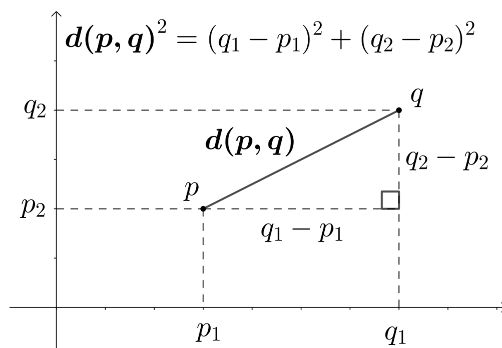
Lesson 4: k-Nearest Neighbors

k-Nearest Neighbors (k-NN) is a popular non-parametric machine learning algorithm used for both classification and regression tasks. In this chapter, we will explore the details of k-NN, including distance metrics, parameter tuning, and ensemble methods. The k-NN algorithm works by finding the k closest data points in the training set to a new data point and using the labels of these neighbors to make a prediction. For classification tasks, the predicted label is the majority class among the k nearest neighbors. For regression tasks, the predicted value is the average of the k nearest neighbors.

Distance Metrics

The choice of distance metric is a crucial component in the k-NN algorithm, as it determines how the algorithm calculates the distance between two data points. The choice of distance metric will have an impact on the performance of the algorithm, as it can affect how well the algorithm is able to identify patterns and relationships in the data.



Euclidean distance is the most commonly used distance metric in k-NN, and it is calculated as the square root of the sum of the squared differences between each feature in the two data points. It assumes that the features are independent and that their contribution to the distance is equal. Euclidean distance works well for data that has similar units and scales.

Manhattan distance, also known as L1 distance, is an alternative distance metric that is calculated as the sum of the absolute differences between each feature in the two data points. Unlike Euclidean distance, Manhattan distance does not assume that the features are independent or that their contribution to the distance is equal. Manhattan distance works well for data that has different units or scales.

$$\text{Mdist} = |x2 - x1| + |y2 - y1|$$

Cosine similarity is another distance metric that is used for high-dimensional data. Instead of measuring the distance between two data points in a vector space, cosine similarity measures the cosine of the angle between the two data points. Cosine similarity is particularly useful for text data, where the features are words and the vector space is high-dimensional.

$$similarity(A,B) = \frac{A \cdot B}{\|A\| \times \|B\|} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

Choosing the appropriate distance metric depends on the nature of the data and the task at hand. It is important to experiment with different distance metrics to determine which one works best for a given problem.

In addition to the choice of distance metric, the choice of the number of neighbors, k, is also an important consideration in k-NN. Choosing k depends on the complexity of the data and the size of the training set. If k is too small, the algorithm may be overly sensitive to noise in the data, while if k is too large, the algorithm may not be able to capture the underlying patterns in the data.

Choosing k

The value of k in k-NN algorithm is a critical hyperparameter that has a significant impact on the accuracy and generalization ability of the model. It determines how many nearest neighbors are considered when making a prediction. Choosing the optimal value of k is crucial for the performance of the model.

If k is too small, the model may suffer from overfitting, where the model learns the noise in the data and fails to generalize to new data. On the other hand, if k is too large, the model may suffer from underfitting, where the model is too simple and fails to capture the underlying patterns in the data.

There are several approaches to choosing the optimal value of k. One popular approach is to use cross-validation, where the data is split into training and validation sets, and

the model is trained on the training set and evaluated on the validation set for different values of k. The value of k that gives the highest accuracy or other performance metric on the validation set is selected as the optimal value of k.

Another approach is to use techniques such as grid search or random search to find the optimal value of k. In grid search, a predefined set of values for k is specified, and the model is trained and evaluated for each value of k using cross-validation. The value of k that gives the highest accuracy or other performance metric on the validation set is selected as the optimal value of k.

In random search, a range of values for k is specified, and the model is trained and evaluated for randomly selected values of k using cross-validation. The value of k that gives the highest accuracy or other performance metric on the validation set is selected as the optimal value of k.

Parameter Tuning

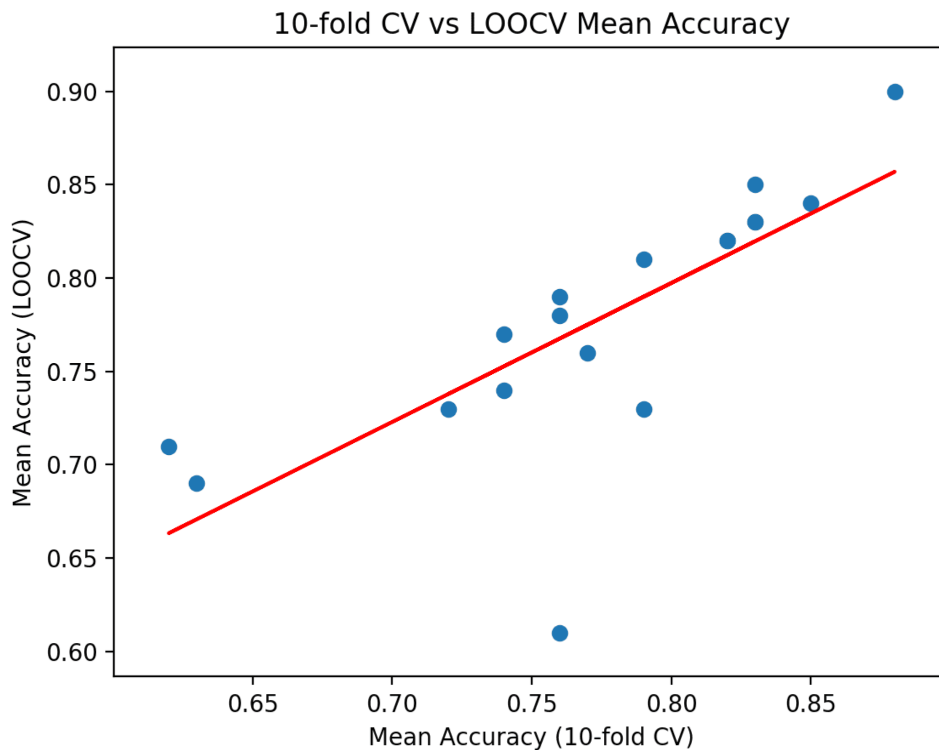
Parameter tuning is a crucial step in machine learning that involves optimizing the hyperparameters of a model to achieve the best performance. In k-NN, the choice of the number of neighbors k is a critical hyperparameter that can significantly impact the performance of the model. A smaller value of k may lead to overfitting, while a larger value of k may lead to underfitting.

To determine the optimal value of k, various techniques can be used for parameter tuning. One such technique is **grid search**, which involves exhaustively searching over a predefined range of k values and evaluating the performance of the model for each value. Grid search can be time-consuming and computationally expensive, especially when the number of hyperparameters and the size of the search space are large.

Another technique for parameter tuning is **random search**, which involves randomly sampling from the search space of hyperparameters and evaluating the performance of the model for each sample. Random search is more efficient than grid search, as it does not require exhaustive searching of the entire parameter space. Random search can be particularly useful when the search space is large and the optimal hyperparameters are not known.

Cross-validation is another advanced technique that can be used for parameter tuning. Cross-validation involves splitting the training data into multiple subsets and using each subset as both training and validation data.

By evaluating the performance of the model on different subsets, cross-validation can provide a more accurate estimate of the performance of the model and help prevent overfitting.



Ensemble Methods

Ensemble methods are a popular technique used to improve the performance of machine learning models. In k-NN, ensemble methods can be used to combine multiple models to achieve better prediction accuracy.

Bagging, short for bootstrap aggregating, is a common ensemble method used in k-NN. It involves training multiple k-NN models on different subsets of the training data and combining their predictions by averaging. This technique helps to reduce overfitting and improve the stability of the model.

Boosting is another popular ensemble method used in k-NN. It involves training a sequence of k-NN models on weighted versions of the training data, with the weights updated based on the performance of the previous model. This technique helps to focus

the model on the areas where it performs poorly and improve the accuracy of the model.

Both bagging and boosting have their advantages and disadvantages. Bagging is a simple and effective method that can reduce overfitting and improve model stability, but it may not improve accuracy as much as boosting. Boosting, on the other hand, can improve the accuracy of the model, but it may also increase the risk of overfitting and reduce model stability.

In practice, the choice of ensemble method and its hyperparameters depends on the specific problem and dataset. The performance of different ensemble methods can be compared using cross-validation or other evaluation metrics.

Code Example

Here is an example of training a k-NN classifier on the Iris dataset using scikit-learn in Python:

```
from sklearn.datasets import load_iris
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import train_test_split

# Load the Iris dataset
iris = load_iris()

# Split the data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2)

# Train a k-NN classifier with k=5
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

# Make predictions on the test set
y_pred = knn.predict(X_test)
```

```
# Evaluate the performance of the model
from sklearn.metrics import accuracy_score
accuracy = accuracy_score(y_test, y_pred)
print("Accuracy: {:.2f}".format(accuracy))
```

This code is an example of using k-Nearest Neighbors (k-NN) classifier to predict the species of iris flowers based on their measurements.

The code imports necessary modules from the scikit-learn library, including the load_iris dataset function, KNeighborsClassifier class, train_test_split function for splitting the data into training and test sets, and accuracy_score function for evaluating the model's performance.

The first step is to load the iris dataset using the load_iris function, which returns an object containing the dataset's features, target values, and other information.

Then, the data is split into training and test sets using the train_test_split function, where 20% of the data is used for testing, and the rest is used for training.

Next, a k-NN classifier with k=5 is trained on the training set using the KNeighborsClassifier class.

The model is then used to make predictions on the test set using the predict method of the classifier.

Finally, the accuracy of the model is evaluated by comparing the predicted values with the actual target values of the test set using the accuracy_score function.

The output of the code is the accuracy of the model on the test set, which represents the percentage of correctly classified instances among all instances in the test set.