AI and Machine Learning Homework3 Report 纪浩正 12311405

Student: 纪浩正, jihz2023@mail.sustech.edu.cn

1 K nearest Neighbors

The k-nearest neighbors (k-NN) algorithm determines the label of a data point by identifying its closest neighbors within the dataset and assigning the most frequently occurring label among them.

1.1 Data Splitting

We use the Breast Cancer Wisconsin dataset, which contains two classes: benign (label 0) and malignant (label 1). The data were split into a training set (70%) and a test set (30%) for each class independently. Table 1 summarizes the number of samples used.

Table 1 Data distribution for training and test sets

Class	Train Samples	Test Samples	Total Samples
Benign (0)	250	107	357
Malignant (1)	148	64	212

Load data and split into training and test sets:

```
np.random.seed(42)
df=pd.read_csv("wdbc.data",header=None)
data_features=df.iloc[:,2:12].values
data_labels=df.iloc[:,1].values
data_labels = (data_labels == 'M').astype(int)

table=np.zeros(len(data_labels),dtype=bool)
for ii in range(2):
    idx=np.where(data_labels==ii)[0]
    np.random.shuffle(idx)
    number=round(len(idx)*0.7)
    table[idx[:number]]=True
    print(f"\nFor label {ii} ({'Benign' if ii == 0 else 'Malignant'}):")
    print(f" Total samples: {len(idx)}")
    print(f" Train samples: {number}")
    print(f" Test samples: {len(idx) - number}")
train_features,train_labels=data_features[table],data_labels[table]
test_features,test_labels=data_features[~table],data_labels[~table]
```

1.2 KNN Algorithm Implementation

In this section, we implement the k-nearest neighbors (KNN) algorithm. The parameter k represents the number of nearest neighbors considered when assigning a label to a test sample.

Initialization and Normalization

```
class KnnClassifier:
    def __init__(self,features,labels,normalization=None):
        self.X_train=features
        self.label=labels
        self.num,self.dim=features.shape

        self.normalization=normalization
        if normalization:
            self.mean=self.X_train.mean(axis=0)
            self.std=self.X_train.mean(axis=0)
            for ii in range(self.dim):
                self.X_train[:,ii]=(self.X_train[:,ii]-self.mean[ii])/self.std[ii]
```

Distance Calculation and Neighbor Selection

```
    def _Euclidean_distance(self,x):
        diff=self.X_train-x
        diff2=abs(diff**self.mode)
        dist=np.sum(diff2,axis=1)
        return dist

    def get_nearest(self,x,k):
        dis_list=self._Euclidean_distance(x)
        return np.argsort(dis_list)[:k]
```

Prediction and Accuracy Calculation

```
    def pred(self,x,k):
        N=x.shape[0]
        x_copy=x.copy().astype(np.float64)
        if self.normalization:
            for ii in range(self.dim):
                x_copy[:,ii]=(x_copy[:,ii]-self.mean[ii])/self.std[ii]
        pred=[]
        for ii in range(N):
            test=x_copy[ii,:]
            k_nearest_idx=self.get_nearest(test,k)
            k_nearest_label=self.label[k_nearest_idx]
            pred.append(Counter(k_nearest_label).most_common(1)[0][0])
        return np.array(pred)
```

```
def accuracy(self,x,label,k,mode):
    self.mode=mode
    pred=self.pred(x,k)
    accuracy=np.mean(pred==label)*100
    true_idx=np.where(pred==label)[0]
    false_idx=np.where(pred!=label)[0]
    return accuracy,pred,true_idx,false_idx
```

1.3 Testing and Analysis

For each mode n, the distance is defined as follows:

$$L_n = \sum_{i=1}^{dim} |x_i - \hat{x}_i|^n$$

From the figures below, we can see that most accuracies are above 90%. As the value of k increases, the accuracy first rises and then decreases. For modes 1, 8, and 10, the accuracy can reach up to 95%, with the optimal values of k being 8 for mode 1, 5 for mode 8, and 3 for mode 10. Among these modes, mode 10 has the fastest computation speed, while modes 1 and 8 achieve the highest accuracies.

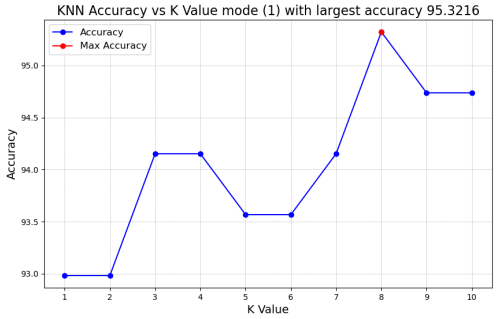


Figure 1 KNN Accuracy with K=1

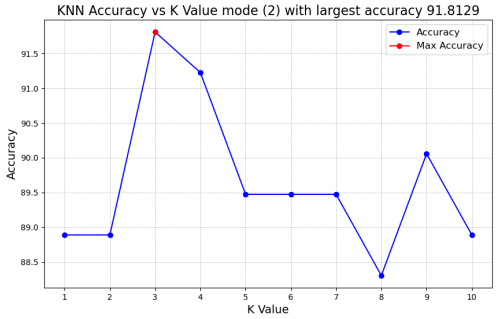


Figure 2 KNN Accuracy with K=2

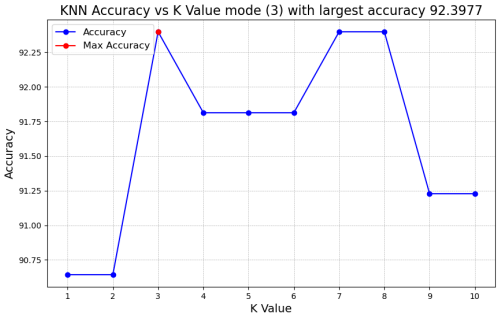


Figure 3 KNN Accuracy with K=3

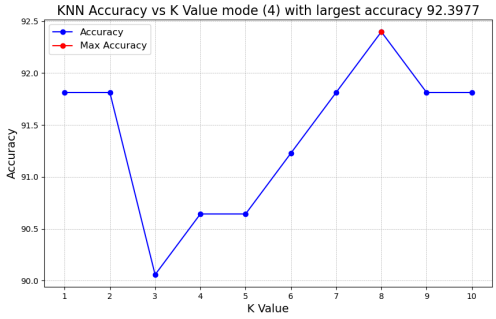


Figure 4 KNN Accuracy with K=4

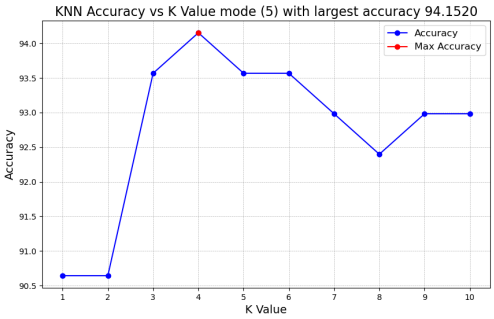


Figure 5 KNN Accuracy with K=5

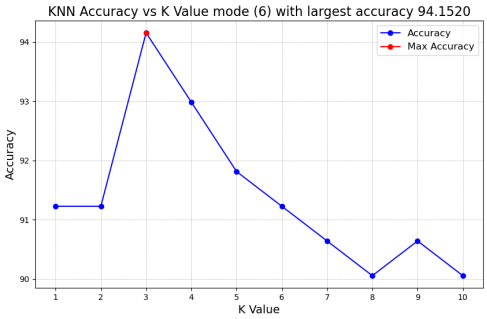


Figure 6 KNN Accuracy with K=6

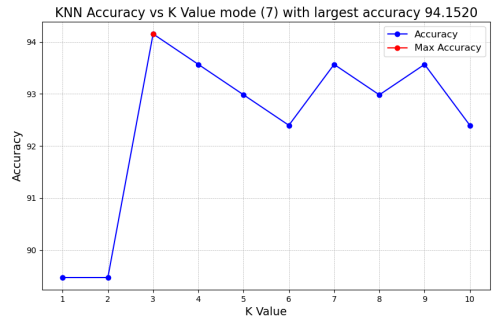


Figure 7 KNN Accuracy with K=7

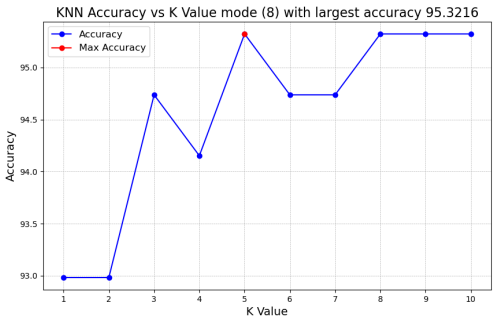


Figure 8 KNN Accuracy with K=8

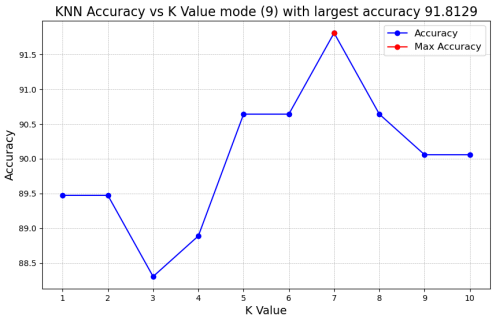


Figure 9 KNN Accuracy with K=9

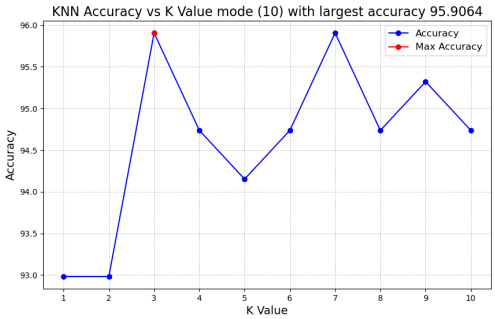


Figure 10 KNN Accuracy with K=10

2 Decision Tree

Decision trees are hierarchical models that recursively split the data based on feature values in order to form decision rules for classification. They are easy to interpret, capable of handling nonlinear relationships, and can capture interactions between features through branching structures. In this experiment, decision trees were trained using different sizes of training data in order to analyze how the amount of data affects tree depth and accuracy.

2.1 Decision Tree Algorithm Implementation

Tree Structure and Initialization

```
class DecisionTree:
    class Node:
        def __init__(self) -> None:
            self.value = None
            self.feature_index = None
            self.children = {}

        def __str__(self) -> str:
            if self.children:
                s = f'Internal node <{self.feature_index}>:\n'
                for fv, node in self.children.items():
                    ss = f'[{fv}]-> {node}'
                    s += '\t' + ss.replace('\n', '\n\t') + '\n'
            else:
                s = f'Leaf node ({self.value})'
            return s

    def __init__(self, gain_threshold=1e-2) -> None:
        self.gain_threshold = gain_threshold
```

Entropy and Information Gain

```
    def _entropy(self, y):
        count_y = np.bincount(y)
        prob_y = count_y[np.nonzero(count_y)] / y.size
        entropy_y = -np.sum(prob_y * np.log2(prob_y))
        return entropy_y

    def _conditional_entropy(self, feature, y):
        feature_values = np.unique(feature)
        h = 0.
        for v in feature_values:
            y_sub = y[feature == v]
```

```
            prob_y_sub = y_sub.size / y.size
            h += prob_y_sub * self._entropy(y_sub)
        return h

    def _information_gain(self, feature, y):
        ig_feature = self._entropy(y) - self._conditional_entropy(feature, y)
        return ig_feature
```

Recursive Tree Construction

```
    def _build_tree(self, X, y, features_list):
        node = self.Node()
        labels_count = np.bincount(y)
        node.value = np.argmax(labels_count)

        if np.count_nonzero(labels_count) != 1:
            index = self._select_feature(X, y, features_list)
            if index is not None:
                node.feature_index = features_list.pop(index)
                feature_values = np.unique(X[:, node.feature_index])
                for v in feature_values:
                    idx = X[:, node.feature_index] == v
                    X_sub, y_sub = X[idx], y[idx]
                    node.children[v] = self._build_tree(X_sub, y_sub, features_list.copy())
        return node
```

Prediction

```
    def _predict_one(self, x):
        node = self.tree_
        while node.children:
            child = node.children.get(x[node.feature_index])
            if not child:
                break
            node = child
        return node.value

    def predict(self, X):
        return np.apply_along_axis(self._predict_one, axis=1, arr=X)
```

2.2 Result and Analysis

Figures 11 and 12 show how tree depth and classification accuracy vary with the size of the training set. As the training set becomes larger, the model generally becomes deeper and is able

to extract more detailed decision rules. Accuracy also tends to improve until reaching a stable range.

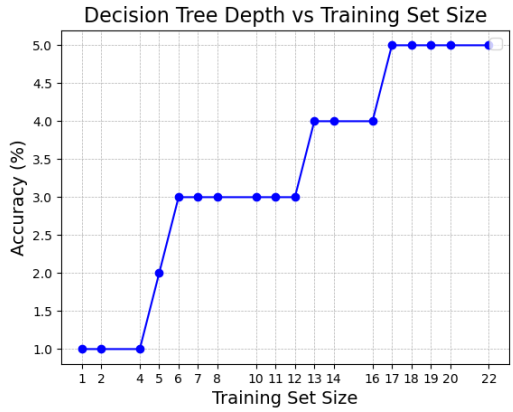


Figure 11 Tree depth as a function of training set size

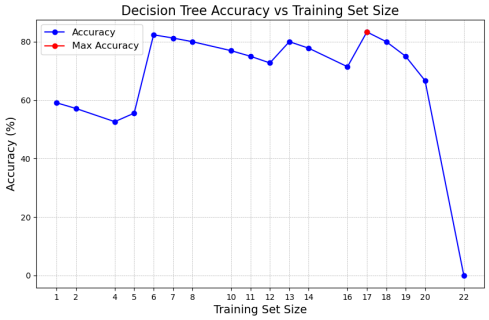
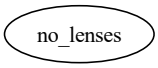


Figure 12 Classification accuracy versus training set size

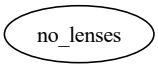
Tree depth calculation:

```
def calculate_tree_depth(self, node):
    if not node.children:
        return 1
    else:
        max_child_depth = max(self.calculate_tree_depth(child) for child in
            node.children.values())
        return max_child_depth + 1
```

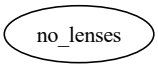
To visualize how tree structure changes with different training sizes, the following figures display all generated decision trees. Six trees are shown per row for compact layout.



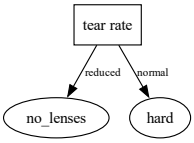
Training Size=1



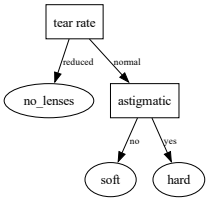
Training Size=2



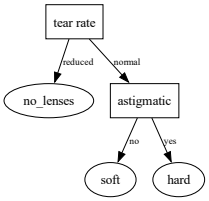
Training Size=4



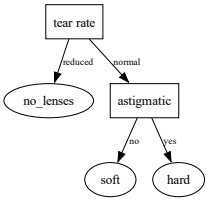
Training Size=5



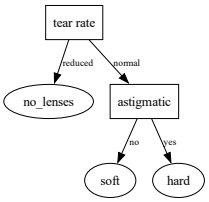
Training Size=6



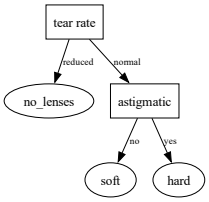
Training Size=7



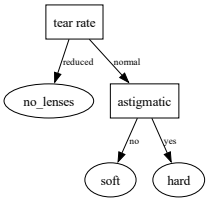
Training Size=8



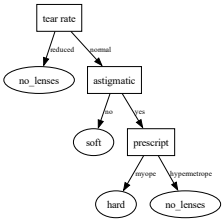
Training Size=10



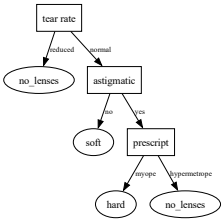
Training Size=11



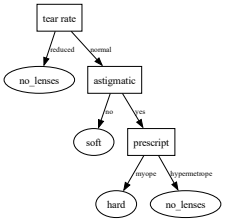
Training Size=12



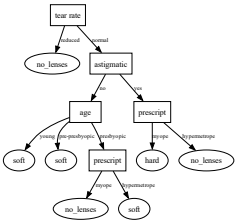
Training Size=13



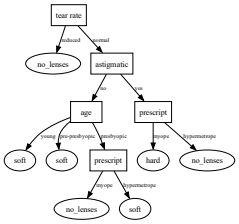
Training Size=14



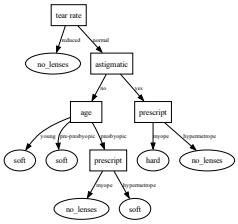
Training Size=16



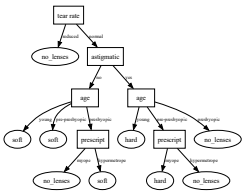
Training Size=17



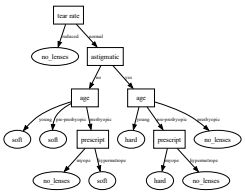
Training Size=18



Training Size=19



Training Size=20



Training Size=22