CSE4288

Introduction to Machine Learning

K-Nearest Neighbor Classifier

Mohamad Nael Ayoubi

150120997

- **Introduction**

The k-Nearest Neighbor (k-NN) Algorithm: The k-Nearest Neighbor (k-NN) algorithm is a simple, yet powerful supervised learning method widely used for classification and regression tasks. It is a lazy learner, meaning it does not build an explicit model during the training phase. Instead, it stores the training data and makes predictions based on the majority class of the k closest neighbors in feature space. All instances correspond to points in the n-D space. The closeness is determined by a distance metric, such as Euclidean or Manhattan distance. The objective of this assignment is to understand and implement the k-NN algorithm from scratch without using machine learning libraries like scikit-learn. The tasks include:

1. Preparing the Play Tennis dataset.
2. Implementing the k-NN algorithm for classification.
3. Evaluating its performance using metrics such as accuracy and a confusion matrix.
4. Analyzing the results and identifying any challenges during implementation.

- **Methodology**

1. Data Preparation:

The Play Tennis dataset consists of categorical features (e.g., Outlook, Temperature, Humidity, Wind) and a target variable (PlayTennis), which were converted to numeric form using one-hot encoding. As shown below:

```python
def preprocess(data):
    # one-hot encode the data to convert categorical features to numerical values
    one_hot_encoded_data = []

    unique_values = {
        "Outlook": ["Sunny", "Overcast", "Rain"],
        "Temperature": ["Hot", "Mild", "Cool"],
        "Humidity": ["High", "Normal"],
        "Wind": ["Weak", "Strong"],
        "PlayTennis": ["No", "Yes"]
    }

    for instance in data:
        one_hot_instance = {}

        # for each unique value of each feature, create a new feature with the value 1 if the instance has that value, 0 otherwise
        for key in unique_values:
            if key in instance:
                for value in unique_values[key]:
                    one_hot_instance[f"{key}_{value}"] = 1 if instance[key] == value else 0

        one_hot_encoded_data.append(one_hot_instance)

    return one_hot_encoded_data
```

Example of converted version:

```
{'Outlook_Sunny': 1, 'Outlook_Overcast': 0, 'Outlook_Rain': 0, 'Temperature_Hot': 1, 'Temperature_Mild': 0, 'Temperature_Cool': 0, 'Humidity_High': 1, 'Humidity_Normal': 0, '
{'Outlook_Sunny': 1, 'Outlook_Overcast': 0, 'Outlook_Rain': 0, 'Temperature_Hot': 1, 'Temperature_Mild': 0, 'Temperature_Cool': 0, 'Humidity_High': 1, 'Humidity_Normal': 0, '
{'Outlook_Sunny': 0, 'Outlook_Overcast': 1, 'Outlook_Rain': 0, 'Temperature_Hot': 1, 'Temperature_Mild': 0, 'Temperature_Cool': 0, 'Humidity_High': 1, 'Humidity_Normal': 0, '
```

The dataset is stored in JSON format (dataset.json).

2. Implementation Details:

The kNN_classifier class was developed with:

- Support for both Euclidean and Manhattan distance metrics.
- Hyperparameter k, allowing user input to control the number of neighbors.
- Predictions based on majority voting among the k nearest neighbors.

Lastly, its performance was evaluated using accuracy and Confusion Matrix.

How distance and predictions has been calculated is shown below:

```python
class kNN_classifier:
    def __init__(self, K, distance_metric):
        self.K = K
        self.distance_metric = distance_metric
        self.data = self.load_data()

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

    def manhattan_distance(self, x1, x2):
        return np.sum(np.abs(x1 - x2))

    def load_data(self):
        with open('dataset.json') as f:
            self.data = json.load(f)
            self.data = preprocess(self.data)
        return self.data

    # calculate the distance between two instances according to the distance metric
    def distance(self, x1, x2):
        if self.distance_metric == 'euclidean':
            return self.euclidean_distance(x1, x2)
        elif self.distance_metric == 'manhattan':
            return self.manhattan_distance(x1, x2)
        else:
            raise ValueError("Invalid distance metric")
```

```python
    def predict(self, test_instance):
        distances = []
        for instance in self.data:
            # features are all the values except the class label
            features = np.array([value for key, value in instance.items() if not key.startswith("PlayTennis")])
            dist = self.distance(features, test_instance)
            distances.append((instance, dist))

        # sort the distances in ascending order. x[1] is the distance
        distances.sort(key=lambda x: x[1])
        neighbors = distances[:self.K]
```

3. Challenges:

One-hot encoding conversion required careful preprocessing to avoid misalignment between features.

- **Results**

For K = 3 and Euclidean distance picked as a distance metric I got the following results using the same dataset for testing.

- Accuracy of 0.71 is achieved
- Confusion matrix is shown below

```
Confusion Matrix
TP: 7, TN: 3, FP: 2, FN: 2
Accuracy:  0.7142857142857143
```

For K = 5 and with the same distance metric I got the following results:

- Accuracy of 0.79
- Confusion matrix is shown below

```
Confusion Matrix
TP: 7, TN: 4, FP: 1, FN: 2
Accuracy:  0.7857142857142857
```

- **Discussion**

1. Analysis of Results:
   - The high accuracy indicates that the k-NN algorithm performed well on the Play Tennis dataset, likely due to the small size and clear separability of the data.
   - Misclassifications (FP and FN) could arise from: Insufficient neighbor information (k value too small).
2. Limitations of the Implementation:
   - Scalability: The algorithm processes every data point in the training set for each prediction, making it inefficient for large datasets
   - One-hot encoding does not capture the ordinal relationships between categories effectively.

- **Conclusion**

The k-NN algorithm's simplicity makes it a good starting point for understanding instance-based learning and lazy classifiers. Proper preprocessing and careful parameter tuning (such as choosing the right value of k) significantly impact the algorithm's performance.
Overfitting and Underfitting in k-NN:

Overfitting: When k is set to a very low value (e.g., k=1), the model may overfit to the training data. It becomes highly sensitive to noise and specific data points, leading to poor generalization on unseen data. This happens because the prediction is based on a single closest neighbor, which might not represent the overall distribution.

Underfitting: Conversely, when k is too large, the model may underfit. It considers too many neighbors, potentially including data points that are not relevant to the test instance.

- **References**

[1] Classification Basic slides given in lecture.

[2] Introduction to Machine Learning by Ethem Alpaydin