KNN

Interview Questions -2

(Practice Project)

PW SKILLS

Question 1: Implement a k-NN classifier from scratch using Python. Use Euclidean distance to determine the nearest neighbors.

Solution:

```
import numpy as np
from collections import Counter

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

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

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

    def _predict(self, x):
        # Compute distances between x and all examples in the training set
        distances = [np.sqrt(np.sum((x - x_train) ** 2)) for x_train in
self.X_train]
        # Sort by distance and return indices of the first k neighbors
        k_indices = np.argsort(distances)[:self.k]
        # Extract the labels of the k nearest neighbor training samples
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        # Return the most common class label
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]

# Example usage:
X_train = np.array([[1, 2], [2, 3], [3, 4], [5, 6]])
y_train = np.array([0, 0, 1, 1])
X_test = np.array([[1, 2], [2, 3], [3, 4]])

knn = KNNClassifier(k=3)
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
print(predictions) #[0 0 0]
```

Question 2: Implement a k-NN regressor from scratch. Use Euclidean distance to determine the nearest neighbors.

Solution:

```
import numpy as np

class KNNRegressor:
    def __init__(self, k=3):
        self.k = k

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

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

    def _predict(self, x):
        # Compute distances between x and all examples in the training set
        distances = [np.sqrt(np.sum((x - x_train) ** 2)) for x_train in
self.X_train]
        # Sort by distance and return indices of the first k neighbors
        k_indices = np.argsort(distances)[:self.k]
        # Extract the target values of the k nearest neighbor training
samples
        k_nearest_values = [self.y_train[i] for i in k_indices]
        # Return the mean of the k nearest neighbors
        return np.mean(k_nearest_values)

# Example usage:
X_train = np.array([[1, 2], [2, 3], [3, 4], [5, 6]])
y_train = np.array([1.0, 2.0, 3.0, 5.0])
X_test = np.array([[1, 2], [2, 3], [3, 4]])

knn = KNNRegressor(k=3)
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
print(predictions) #[2. 2. 2.]
```

Question 3: Modify the k-NN classifier to use weighted voting, where closer neighbors have a higher vote weight.

Solution:

```
import numpy as np
from collections import Counter

class WeightedKNNClassifier:
    def __init__(self, k=3):
        self.k = k

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

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

    def _predict(self, x):
        # Compute distances between x and all examples in the training set
        distances = [np.sqrt(np.sum((x - x_train) ** 2)) for x_train in
self.X_train]
        # Sort by distance and return indices of the first k neighbors
        k_indices = np.argsort(distances)[:self.k]
        # Extract the labels of the k nearest neighbor training samples
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        k_nearest_distances = [distances[i] for i in k_indices]
        # Calculate weights as inverse of distance
        weights = [1 / d if d != 0 else 0 for d in k_nearest_distances]
        weighted_votes = Counter()
        for label, weight in zip(k_nearest_labels, weights):
            weighted_votes[label] += weight
        # Return the label with the highest weighted vote
        return weighted_votes.most_common(1)[0][0]

# Example usage:
X_train = np.array([[1, 2], [2, 3], [3, 4], [5, 6]])
y_train = np.array([0, 0, 1, 1])
X_test = np.array([[1, 2], [2, 3], [3, 4]])

knn = WeightedKNNClassifier(k=3)
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
print(predictions) # [0 0 0]
```

Question 4: Implement a distance-weighted k-NN regressor where the influence of each neighbor is inversely proportional to its distance.

Solution:

```
import numpy as np

class DistanceWeightedKNNRegressor:
    def __init__(self, k=3):
        self.k = k

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

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

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

    def _predict(self, x):
        # Compute distances between x and all examples in the training set
        distances = [np.sqrt(np.sum((x - x_train) ** 2)) for x_train in
self.X_train]
        # Sort by distance and return indices of the first k neighbors
        k_indices = np.argsort(distances)[:self.k]
        # Extract the target values of the k nearest neighbor training
samples
        k_nearest_values = [self.y_train[i] for i in k_indices]
        k_nearest_distances = [distances[i] for i in k_indices]
        # Calculate weights as inverse of distance
        weights = [1 / d if d != 0 else 0 for d in k_nearest_distances]
        # Return the weighted mean of the k nearest neighbors
        return np.sum(np.multiply(k_nearest_values, weights)) /
np.sum(weights)

# Example usage:
X_train = np.array([[1, 2], [2, 3], [3, 4], [5, 6]])
y_train = np.array([1.0, 2.0, 3.0, 5.0])
X_test = np.array([[1, 2], [2, 3], [3, 4]])

knn = DistanceWeightedKNNRegressor(k=3)
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
print(predictions)

# The output:
[2.33333333 2.         1.66666667]
```

Question 5: Implement an approximate k-NN classifier using Locality-Sensitive Hashing (LSH) to speed up the nearest neighbor search.

Solution:

```
    def _predict(self, x):
        candidates = self._get_candidates(x)
        if not candidates:
            return None

        # Compute distances between x and all candidate examples
        distances = [np.sqrt(np.sum((x - self.X_train[i]) ** 2)) for i in
candidates]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[list(candidates)[i]] for i in
k_indices]

        # Return the most common class label among the k nearest neighbors
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]

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

# Example usage:
X_train = np.array([[1, 2], [2, 3], [3, 4], [5, 6], [6, 7], [7, 8]])
y_train = np.array([0, 0, 1, 1, 1, 0])
X_test = np.array([[1, 2], [2, 3], [3, 4]])

knn = LSHKNNClassifier(k=3, num_hashes=5)
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
print(predictions)

#The Output
[0,0,0]
```

Question 6: Implement a brute-force k-NN classifier and compare its performance with an optimized k-NN implementation using KD-Tree.

Solution:

```
from sklearn.neighbors import KDTree

class BruteForceKNNClassifier:
    def __init__(self, k=3):
        self.k = k

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

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

    def _predict(self, x):
        distances = [np.sqrt(np.sum((x - x_train) ** 2)) 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 = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]

# Example usage:
X_train = np.random.rand(1000, 2)
y_train = np.random.randint(0, 2, size=1000)
X_test = np.random.rand(10, 2)

# Brute-force k-NN
brute_knn = BruteForceKNNClassifier(k=5)
brute_knn.fit(X_train, y_train)
brute_predictions = brute_knn.predict(X_test)

# Optimized k-NN using KD-Tree
kd_tree = KDTree(X_train)
_, indices = kd_tree.query(X_test, k=5)
kd_predictions = [Counter(y_train[i]).most_common(1)[0][0] for i in
indices]

print("Brute-force k-NN predictions:", brute_predictions)
print("KD-Tree k-NN predictions:", kd_predictions)

#The Output:
Brute-force k-NN predictions: [0 1 0 0 1 0 0 1 0 0]
KD-Tree k-NN predictions: [0, 1, 0, 0, 1, 0, 0, 1, 0, 0]
```

Question 7: Implement the construction of a k-D Tree from a given dataset.

Solution:

```
class KDTreeNode:
    def __init__(self, point, left=None, right=None):
        self.point = point
        self.left = left
        self.right = right

def build_kd_tree(points, depth=0):
    if not points:
        return None

    k = len(points[0])
    axis = depth % k

    points.sort(key=lambda x: x[axis])
    median = len(points) // 2


    return KDTreeNode(
        point=points[median],
        left=build_kd_tree(points[:median], depth + 1),
        right=build_kd_tree(points[median + 1:], depth + 1)
    )

# Example usage:
points = [(2, 3), (5, 4), (9, 6), (4, 7), (8, 1), (7, 2)]
kd_tree = build_kd_tree(points)

def print_kd_tree(node, depth=0):
    if not node:
        return
    print(" " * depth, node.point)
    print_kd_tree(node.left, depth + 2)
    print_kd_tree(node.right, depth + 2)

print_kd_tree(kd_tree)

#The output:
(7, 2)
  (5, 4)
    (2, 3)
    (4, 7)
  (9, 6)
    (8, 1)
```

Question 8. k-Dimensional Tree (k-D Tree) Nearest Neighbor Search
Question: Implement a nearest neighbor search using a k-D Tree.

Solution:

```
def kd_tree_nearest_neighbor(root, point, depth=0, best=None):
    if root is None:
        return best

    k = len(point)
    axis = depth % k


    next_best = None
    next_branch = None

    if best is None or np.linalg.norm(np.array(point) -
np.array(root.point)) < np.linalg.norm(np.array(point) -
np.array(best.point)):
        next_best = root
    else:
        next_best = best

    if point[axis] < root.point[axis]:
        next_branch = kd_tree_nearest_neighbor(root.left, point, depth + 1,
next_best)
        if next_branch:
            next_best = next_branch
        if np.abs(point[axis] - root.point[axis]) <
np.linalg.norm(np.array(point) - np.array(next_best.point)):
            next_branch = kd_tree_nearest_neighbor(root.right, point, depth
+ 1, next_best)
            if next_branch:
                next_best = next_branch
    else:
        next_branch = kd_tree_nearest_neighbor(root.right, point, depth +
1, next_best)
        if next_branch:
            next_best = next_branch
        if np.abs(point[axis] - root.point[axis]) <
np.linalg.norm(np.array(point) - np.array(next_best.point)):
            next_branch = kd_tree_nearest_neighbor(root.left, point, depth
+ 1, next_best)
            if next_branch:
                next_best = next_branch

    return next_best

# Example usage:
point = (9, 2)
nearest = kd_tree_nearest_neighbor(kd_tree, point)
print("Nearest neighbor to", point, "is", nearest.point)

#The Output:
```

```
Nearest neighbor to (9, 2) is (8, 1)
```

Question 9: Implement a weighted k-NN regressor that allows different distance metrics (e.g., Manhattan, Minkowski).

Solution:

```
import numpy as np

class WeightedKNNRegressor:
    def __init__(self, k=3, metric='euclidean'):
        self.k = k
        self.metric = metric

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

    def _distance(self, x1, x2):
        if self.metric == 'euclidean':
            return np.sqrt(np.sum((x1 - x2) ** 2))
        elif self.metric == 'manhattan':
            return np.sum(np.abs(x1 - x2))
        elif self.metric == 'minkowski':
            p = 3  # Example value for p
            return np.sum(np.abs(x1 - x2) ** p) ** (1 / p)
        else:
            raise ValueError("Unsupported metric")

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

    def _predict(self, x):
        distances = [self._distance(x, x_train) for x_train in
self.X_train]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_values = [self.y_train[i] for i in k_indices]
        weights = [1 / d if d != 0 else 0 for d in [distances[i] for i in
k_indices]]
        return np.sum(np.multiply(k_nearest_values, weights)) /
np.sum(weights)

# Example usage:
X_train = np.array([[1, 2], [2, 3], [3, 4], [5, 6]])
y_train = np.array([1.0, 2.0, 3.0, 5.0])
X_test = np.array([[1, 2], [2, 3], [3, 4]])

knn = WeightedKNNRegressor(k=3, metric='manhattan')
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
print(predictions)

#The Output:
[2.33333333 2.         1.66666667]
```

Question 10: Implement k-fold cross-validation for a k-NN classifier and report the average accuracy.

Solution:

```
from sklearn.model_selection import KFold

def cross_val_score(knn, X, y, k_folds=5):
    kf = KFold(n_splits=k_folds)
    accuracies = []

    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]

        knn.fit(X_train, y_train)
        predictions = knn.predict(X_test)
        accuracy = np.mean(predictions == y_test)
       accuracies.append(accuracy)

    return np.mean(accuracies)

# Example usage:
from sklearn.datasets import load_iris

# Load Iris dataset
iris = load_iris()
X, y = iris.data, iris.target

# Initialize k-NN classifier
knn = BruteForceKNNClassifier(k=3)

# Perform 5-fold cross-validation
average_accuracy = cross_val_score(knn, X, y, k_folds=5)
print(f"Average accuracy from cross-validation: {average_accuracy:.2f}")
# The output :Average accuracy from cross-validation: 0.91
```

Question 11: Implement an approximate k-NN search using Locality-Sensitive Hashing (LSH) and compare its performance to brute-force k-NN.

Solution:

```
from sklearn.utils import murmurhash3_32

class LSHApproximateKNN:
    def __init__(self, k=3, num_hashes=10):
        self.k = k
        self.num_hashes = num_hashes

    def _hash_function(self, x, seed):
        return murmurhash3_32(x, seed=seed)

    def _lsh_signature(self, x):
        return [self._hash_function(x, seed) for seed in
range(self.num_hashes)]

    def fit(self, X_train, y_train):
        self.X_train = X_train
        self.y_train = y_train
        self.lsh_buckets = {}
        for i, x in enumerate(X_train):
            signature = tuple(self._lsh_signature(x.tostring()))
            if signature not in self.lsh_buckets:
                self.lsh_buckets[signature] = []
            self.lsh_buckets[signature].append(i)

    def _predict(self, x):
        signature = tuple(self._lsh_signature(x.tostring()))
        candidates = self.lsh_buckets.get(signature, [])
        if not candidates:
            return None

        distances = [np.linalg.norm(x - self.X_train[i]) for i in
candidates]
        k_indices = np.argsort(distances)[:self.k]
        k_nearest_labels = [self.y_train[i] for i in k_indices]
        most_common = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]

    def predict(self, X_test):
        return np.array([self._predict(x) for x in X_test])
```

```
# Example usage:
X_train = np.random.rand(1000, 10)
y_train = np.random.randint(0, 2, size=1000)
X_test = np.random.rand(10, 10)

# Brute-force k-NN
brute_knn = BruteForceKNNClassifier(k=5)
brute_knn.fit(X_train, y_train)
brute_predictions = brute_knn.predict(X_test)

# LSH k-NN
lsh_knn = LSHApproximateKNN(k=5, num_hashes=15)
lsh_knn.fit(X_train, y_train)
lsh_predictions = lsh_knn.predict(X_test)

print("Brute-force k-NN predictions:", brute_predictions)
print("LSH Approximate k-NN predictions:", lsh_predictions)
```

The output:

```
Brute-force k-NN predictions: [0 0 0 1 0 1 1 0 0 0]
LSH Approximate k-NN predictions: [None None None None None None None None None None]
```

Question 12: Implement a k-NN classifier where neighbors are weighted by their inverse distance.

Solution:

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

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

    def predict(self, X_test):
        return np.array([self._predict(x) for x in X_test])
```

```
    def _predict(self, x):
        distances = [np.linalg.norm(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]
        weights = [1 / d if d != 0 else 0 for d in
np.array(distances)[k_indices]]
        weighted_vote = np.bincount(k_nearest_labels, weights=weights)
        return np.argmax(weighted_vote)

# Example usage:
X_train = np.random.rand(100, 2)
y_train = np.random.randint(0, 2, size=100)
X_test = np.random.rand(10, 2)

knn = DistanceWeightedKNNClassifier(k=5)
knn.fit(X_train, y_train)
predictions = knn.predict(X_test)
print(predictions)

# The output: [0 1 0 0 0 1 0 0 0 1]
```