k-nearest-neighbor

June 5, 2023

1 KNN - K-Nearest Neighbors Notes

1.1 What is K-Nearest Neighbors?

- K-Nearest Neighbors (KNN) is a supervised learning algorithm used for classification and regression tasks.
- KNN is a distance-based classifier that assumes smaller distances between points indicate more similarity.
- Each column in a dataset acts as a dimension, making it easy to visualize with X and Y coordinates.
- KNN requires labels for each point in the dataset to make predictions.

1.2 The algorithm works as follows:

1. Choose a point
2. Find the K-nearest points
    1. K is a predefined user constant such as 1, 3, 5, or 11
3. Predict a label for the current point:
    1. Classification - Take the most common class of the k neighbors
    2. Regression - Take the average target metric of the k neighbors
    3. Both classification or regression can also be modified to use weighted averages based on the distance of the neighbors

1.3 Fitting the model

- KNN is a classifier that works differently from others.
- It doesn't do much during the "fit" step.
- KNN just stores training data and labels.
- No distances are calculated during the "fit" step.
- All the work is done during the "predict" step.

1.4 Making predictions with K

- KNN algorithm predicts a class for a point during the "predict" step.
- It calculates distances between the point and every point in the training set.
- K closest points (neighbors) are found and their labels are examined.
- Each of the K-closest points gets to 'vote' about the predicted class.
- The majority wins and the algorithm predicts the point as whichever class has the highest count among all of the k-nearest neighbors.

1.5 Distance metrics

- Choosing the right distance metric is crucial when using the KNN algorithm.
- The distance metric significantly affects the algorithm's output.
- Euclidean distance and Minkowski distance are the standard distance metrics to consider.

1.6 Evaluating model performance

- How to evaluate model performance depends on whether it's being used for classification or regression tasks
- KNN can be used for regression and binary/multicategorical classification tasks
- Evaluating classification performance for KNN is similar to any other classification algorithm
- You need a set of predictions and corresponding ground-truth labels to compute evaluation metrics such as Precision, Recall, Accuracy, F1-Score, etc.

1.6.1 K-means

- K-means algorithm is unsupervised learning clustering algorithm related to KNN.
- K represents the number of clusters in K-means, not the number of neighbors.
- Unlike KNN, K-means is an iterative algorithm that repeats until convergence.
- K-means groups data points together using a distance metric to create homogeneous groupings.

2 More On Distance Metrics:

- The K-Nearest Neighbors (KNN) algorithm is a foundational Supervised Learning algorithm.
- Distance metrics are used to determine how similar two objects are in KNN.
- Distance helps quantify similarity between objects.
- Each column in a dataset is treated as a separate dimension in KNN.
- There are multiple distance metrics available to calculate the distance between data points.
- Learning different distance metrics is important to evaluate how similar or different data points are in KNN.

2.1 Manhattan distance

- Manhattan distance is a distance metric that measures the distance between two points traveling along the axes of a grid.
- It calculates the number of units moved in the X and Y dimensions, which is the same for the red, blue, and yellow lines in the image.
- Manhattan distance can be remembered by thinking of the famous grid of streets in Manhattan.
- It can be calculated in any n-dimensional space by taking into account the number of units moved in each dimension and summing them.

Here's the formula for Manhattan distance:

$$d(x, y) = \sum_{i=1}^{n} |x_i - y_i|$$

Let's break this formula down:

- The left side of the equals sign measures the distance between two points.
- The right side of the equals sign calculates the absolute number of units moved in each dimension and adds them up.
- The $\sum$ means the cumulative sum of each step in the calculation.
- To calculate distance on a grid, movements in the opposite direction must count, so the absolute difference between them is calculated.
- Code can easily calculate the distance between two points stored as tuples using a `for` loop.

[1]:
```
# Locations of two points A and B
A = (2, 3, 5)
B = (1, -1, 3)

manhattan_distance = 0

# Use a for loop to iterate over each element
for i in range(3):
    # Calculate the absolute difference and add it
    manhattan_distance += abs(A[i] - B[i])

manhattan_distance
```

[1]: 7

2.1.1 A hint on turning mathematical notation into code

- $\sum$ symbol in mathematical notation can be represented as a `for` loop.
- The math on the right of the $\sum$ symbol tells you what the body of the `for` loop should look like.
- The numbers on the bottom and top of the $\sum$ sign tell you the starting and stopping indexes.
- n in the Manhattan distance equation means "length n", the length of the entire number of dimensions.
- Be careful interpreting the starting dimensions, as computer scientists start counting at 0 while mathematicians start at 1.

2.2 Euclidean distance

- The Euclidean distance is the most common distance metric.
- The Pythagorean theorem is at the heart of this metric.
- The green line measures the Euclidean distance between two points by moving in a straight line.
- The length of the green line can be calculated using the Pythagorean theorem.
- The Euclidean distance between two points in the diagram above is approximately 8.485.

2.2.1 Working with more than two dimensions

- You can generalize the Euclidean distance equation to any number of dimensions.
- The formula for the Euclidean distance in a 3-dimensional space is: $d^2 = a^2 + b^2 + c^2$.
- The Euclidean distance equation is straightforward - for each dimension, subtract one point's value from the other's, square it, and add it to the running total.

$$d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

In Python, you can easily calculate Euclidean distance as follows:

[2]:
```
from math import sqrt

# Locations of two points A and B
A = (2, 3, 5)
B = (1, -1, 3)

euclidean_distance = 0

# Use a for loop to iterate over each element
for i in range(3):
    # Calculate the difference, square, and add it
    euclidean_distance += (A[i] - B[i]) ** 2

# Square root of the final result
euclidean_distance = sqrt(euclidean_distance)

euclidean_distance
```

[2]:
```
4.58257569495584
```

- Minkowski distance is a generalized distance metric across a Normed Vector Space
- A Normed Vector Space is a collection of space where each point has been run through a function
- Every vector must have a positive length and the zero vector outputs a length of 0
- Manhattan and Euclidean distances are special cases of Minkowski distance
- The function in Minkowski distance is just an exponent.

If you were to define a value for the exponent, you could say that:

```
# Manhattan Distance is the sum of all side lengths to the first power
manhattan_distance = np.power((length_side_1**1 + length_side_2**1 + ... length_side_n**1), 1/1)

# Euclidean Distance is the square root of the sum of all side lengths to the second power
euclidean_distance = np.power((length_side_1**2 + length_side_2**2 + ... length_side_n**2), 1/2)

# Minkowski Distance with a value of 3 would be the cube root of the sum of all side lengths to
minkowski_distance_3 = np.power((length_side_1**3 + length_side_2**3 + ... length_side_n**3), 1/

# Minkowski Distance with a value of 5
minkowski_distance_5 = np.power((length_side_1**5 + length_side_2**5 + ... length_side_n**5), 1/
```

> **NOTE**: You'll often see Minkowski distance used as a parameter for any distance-based machine learning algorithms inside `sklearn`.

3 Generatlized Minkowski distance function

Formula for Minkowski distance:

$$d(x, y) = \left(\sum_{i=1}^{n} |x_i - y_i|^c \right)^{\frac{1}{c}}$$

- Minkowski distance is a formula used to calculate distance between two points.
- Manhattan distance is a special case of Minkowski distance where c=1.
- Euclidean distance is a special case of Minkowski distance where c=2.

[3]:
```
import numpy as np

# minkowski distance function that takes 4 arguments: two arrays, the norm to␣
 ↪calculate, and verbose (default True)
def distance(x1, x2, c=2, verbose=True):
    # ensure numpy arrays
    x1 = np.array(x1)
    x2 = np.array(x2)

    # calculate distance
    distance = (sum(abs(x1 - x2)**c))**(1/c)

    # print verbose
    if verbose:
        print(f"Distance between {x1} and {x2} is {distance:.2f}")

    return distance

test_point_1 = (1, 2)
test_point_2 = (4, 6)
print(distance(test_point_1, test_point_2)) # Expected Output: 5.0
print(distance(test_point_1, test_point_2, c=1)) # Expected Output: 7.0
print(distance(test_point_1, test_point_2, c=3)) # Expected Output: 4.
 ↪497941445275415
```

```
Distance between [1 2] and [4 6] is 5.00
5.0
Distance between [1 2] and [4 6] is 7.00
7.0
Distance between [1 2] and [4 6] is 4.50
4.497941445275415
```

4 Finding The Best Value Of K

4.1 Finding the optimal number of neighbors

- The K-Nearest Neighbors algorithm requires selecting a value for K

- There is no one best value for K
- Strategies can be used to select a good or near optimal value for K

4.2 K, overfitting, and underfitting

- A smaller value of K results in a tighter fit of the model in supervised learning.
- Overfitting can occur if the model pays too much attention to every detail and creates a complex decision boundary.
- Conversely, underfitting occurs if the model is too simplistic.
- A visual explanation can help understand this concept.
- It's important to find the best value for K by iterating over multiple values and comparing performance at each step.

As you can see from the image above, k=1 and k=3 will provide different results!

4.3 Iterating over values of K

- Use odd values for k in KNN to avoid ties and guesswork
- Fit a KNN classifier for each value of K within a minimum and maximum boundary
- Generate predictions and evaluate performance metrics for each model
- Compare results and choose the model with the lowest overall error or highest overall score
- Plot the error for each value of K to find the value where the error is lowest.

4.4 KNN and the curse of dimensionality

- KNN is not ideal for large datasets or models with high dimensionality.
- The time complexity of KNN is exponential, meaning it takes a lot of operations to complete.
- For smaller datasets, KNN can work well due to its simplicity.
- However, for datasets with millions of rows and thousands of columns, another algorithm may be a better choice as KNN could take years to complete.

5 KNN From Scratch

To keep things simple, use a helper function, `euclidean()`, from the `spatial.distance` module of the `scipy` library.

[4]:
```
from scipy.spatial.distance import euclidean
import numpy as np
```

5.1 Create the `KNN` class

[5]:
```
# Define the KNN class with two empty methods - fit and predict
class KNN:
    def fit(self, X_train, y_train):
        return

    def predict(self, X_test):
        return
```

```
    def closest(self, row):
        return
```

5.2 Complete the `fit()` method

- When fitting a KNN classifier, you're just storing points and their labels
- There's no actual fitting involved, just data storage
- The stored data is used to calculate nearest neighbors when predicting

The inputs for this function are:

- `self`: since this will be an instance method inside the `KNN` class
- `X_train`: an array, each row represents a *vector* for a given point in space

- `y_train`: the corresponding labels for each vector in `X_train`. The label at `y_train[0]` is the label that corresponds to the vector at `X_train[0]`, and so on

[6]:
```
def fit(self, X_train, y_train):
    self.X_train = X_train
    self.y_train = y_train

# This line updates the knn.fit method to point to the function you've just␣
 ↪written
KNN.fit = fit
```

5.2.1 Helper functions

Three helper functions.

`_get_distances()` function.

- Take in two arguments: `self` and `x`
- Create an empty array, `distances`, to hold all the distances you're going to calculate
- Enumerate through every item in `self.X_train`. For each item:
  - Use the `euclidean()` function to get the distance between x and the current point from `X_train`
  - Create a tuple containing the index and the distance (in that order!) and append it to the `distances` array
- Return the `distances` array when a distance has been generated for all items in `self.X_train`

[7]:
```
def _get_distances(self, x):
    distances = []
    for ind, val in enumerate(self.X_train):
        dist_to_i = euclidean(x, val)
        distances.append((ind, dist_to_i))
    return distances

# This line attaches the function you just created as a method to KNN class
KNN._get_distances = _get_distances
```

`_get_k_nearest()` function

- Take three arguments:
  - `self`
  - `dists`: an array of tuples containing (index, distance), which will be output from the `_get_distances()` method.
  - `k`: the number of nearest neighbors you want to return
- Sort the `dists` array by distances values, which are the second element in each tuple
- Return the first `k` tuples from the sorted array

[8]:
```
def _get_k_nearest(self, dists, k):
    sorted_dists = sorted(dists, key=lambda x: x[1])
    return sorted_dists[:k]


# This line attaches the function you just created as a method to KNN class
KNN._get_k_nearest = _get_k_nearest
```

`_get_label_prediction()` function

- Create a list containing the labels from `self.y_train` for each index in `k_nearest` (remember, each item in `k_nearest` is a tuple, and the index is stored as the first item in each tuple)
- Get the total counts for each label (use `np.bincount()` and pass in the label array created in the previous step)
- Get the index of the label with the highest overall count in counts (use `np.argmax()` for this, and pass in the counts created in the previous step)

[9]:
```
def _get_label_prediction(self, k_nearest):

    labels = [self.y_train[i] for i, _ in k_nearest]
    counts = np.bincount(labels)
    return np.argmax(counts)

# This line attaches the function you just created as a method to KNN class
KNN._get_label_prediction = _get_label_prediction
```

Can now complete the predict method.

5.3 Complete the `predict()` method

This method does all the heavy lifting for KNN, so this will be a bit more complex than the `fit()` method.

- In addition to `self`, our `predict` function should take in two arguments:
  - `X_test`: the points we want to classify
  - `k`: which specifies the number of neighbors we should use to make the classification. Set `k=3` as a default, but allow the user to update it if they choose
- For each item:
  - Calculate the distance to all points in `X_train` by using the `._get_distances()` helper method

  - Find the k-nearest points in `X_train` by using the `._get_k_nearest()` method
  - Use the index values contained within the tuples returned by `._get_k_nearest()` method to get the corresponding labels for each of the nearest points

  - Determine which class is most represented in these labels and treat that as the prediction for this point. Append the prediction to `preds`
- Once a prediction has been generated for every item in `X_test`, return `preds`

[10]:
```
def predict(self, X_test, k=3):
    preds = []
    # Iterate through each item in X_test
    for i in X_test:
        # Get distances between i and each item in X_train
        dists = self._get_distances(i)
        k_nearest = self._get_k_nearest(dists, k)
        predicted_label = self._get_label_prediction(k_nearest)
        preds.append(predicted_label)
    return preds

# This line updates the knn.predict method to point to the function you've just␣
 ↪written
KNN.predict = predict
```

5.4 Test the KNN classifier

Note that there are ***3 classes*** in the Iris dataset, making this a multi-categorical classification problem. This means that you can't use evaluation metrics that are meant for binary classification problems. For this, just stick to accuracy for now.

[11]:
```
# Import the necessary functions
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

iris = load_iris()
data = iris.data
target = iris.target
```

[12]:
```
X_train, X_test, y_train, y_test = train_test_split(data, target, test_size=0.
 ↪25, random_state=0)
```

[13]:
```
# Instantiate and fit KNN
knn = KNN()
knn.fit(X_train, y_train)
```

[14]:
```
# Generate predictions
preds = knn.predict(X_test)
```

[15]:
```
print("Testing Accuracy: {}".format(accuracy_score(y_test, preds)))
# Expected Output: Testing Accuracy: 0.9736842105263158
```

```
Testing Accuracy: 0.9736842105263158
```

6 KNN With Scikit Learn

6.1 Why use scikit-learn?

- Implementing the KNN algorithm is a valuable experience but professional toolsets like scikit-learn are recommended.
- Scikit-learn has backend optimizations that make the algorithm faster and more efficient.
- Professional toolsets will have best-in-class implementations that a single developer or data scientist cannot rival.
- Scikit-learn's KNN implementation is more robust and fast due to clever optimizations like caching distances.

6.2 Read the sklearn docs

- Familiarize yourself with documentation for libraries and frameworks you use.
- scikit-learn provides high-quality documentation for algorithms.
- General documentation pages provide inputs, parameters, outputs, and caveats of any algorithm.
- User Guides explain how the algorithm works and how to best use it, complete with sample code.
- The scikit-learn user guide for K-Nearest Neighbors includes an image and explanation of how different parameters affect model performance.

Documentation Page

User Guide

6.3 Best practices

- Scikit-learn has built-in functions for evaluating models with precision, accuracy, or recall scores.
- Focus on practical questions when completing the lab, such as decisions regarding data and predictors.
- Determine the optimal parameter values for your model and choose appropriate metrics for evaluation.
- Assess whether there is room for improvement with your model and if potential gains are worth the time needed to achieve them.

7 Functions For Improving KNN Performance - find best k

7.1 Function using f1 score

[2]:
```
def find_best_k(X_train, y_train, X_test, y_test, min_k=1, max_k=25):
    best_k = 0
    best_score = 0.0
    for k in range(min_k, max_k+1, 2):
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X_train, y_train)
        preds = knn.predict(X_test)
        f1 = f1_score(y_test, preds)
        if f1 > best_score:
            best_k = k
            best_score = f1

    print("Best Value for k: {}".format(best_k))
    print("F1-Score: {}".format(best_score))
    return best_k
```

7.2 Function using log loss

[1]:
```
# modify find_best_k to find the best value for k using log loss instead of␣
↪f1-score
def find_best_k(X_train, y_train, X_test, y_test, min_k=1, max_k=50):
    best_k = 0
    best_score = 0.125
    for k in range(min_k, max_k+1, 2):
        knn = KNeighborsClassifier(n_neighbors=k)
        knn.fit(X_train, y_train)
        preds = knn.predict_proba(X_test)
        log_loss = -cross_val_score(knn, X_train, y_train,␣
↪scoring="neg_log_loss").mean()
        if log_loss < best_score:
            best_k = k
            best_score = log_loss

    print("Best Value for k: {}".format(best_k))
    print("Log Loss: {}".format(best_score))
    return best_k
```