the accuracy scores for each of the *k* evaluation sessions and obtains a final measure of the model's predictive accuracy by averaging these scores across the *k* folds. Lines 70 – 75 of Figure 2 show the function *accuracy_metric()*, which calculates the accuracy of the model on each fold using the following 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 the Euclidean Distance

As described, *k*-NN-C is a lazy learning algorithm and therefore does not have a specialized training phase but uses all its training data to generate predictions during the evaluation phase, which happens once the program constructs the CV splits. The method *evaluate_algorithm()*, shown in lines 78 – 98 of Figure 2, calls another method, *k_nearest_neighbors()*, which iterates over the test dataset, calling *predict_classification()* for each row of the test data. The *predict_classification()* method, in turn, calls the *get_neighbors()* method, which, in turn, calls the *euclidean_distance()* function, which calculates the Euclidean distance between each of the training examples and a given test example. The Euclidean distance is calculated using the following 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 each training and test example's sepal length, sepal width, petal length, and petal width. Lines 101 – 105 of Figure 2 show the *euclidean_distance()* function (Brownlee, 2019).

The program then sorts the training records in ascending order based on their distance from the test observation. The examples at the top of this sorted list correspond to the test cases' *k* nearest neighbors. Next, the model sums the class frequencies of these *k* nearest neighbors to generate a class prediction for a given test example. Whatever class occurs most frequently is what the model chooses as its prediction for the test case class. If there is a tie among the frequencies, the model chooses the class of the neighbor nearest the test example for its prediction. The program then compares the classifier's predictions with reality and calculates the accuracy for each test fold before averaging these accuracies over all the validation splits to produce the mean accuracy for the model.

Model Evaluation, User Input, and Future Research Recommendations

Figure 4 shows that 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. Then, the program allows the user to input four floating-point numbers that correspond to the sepal length, sepal width, petal length, and petal width, respectively, of iris plants previously unseen to the model. For example, after inputting features similar to, but not identical with, the training examples shown on the right-hand side of Figure 4, one can see the model predicts each of the iris plant categories correctly. Lines 134 – 177 of Figure 3 show the program's driver code. Users are required to input four valid numerical features separated by spaces, commas, or a combination of the two for the model to generate a valid prediction, or else the program notifies the user of their error and re-prompts the user for input. Finally, the program contains some additional functions,

such as the *normalize_dataset()* function, which can be used in future research to assess how normalizing the input features of the Iris Dataset affects the mean accuracy of the classifier (Brownlee, 2019).

Conclusion

In conclusion, this paper presented an overview of a *k*-Nearest Neighbor classifier (*k*-NN-C) built from scratch in Python that achieves 96.67% accuracy on the Iris Dataset using 5-fold cross-validation (CV), with the models' number of neighbors set to 10. The paper gave an overview of the Iris Dataset, *k*-NN-C models, and CV. The paper also described the classifier's inner workings, including how it calculates the Euclidean distance for each training example per test example, sorts these results, and sums the frequencies of the *k* nearest neighbor classes to produce predictions. Additionally, the program this paper described loaded the Iris Dataset, initialized the value of *k* to 10, iterated over all the training data points to obtain predictions for the test data by calculating the Euclidean distance between records, sorted the calculated distances in ascending order, obtained the top *k* rows from the sorted list, and made predictions for test cases based on the most frequently occurring classes in these top *k* rows. The program also accepted user input corresponding to iris plant features previously unseen by the model and generated predictions for this input 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