Programming the KNN algorithm in python

KNN algorithm comes under the big vast supervised-learning umbrella of machine learning. It is one of the easiest algorithm to understand. This algortihm is used for both **regression** and **classification** problems. It performs great when compared to various other techniques such as linear regression, logistic regression, support vector machine etc.

Algorithms such as **linear regression** requires algorithms such as gradient descent to balance the weights of the feautures to improve the predictions. **Decision trees**, another amazing algorithm, has the problem of overfitting. ***KNN*** has no problem like these. It does not even require regularization techniques. It is just that good.

Understanding the algorithm

The word ***KNN*** is the short-form of k-nearest neighbors. The *k* here denotes the number of neighbors we want to account in the model. Let us that this is a binary classification problem, that is the target variable can only have values as either 0 or 1. Imagine a n dimensional space where n is the number of features in the training dataset. Each datapoint represents a coordinate position. Now that the **fitting** process has finished let us predict the target. Consider a test array. The array has a value for each feature. Now imagine this array in that n dimensional space. The algorithm just calculates the distance between the array and all the other points present in the training data. Assuming the value of *k* be 3 (you can assume any value of *k*). It then sorts the distances in ascending order, store the target values of the *k* arrays present in training set which are closest to the test array. It counts the number of 1's and 0's and then assigns the value which is maximum between these counts. There we go we have a prediction for the test array.

Note :- You can use either of **euclidean** or **manhatten** distance. It doesn't really matter which one.

Coding the algorithm

1. We will create a class for the algorithm so that when an instantiated of this class it has access to functions like fit() and predict() to fit the training data and predict target respectively. We will use euclidean distance in the algorithm, along with `numpy` and `collections` libraries.

```
import numpy as np
from collections import Counter


class KNNClassifier:
    def __init__(self,k=3):
        self.k = k
        return self
```

2. Now we will define the fit method for the class. The intended use of the function is to pass `X` , the feature array of the train set passed along with `y` the target values of the respected array in `X` . The method actually does nothing because the calculation of the distances matters during the prediction part.

```
import numpy as np
from collections import Counter


class KNNClassifier:
    def __init__(self, k=3) -> None:
        self.k = k
        return self

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y
```

3. We make another function named predict for the class. This is where the best part comes. This function returns the array of predicted values. It will take `X` , a collection of array(s) for which you want the labels. We will need a helper function for this which returns the target value for just a single array. We will also need a function which calculates the euclidean distance.

```python
import numpy as np
from collections import Counter


class KNNClassifier:
    def __init__(self, k=3) -> None:
        self.k = k
        return self

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def predict(self, X):
        predicted_labels = [self._predict(x) for x in X]
        return np.array(predicted_labels)

    def _euclidean_distance(self, x1, x2):
        return np.sqrt(np.sum((x1-x2)**2))

    def _predict(self, x):
        distances = [self._euclidean_distance(
            x, x_train) for x_train in self.X_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        most_common_class = Counter(k_nearest_labels).most_common(1)
        return most_common_class[0][0]
```

Hey there we go! We just made a machine learning algorithm! The program we just made can be imported just as any other module and you can use this to make predictions.

But we forgot about the regression part. There is just a small change in the algorithm. When using ***KNN*** for regresion it just takes the **mean** of the target values. The code for ***KNN*** for regression :-

```
import numpy as np


class KNNRegressor:
    def __init__(self, k=3) -> None:
        self.k = k

    def fit(self, X, y):
        self.X_train = X
        self.y_train = y

    def predict(self, X):
        predicted_labels = [self._predict(x) for x in X]
        return np.array(predicted_labels)

    def _euclidean_distance(self, x1, x2):
        return np.sqrt(np.sum((x1-x2)**2))

    def _predict(self, x):
        distances = [self._euclidean_distance(
            x, x_train) for x_train in self.X_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        return np.mean(k_nearest_labels)


def main():
    from sklearn.datasets import load_boston
    from sklearn.model_selection import train_test_split
    from sklearn.metrics import mean_squared_error

    boston = load_boston()
    X, y = boston.data, boston.target
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.3, random_state=42)

    clf = KNNRegressor(k=5)
    clf.fit(X_train, y_train)
    predicted = clf.predict(X_test)
    mse = mean_squared_error(y_test, predicted)
    print(f"MSE: {mse}")


# Testing the algorithm
if __name__ == '__main__':
    main()
```

Thank you for reading. Hope you enjoyed it.