Lab 01: KNN

Fabian Haas, Markus Reichl, Florian Weingartshofer

KNN Implementation

Assumptions: We are free to implement any distance function

The KNN uses a simple Euclidean distance function for calculating the distance. This has been omitted for conciseness.

The algorithm itself calculates the distance for every data entry in the train set for its given sample. This result is sorted ascending and the first `k` entries are used to calculate the most common result.

Basic KNN Implementation

```
class KNNClassifier:
    # Initialization and fitting...

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

    def _predict(self, x):
        distances = [self.distance_function(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 = Counter(k_nearest_labels).most_common(1)
        return most_common[0][0]
```

Creating the Train/Test Sets

The train/test sets are created with a split of `n=0.7`, meaning that the train set will contain 70% of the available data.

Evaluating KNN

The following code evaluates the KNN algorithm with `0 < k < 20`. The results indicate that the best `k` is `18`.

The implementation of the KNN algorithm shows how tweaking the `k` value can drastically improve the result of the model. Finding the right `k` can take some time, but can be optimized by creating a simple algorithm for the hyperparameter selection – too small, and it can overfit, too big, and it might underfit.