Algorithm Strategies used in k-Nearest Neighbors Classifier

Using Part-by-Part Explanations

Leonardo
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13517048@std.stei.itb.ac.id

***Abstract*—In real life cases, sometimes a single algorithm is used to solve many problems, and of course sometimes, many algorithms can be used to solve a single problem. For example, Greedy algorithm can be used to solve Fractional Knapsack Problem, Shortest Path, etc. But there are problems that can be solved by many algorithms. Problems like TSP can be solved with Brute Force, Divide and Conquer, Branch and Bound, and even Dynamic Programming. But what if a problem can't be solved with just a single algorithm? We need more than one algorithm and method to solve that one problem. In this paper, one of those kind of problems are going to be solved. It's one of machine learning classifiers, k-Nearest Neighbors. Algorithms needed to solve this problem will be explained part by part, be it Brute Force, Divide and Conquer, or something else.**

Keywords—Classification, k-Nearest Neighbors, Nearest Neighbor Search

I. Introduction

In this time of fourth industrial revolution, technologies are developing rapidly. Especially in the wide-and-deep cyber informatics world. The major itself provides the ability to automate things, the ability to precisely analyze flows, and so on. Because of the fast-progressive development, many majors are quickly renewed and updated. One of those major mentioned above is Data Science.

Data Science is a multi-discipline major utilizing Computer Science, Mathematics and Statistics, and Business knowledge to be able to get a better understanding of data. Usually data science workflow is very structured. It starts with Exploratory Data Analysis, followed by Data Preprocessing/Cleaning, followed by Feature Engineering, and ends with Machine Learning Modeling and evaluation. In the Machine Learning Modeling segment, there are two generalization of models, Unsupervised Learning, and Supervised Learning.

Unsupervised Learning algorithms are usually used in Clustering and Association problems. While Supervised Learning are being used in Regression and Classification problems. We will use Supervised Learning in this paper. People used many types of regression and classification models to predict needed test cases or data. In this case, we are focusing more in classification problems. There are many types of classifiers, and every classifier has their own algorithms and parameters used to learn and to predict an outcome from every specific test case.

K-Nearest Neighbors algorithm is one of the most used algorithm in Supervised Learning to do a classification and regression. It's easily put among the simplest Machine Learning algorithms. K-NN is very strong and simple to the point it's so easy to study and it's easy to understand what's working behind this classifier.

To understand fully about the algorithms and strategies used in k-NN, let's explore further into the paper.

II. Theory

In this section, theories used to make the classifier will be explained by sub-sections. What k-Nearest-Neighbors classifier is, what Nearest Neighbor Search is, and what algorithms build up this whole kNN classifier will be explained in detail.

A. K-Nearest Neighbors Classifier

K-Nearest Neighbor Classifier (in short k-NN, will be mentioned as this for the rest of the paper) is a Supervised-Machine Learning classifier which classifies based on its parameter's "distance" with other existing data. And no. k-NN is not a Neural Networking algorithm, the name can be misleading.

The train data are data consisting of features, and classes. Meanwhile the test datum consists of features without class, the class is a feature we are trying to predict. In this paper, we will only predict one test case.

The algorithm steps to implement k-NN is as follows:

1. Calculate distance between each element of training data and the test data.
2. Get the top *k* (amount of neighbors) data with shortest distances.
3. Classify the result based on the majority of classes from the top *k* neighbors.

The algorithmic notation is as follows:

```
function KNN(train_data, test : DataFrame,
k : integer)
DECLARATION
    neighbors : array[0..k-1] of data
    distances : array[0..k-1] of data
ALGORITHM
    train_data = load_data()
    { Calculate distances }
    distances = distanceOf(train_data,
test)
    sort_distances(distances)
    { Take k amount of Nearest Neighbors }
    i traversal [0..k]
        neighbors = neighbors +
distances[i]

    result = majorityOfClass(neighbors)
    ➔ result
```

Table 2.1: k-NN Pseudo-code

It seems so simple to look at, but the real implementation will be much more complex. For example, to find *k* nearest neighbors, we need another algorithm, and for the sake of tuning down the complexity, we need to smartly choose our algorithms to sort the list of distances and to find the majority of classes in neighbors.

Below is a picture of a k-NN example to better understand how this classifier works.

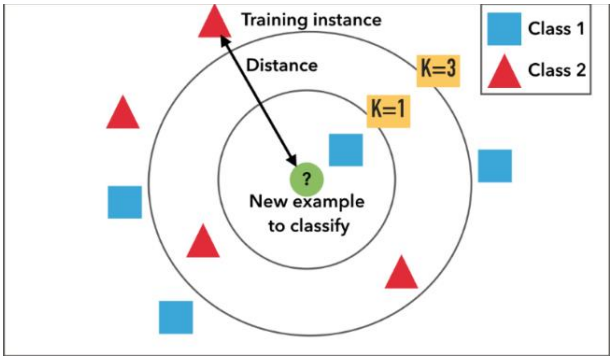

Figure 2.1: Example of k-NN classification (taken from: [1], accessed: April 26th 2019, at 00.31)

We will take Figure 2.1 as an example to find the class of the new example datum. We will use $k = 1$ and $k = 3$, so it makes 2 examples

1. $k = 1$: The nearest neighbor of the example is class1, and class1 is obviously the majority of the class. So the example class is class1. Thus the test is classified as class1

2. $k = 3$: The 3 nearest neighbors of the example is class1, class2, and class2. The majority of the class is class2. Thus the test is classified as class2.

As you can see, there can be different result returned just because we change the value of *k*. What if *k* is a number of multiplies of the total classes? There will be a case where all classes get exactly the same amount in the neighbors and there is no majority, so the classification will take the first index and much probably resulting in a false prediction. For example, in Figure 2.1, if we find 2 nearest neighbors, we will get exactly 1 class1, and 1 class2. There will be no majority of class. So it's highly recommended to not use a random number as *k*. It's highly commendable to use *k* that's relatively prime to the total number of classes.

As mentioned before, we need to find *k* nearest neighbors to make an assumption classifying a test case. But how do we find *k* amount of nearest neighbors? We used Nearest Neighbor Search algorithm to find them.

B. Nearest Neighbor Search

Nearest Neighbor Search (in short NNS, will be mentioned as this for the rest of the paper) is NOT a searching algorithm like Binary Search or Linear Search. It's an algorithm to find a minimum value on which the values are distances between a test case and the whole training data.

From the algorithm notation in the k-NN, we take this sub-program:

```
{ Calculate distances }
distances = distanceOf(train_data, test)
sort_distances(distances)
{ Take k amount of Nearest Neighbors }
i traversal [0..k]
    neighbors = neighbors + distances[i]
```

Table 2.2: NNS Pseudo-code

This part of k-NN is NNS. It returns exactly *k* amount of nearest neighbors.

How to find the neighbors? The naïve way to do this is to linearly calculate the distance and then find the *k* smallest distance-neighbors while iterating through the whole train data for the second time. In this paper, we will not use the naïve / brute force way to get the result.

As you can see in the algorithm notation above, we sort the distances ascendingly so we can just flexibly take the *k* first elements of distances to be our nearest neighbors.

Below is a picture of a test case pointing at all of its neighbors to find the distance between itself and the training data (d1 to d6):

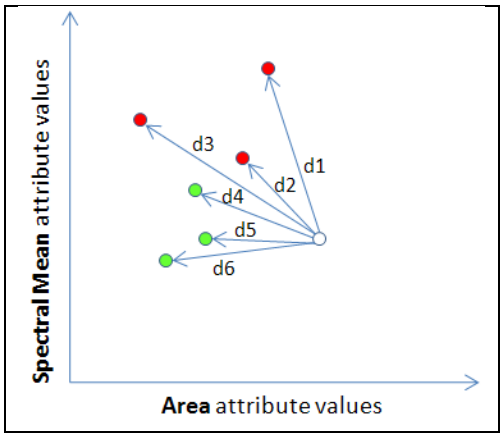

Figure 2.2: Distances between a test case (white) to its neighbors (green or red) (taken from: [2], accessed: April 26th 2019, at 00.45)

Despite its name, NNS is not only used in k-NN. It is frequently used in Computer Vision, Plagiarism Checker, Pattern Recognition, and many other else. For this paper, we use NNS to do Statistical Classification.

By now, we always mentioned the distance in a R^2 plane. But real life data does not only have 2 features, so it cannot be represented by a two-dimensional graph like Figure 2.2.

The distance always mentioned is actually the distance of each feature of the test and train data. So it's actually not as physical as we can imagine.

C. *Strategies used in kNN Classifier*

From the before-mentioned algorithms, we need the right strategies to be able to get the faster results. The usage of Divide and Conquer strategy really shines in this classifier. There are many usage of Divide and Conquer strategy in solving the sub-algorithms above. For ones forgetting, Divide and Conquer follows three main methods, Divide, Conquer, and Combine. The algorithm works like this: [3]

1. Divide the problem into many sub-problems recursively.
2. Dividing stops when the sub-problem is small enough.
3. Conquer the small-enough sub-problem.
4. Combine the result of the conquered sub-problem into one bigger sub-problem until it's the same as the original problem.
5. The combined result will be the final solution of the problem.

The first Divide and Conquer algorithm strategy is used after we get the distances of the neighbors. As we can see from the Figure 2.3, we need to sort the values ascending so the group of smallest distance-neighbors (the nearest neighbors) will be taken as the neighbors. When one mentions sorting, and fast, one is always mentioning either Quick Sort, or Merge Sort. Those are obviously two fastest sorting algorithms existing for now.

The second Divide and Conquer used in this classifier is in the majorityofClass function. How to find Majority and its count? It is found by the max-min search algorithm. Usually with linear search, we can find Maximum value with complexity of $O(n)$ from $T(n) = 2n - 2$. But with Divide and Conquer, we can get $O(n)$ but from a smaller $T(n)$. It's from $T(n) = \frac{3}{2}n - 2$.[3]

The rest of the program is run with Brute Force Algorithm Strategy. Because it's very understandable and we can still reach our goals. And Understandability is one of the goal of this paper.

III. Problem Solving Method

We will solve the problem not just in theories mentioned at section II, but also with experiment. The writer already wrote a Python3 code regarding this problem and already made a custom-made k-NN classifier to be analyzed and studied.

So the classifier will be divided into two big parts, and each part will be divided into many small parts that will be solved using the algorithm strategies mentioned in section II.

The first big part is the NNS part. In this part, the realization and implementation of the NNS algorithm in Table 2.2 will be shown and explained. This big part is fractionized into Distance Calculation, Sorting the distances, and Combining and Making NNS function.

The second big part is Finding the Majority of Class. In this part, it will be explained programmatically about one of the ways to find the majority / mode class in neighbors. This big part consists of the Maximum value finder, and Combining and Returning conclusion class of the classifier.

IV. Implementation and Analysis

Besides having to be fast, the program made must also be correct. It should implement the algorithms mentioned above and it should run without an error and producing the right result. To know the right result, we will compare our classification results with Python library sklearn's own KNeighborsClassifier. Meanwhile we also print the time taken to produce the result and again we compare it with sklearn's KNeighborsClassifier.

We are using Jupyter Notebook to make, edit, compile, and run our code. See this for further details: https://jupyter.org/.

Before entering the classifier, we will take the data from a CSV file using Python library pandas.

```
In [2]:
data = pd.read_csv('train.csv')
data.head()
```

Out[2]:

	id	rank_team1	total_points_team1	rank_team2	total_points_team2	is_team1_won
0	0	38.0	0.00	30.0	0.00	0
1	1	11.0	0.00	18.0	0.00	1
2	4	71.0	0.00	123.0	0.00	1
3	5	118.0	0.00	3.0	0.00	0
4	6	30.0	722.65	56.0	555.61	1

Figure 4.1: Take train data from csv(Jupyter Notebook: LeoW_KNN.ipynb, accessed: April 26th 2019, at 03.26)

```
In [3]:
test = pd.read_csv('test.csv')
test.head()
```

Out[3]:

	id	rank_team1	total_points_team1	rank_team2	total_points_team2
0	2	90.0	386.2	49.0	629.6
1	3	103.0	0.0	18.0	0.0
2	7	20.0	0.0	7.0	0.0
3	9	5.0	0.0	67.0	0.0
4	11	31.0	0.0	16.0	0.0

Figure 4.2: Take test data from csv(Jupyter Notebook: LeoW_KNN.ipynb, accessed: April 26th 2019, at 03.27)

As we can see above, test data doesn't contain feature 'is_team1_won', as in fact we are going to predict one of the test case from test dataset.

But before predicting, to be precise, we will do a little Feature Engineering and convert 'is_team1_won' to a string attribute 'result', if team1 won, then 'result' will be 'win', else the 'result' will be 'lose'.

```
In [6]:
data = data.drop('is_team1_won', axis = 1)
data.head()
```

Out[6]:

	rank_team1	total_points_team1	rank_team2	total_points_team2	result
0	38.0	0.00	30.0	0.00	lose
1	11.0	0.00	18.0	0.00	win
2	71.0	0.00	123.0	0.00	win
3	118.0	0.00	3.0	0.00	lose
4	30.0	722.65	56.0	555.61	win

Figure 4.3: Converting is_team1_won to 'result' (Jupyter Notebook: LeoW_KNN.ipynb, accessed: April 26th 2019, at 03.33)

At this point, the train data is ready to be injected to our KNN classifier. Following the parts division from section III, let's start making our classifier.

1. NNS

   Here we save not only the distance of each neighbor, we also need to save the index of the neighbor in train data for future use. It is implemented by making a list of list.

   a. Distance Calculation

      In this algorithm of finding distance, we will use Euclidean Distance algorithm as it is simple to be implemented and it produces a nice exact result for our NNS algorithm.

```
In [39]:
def euclideanDistance(train, test):
    temp_dist = 0
    for x in train.columns:
        #calculate from every feature
        temp_dist += (train[x] - test[x])**2
    return np.sqrt(temp_dist)
```

Figure 4.4: Euclidean Distance (Jupyter Notebook: LeoW_KNN.ipynb, accessed: April 26th 2019, at 03.39)

This Euclidean distance function will be called for each rows of the train data.

b. Sorting the distances

For the sorting algorithm, the writer used Merge Sort. And here we sort based on the distances,

```
In [40]:
def mergesort(list_of_list):
    if (len(list_of_list) > 1):
        middle = len(list_of_list)//2
        #divide
        left = list_of_list[middle:]
        right = list_of_list[:middle]
        #conquer
        mergesort(left)
        mergesort(right)
        #combine
        i = 0; j = 0; k = 0
        while ((i < len(left)) and (j < len(right))):
            if (left[i][1] < right[j][1]):
                list_of_list[k] = left[i]
                i += 1
            else:
                list_of_list[k] = right[j]
                j += 1
            k += 1
        #check for leftover elements
        while (i < len(left)):
            list_of_list[k] = left[i]
            k += 1
            i += 1

        while (j < len(right)):
            list_of_list[k] = right[j]
            k += 1
            j += 1
    #else 1 element, do nothing (conquering)
```

Figure 4.5: MergeSort implementation (Jupyter Notebook: LeoW_KNN.ipynb, accessed: April 26th 2019, at 03.47)

c. Combine and make NNS function

We now make the full NNS function that calls the Euclidean Distance function and Merge Sort procedure.

```
In [41]:
def NNS(train, test, k):
    #Nearest Neighbors Search
    distances = []
    for i in range(len(train)):
        temp_dist = euclideanDistance(test, train.iloc[i])
        distances.append([i, temp_dist[index]])

    mergesort(distances)#merge sort
    #take n_neighbors amount of sorted distances
    neighbors = []
    for j in range(k):
        neighbors.append(distances[j])

    #saves it's index in train data and the distance
    return neighbors
```

Figure 4.6: NNS final implementation (Jupyter Notebook: LeoW_KNN.ipynb, accessed: April 26th 2019, at 03.58)

Note that there is a variable "index". It's a global variable declaring what the index is of the test data is going to be predicted.

2. Find Majority of Class

In this part of the program, we're just simply finding the majority of the class from the neighbors. It's implemented by making a Python's dictionary and putting class names besides it's counts.

The counting algorithm is implemented right below the NNS being called in KNN function. The counting algorithm will be shown in the whole KNN function in the Combining and Returning conclusion class (2b).

a. Maximum value finder

As mentioned in section III, the finding maximum value from the dictionary is implemented with Divide and Conquer strategy.

```
In [42]:
def findMaxfromDict(dictionary, start, end):
    #divide and conquer implementation to find
    #maximum value out of dictionary values
    dict_list = list(dictionary.keys())
    if ((end - start) <= 1):
        return dict_list[start]
    else:
        middle = (end + start)//2
        left = findMaxfromDict(dictionary, start, middle)
        right = findMaxfromDict(dictionary, middle, end)
        if (dictionary[left] > dictionary[right]):
            return left
        else:
            return right
```

Figure 4.7: findMaxfromDict implementation (Jupyter Notebook: LeoW_KNN.ipynb, accessed: April 26th 2019, at 04.16)

b. Combining and Returning conclusion class

Now that every function and procedure needed to perform a classification with k-NN, we now make the k-NN function that calls all of the above methods.

```
In [43]:
def KNN(train, test, k):
    #Nearest Neighbors Search
    neighbors = NNS(train, test, k)

    #take conclusion from the most result
    dict_result = {}
    for i in range(len(neighbors)):
        temp_res = train.iloc[neighbors[i][0]]['result']
        if (temp_res in dict_result):
            dict_result[temp_res] += 1
        else:
            dict_result[temp_res] = 1

    #divide and conquer max finding
    conclusion = findMaxfromDict(dict_result, 0, len(list(dict_result.keys())))
    returned_neighbors = [i[0] for i in neighbors]
    return conclusion, returned_neighbors
```

Figure 4.8: Implementation of our custom KNN Classifier (Jupyter Notebook: LeoW_KNN.ipynb, accessed: April 26th 2019, at 04.23)

Note that the function returns two values. The conclusion class from the classification, and the index of neighbors involved in the prediction.

Now that we have our own KNN Classifier, let's test it with the third row of the test data. Save it to new DataFrame, test_df, that will be the second parameter of the KNN function and check its shape.

```
In [44]:
index = 3

In [45]:
test_df = pd.DataFrame([test.iloc[index]])
test_df.shape

Out[45]: (1, 4)
```

Figure 4.9: Declaring test_df (Jupyter Notebook: LeoW_KNN.ipynb, accessed: April 26th 2019, at 04.29)

Note that the shape is (1, 4). This means that it consists of 1 row and 4 columns because it's one of the test cases from test dataset. Now let's put it to our k-NN classifier. We will set the *k* to 11.

```
In [46]:
temp_k = 11
KNN(data, test_df, temp_k)

Out[46]: ('win', [8219, 6434, 1722, 4238, 1428, 5116, 1004, 1005, 2776, 7529, 7590])
```

Figure 4.10: Testing our k-NN Classiffier (Jupyter Notebook: LeoW_KNN.ipynb, accessed: April 26th 2019, at 04.34)

The output means that it predicts a win, and prints *k* indexes of the nearest neighbors contributing in giving us the conclusion.

Now let's compare it to a real-sklearn KNeighborsClassifier. This time let's use the eighth row of test dataset. Below it is shown the libraries imported to make our comparison. We imported time library as well to compare our custom-made k-NN duration needed versus that of the sklearn KNeighborsClassifier. We will have three comparisons. Comparison with $k = 1$, $k = 3$, and $k = 5$.

1. $k = 1$

```
In [63]:
#When k = 1
k = 1
a = time.time()
result, neighbors = KNN(data, test_df, k)
b = time.time()
print(result)
print(neighbors)#indexes of neighbors
print("used time =", b-a)

lose
[9784]
used time = 21.95661234855652

In [70]:
_k = 1

In [71]:
md = KNeighborsClassifier(n_neighbors = _k)
a = time.time()
md.fit(data.iloc[:,0:4], data['result'])
b = time.time()

# Predicted class
print(md.predict(test_df))

# k nearest neighbors
print(md.kneighbors(test_df)[1])

print("used time =", b-a)

['lose']
[[9784]]
used time = 0.18896889686584473
```

Figure 4.11: Comparison $k = 1$ (Jupyter Notebook: LeoW_KNN.ipynb, accessed: April 26th 2019, at 04.57)

2. $k = 3$

```
In [64]:
#When k = 3
k = 3
a = time.time()
result, neighbors = myknn(data, test_df, k)
b = time.time()
print(result)
print(neighbors)
print("used time =", b-a)

lose
[9784, 10656, 3400]
used time = 20.641812562942505
```

```
In [72]:
_k = 3
```

```
In [73]:
md = KNeighborsClassifier(n_neighbors = _k)
a = time.time()
md.fit(data.iloc[:,0:4], data['result'])
b = time.time()

# Predicted class
print(md.predict(test_df))

# k nearest neighbors
print(md.kneighbors(test_df)[1])

print("used time =", b-a)

['lose']
[[ 9784 10656  3400]]
used time = 0.1239769458770752
```

Figure 4.12: Comparison $k = 3$ (Jupyter Notebook: LeoW_KNN.ipynb, accessed: April 26th 2019, at 05.00)

3. $k = 5$

```
In [65]:
#When k = 5
k = 5
a = time.time()
result, neighbors = KNN(data, test_df, k)
b = time.time()
print(result)
print(neighbors)
print("used time =", b-a)

lose
[9784, 10656, 3400, 717, 5025]
used time = 21.132737159729004
```

```
In [74]:
_k = 5
```

```
In [75]:
md = KNeighborsClassifier(n_neighbors = _k)
a = time.time()
md.fit(data.iloc[:,0:4], data['result'])
b = time.time()

# Predicted class
print(md.predict(test_df))

# k nearest neighbors
print(md.kneighbors(test_df)[1])

print("used time =", b-a)

['lose']
[[ 9784 10656  3400   717  5025]]
used time = 0.11399435997009277
```

Figure 4.13: Comparison $k = 5$ (Jupyter Notebook: LeoW_KNN.ipynb, accessed: April 26th 2019, at 05.04)

V. Conclusion

K-NN Classifier is one of the most used Machine Learning classifiers existing in programming world. It's easy to use, but it got the strength to compete with high-end classifier models.

By now we managed to get a grip of understanding of the classifier and also implementing a custom-made one. Despite having exactly correct result, our custom-made k-NN performs much slowly in comparison to sklearn's KNeighborsClassifier,

The evaluation is probably the distance calculator is too slow because the writer is using Euclidean distance on so many data. It can be changed to another metric like Chebyshev or Manhattan. The other evaluation is probably not using brute force strategy as the primary workflow because the time complexity is so big that time consumed in our classifier is almost 200% more than those of sklearn's. Finally is to make use of Python's primitive functions. Although Max and Sort have been implemented with the fastest algorithm possible, usually the primitive *max* function and *sorted* function still performs much faster.

The writer hopes that by reading this paper, readers will be able to expand his/her knowledge towards Artificial Intelligence, Machine Learning, and the algorithm behind it. The writer also hopes that the implementation example can be much of a help for new starters to experiment, make, and further evaluate their first own custom-made classifier. Hopefully this paper can make the readers explore more about Machine Learning, and make use of it in Data Science, in Engineering, to solve world problems.

Figures and Tables

ACKNOWLEDGMENT *(Heading 5)*

First of all, I would thank my family for their prayer and emotional and material support for all these years of my life, especially in the process of making this paper. I would also thank my friends for giving me mental support when I need it. Lastly I would like to thank Dr. Ir. Rinaldi Munir, MT. as my lecturer who teaches me in IF2211 Algorithm Strategies course and for this assignment that teaches us, your students, our professional and informatics skills by conducting a research on algorithms. I would also like to thank my references for giving me such knowledge and inspiration to make this paper.

REFERENCES

[1] https://blog.usejournal.com/a-quick-introduction-to-k-nearest-neighbors-algorithm-62214cea29c7, accessed: April 25th 2019, at 23.57

[2] https://www.harrisgeospatial.com/docs/BackgroundKNN.html, accessed: April 26th 2019, at 00.43

[3] http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2017-2018/Algoritma-Divide-and-Conquer-(2018).pdf, accessed: April 26th 2019, at 02.32

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 26 April 2019

Leonardo
13517048