


 [learn-co-curriculum](#) / [dsc-k-nearest-neighbors-lab](#) Public [View license](#) 0 stars  161 forks Star Watch ▾

<> Code

 Issues Pull requests Actions Projects Security Insights solution ▾

...

This branch is [6 commits ahead](#), [6 commits behind](#) master.

sumedh10 update readme ...

on Nov 13, 2019

 7[View code](#) README.md

K-Nearest Neighbors - Lab

Introduction

In this lesson, you'll build a simple version of a *K-Nearest Neighbors classifier* from scratch, and train it to make predictions on a dataset!

Objectives

In this lab you will:

- Implement a basic KNN algorithm from scratch

Getting Started

You'll begin this lab by creating a classifier. To keep things simple, you'll be using a helper function, `euclidean()`, from the `spatial.distance` module of the `scipy` library. Import this function in the cell below:

```
from scipy.spatial.distance import euclidean
import numpy as np
```

Create the KNN class

You will now:

- Create an class called `KNN`
- This class should contain two empty methods -- `fit` and `predict`

```
# Define the KNN class with two empty methods - fit and predict
class KNN:

    def fit():
        pass

    def predict():
        pass
```

Comple the `fit()` method

Recall that when "fitting" a KNN classifier, all you're really doing is storing the points and their corresponding labels. There's no actual "fitting" involved here, since all you need to do is store the data so that you can use it to calculate the nearest neighbors when the `predict()` method is called.

The inputs for this function should be:

- `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

In the cell below, complete the `fit` method:

```
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
```

Helper functions

Next, you will write three helper functions to make things easier when completing the `predict()` method.

In the cell below, complete the `_get_distances()` function. This function should:

- 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`

```
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
```

Well done! You will now create a `_get_k_nearest()` function to retrieve indices of the `k`-nearest points. This function should:

- 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

Hint: To easily sort on the second item in the tuples contained within the `dists` array, use the `sorted()` function and pass in `lambda` for the `key=` parameter. To sort on the second element of each tuple, you can just use `key=lambda x: x[1]` !

```
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
```

The final helper function you'll create will get the labels that correspond to each of the `k`-nearest point, and return the class that occurs the most.

Complete the `_get_label_prediction()` function in the cell below. This function should:

- 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)

```
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
```

Great! Now, you now have all the ingredients needed to complete the `predict()` method.

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. Here's an outline of how this method should work:

- 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
- Your method will need to iterate through every item in `X_test`. 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`

Follow these instructions to complete the `predict()` method in the cell below:

```
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 writ
KNN.predict = predict
```

Great! Now, try out your new KNN classifier on a sample dataset to see how well it works!

Test the KNN classifier

In order to test the performance of your model, import the *Iris dataset*. Specifically:

- Use the `load_iris()` function, which can be found inside of the `sklearn.datasets` module. Then call this function, and use the object it returns
- Import `train_test_split()` from `sklearn.model_selection`, as well as `accuracy_score()` from `sklearn.metrics`
- Assign the `.data` attribute of `iris` to `data` and the `.target` attribute to `target`

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.

```
# 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
```

Use `train_test_split()` to split the data into training and test sets. Pass in the `data` and `target`, and set the `test_size` to 0.25 and `random_state` to 0.

```
X_train, X_test, y_train, y_test = train_test_split(data, target, test_size=0.25, ra
```



Now, instantiate the `KNN` class, and `fit` it to the data in `X_train` and the labels in `y_train`.

```
# Instantiate and fit KNN
knn = KNN()
knn.fit(X_train, y_train)
```

In the cell below, use the `.predict()` method to generate predictions for the data stored in `X_test`:

```
# Generate predictions
preds = knn.predict(X_test)
```

Finally, the moment of truth! Test the accuracy of your predictions. In the cell below, complete the call to `accuracy_score()` by passing in `y_test` and `preds` !

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

Testing Accuracy: 0.9736842105263158

Over 97% accuracy! Not bad for a handwritten machine learning classifier!

Summary

That was great! Next, you'll dive a little deeper into evaluating performance of a KNN algorithm!

Releases

No releases published

Packages

No packages published

Contributors 4



sumedh10 Sumedh Panchadhar



mike-kane Mike Kane



LoreDirick Lore Dirick



mathymitchell

Languages

● Jupyter Notebook 100.0%