k-Nearest Neighbors Classifier

In this notebook, you will implement your own k-nearest neighbors (k-NN) algorithm for the classification problem. You are supposed to learn:

- How to prepare the dataset for "training" and testing of the model.
- How to implement k-nearest neighbors classification algorithm.
- How to evaluate the performance of your classifier.

Packages

Following packages is all you need. Do not import any additional packages!

- Pandas is a library providing easy-to-use data structures and data analysis tools.
- Numpy library provides support for large multi-dimensional arrays and matrices, along with functions to operate on these.

In [ ]:
```
import pandas as pd
import numpy as np
```

Problem

You are given a dataset `mushrooms.csv` with characteristics/attributes of mushrooms, and your task is to implement and evaluate a k-nearest neighbors classifier able to say whether a mushroom is poisonous or edible based on its attributes.

Dataset

The dataset of mushroom characteristics is freely available at Kaggle Datasets where you can find further information about the dataset. It consists of 8124 mushrooms characterized by 23 attributes (including the class). Following is the overview of attributes and values:

- class: edible=e, poisonous=p
- cap-shape: bell=b,conical=c,convex=x,flat=f, knobbed=k,sunken=s
- cap-surface: fibrous=f,grooves=g,scaly=y,smooth=s
- cap-color: brown=n,buff=b,cinnamon=c,gray=g,green=r,pink=p,purple=u,red=e,white=w,yellow=y
- bruises: bruises=t,no=f
- odor: almond=a,anise=l,creosote=c,fishy=y,foul=f,musty=m,none=n,pungent=p,spicy=s
- gill-attachment: attached=a,descending=d,free=f,notched=n
- gill-spacing: close=c,crowded=w,distant=d
- gill-size: broad=b,narrow=n

- gill-color: black=k,brown=n,buff=b,chocolate=h,gray=g, green=r,orange=o,pink=p,purple=u,red=e,white=w,yellow=y
- stalk-shape: enlarging=e,tapering=t
- stalk-root: bulbous=b,club=c,cup=u,equal=e,rhizomorphs=z,rooted=r,missing=?
- stalk-surface-above-ring: fibrous=f,scaly=y,silky=k,smooth=s
- stalk-surface-below-ring: fibrous=f,scaly=y,silky=k,smooth=s
- stalk-color-above-ring: brown=n,buff=b,cinnamon=c,gray=g,orange=o,pink=p,red=e,white=w,yellow=y
- stalk-color-below-ring: brown=n,buff=b,cinnamon=c,gray=g,orange=o,pink=p,red=e,white=w,yellow=y
- veil-type: partial=p,universal=u
- veil-color: brown=n,orange=o,white=w,yellow=y
- ring-number: none=n,one=o,two=t
- ring-type: cobwebby=c,evanescent=e,flaring=f,large=l,none=n,pendant=p,sheathing=s,zone=z
- spore-print-color: black=k,brown=n,buff=b,chocolate=h,green=r,orange=o,purple=u,white=w,yellow=y
- population: abundant=a,clustered=c,numerous=n,scattered=s,several=v,solitary=y
- habitat: grasses=g,leaves=l,meadows=m,paths=p,urban=u,waste=w,woods=d

Let's load the dataset into so called Pandas dataframe.

In [ ]:
```
mushrooms_df = pd.read_csv('mushrooms.csv')
```

Now we can take a closer look at the data.

In [ ]:
```
mushrooms_df
```

Out[ ]:

	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	stalk-surface-below-ring
0	p	x	s	n	t	p	f	c	n	k	...	s
1	e	x	s	y	t	a	f	c	b	k	...	s
2	e	b	s	w	t	l	f	c	b	n	...	s
3	p	x	y	w	t	p	f	c	n	n	...	s
4	e	x	s	g	f	n	f	w	b	k	...	s
...	...	...	...	...	...	...	...	...	...	...	...	...
8119	e	k	s	n	f	n	a	c	b	y	...	s
8120	e	x	s	n	f	n	a	c	b	y	...	s
8121	e	f	s	n	f	n	a	c	b	n	...	s
8122	p	k	y	n	f	y	f	c	n	b	...	k
8123	e	x	s	n	f	n	a	c	b	y	...	s

8124 rows × 23 columns

You can also print an overview of all attributes with the counts of unique values.

In [ ]:

```
mushrooms_df.describe().T
```

Out[ ]:

	count	unique	top	freq
class	8124	2	e	4208
cap-shape	8124	6	x	3656
cap-surface	8124	4	y	3244
cap-color	8124	10	n	2284
bruises	8124	2	f	4748
odor	8124	9	n	3528
gill-attachment	8124	2	f	7914
gill-spacing	8124	2	c	6812
gill-size	8124	2	b	5612
gill-color	8124	12	b	1728
stalk-shape	8124	2	t	4608
stalk-root	8124	5	b	3776
stalk-surface-above-ring	8124	4	s	5176
stalk-surface-below-ring	8124	4	s	4936
stalk-color-above-ring	8124	9	w	4464
stalk-color-below-ring	8124	9	w	4384
veil-type	8124	1	p	8124
veil-color	8124	4	w	7924
ring-number	8124	3	o	7488
ring-type	8124	5	p	3968
spore-print-color	8124	9	w	2388
population	8124	6	v	4040
habitat	8124	7	d	3148

The dataset is pretty much balanced. That's a good news for the evaluation.

Dataset Preprocessing

As our dataset consist of nominal/categorical values only, we will encode the strings into integers which will allow us to use similiraty measures such as Euclidean distance.

In [ ]:

```
def encode_labels(df):
    import sklearn.preprocessing
    encoder = {}
    for col in df.columns:
        le = sklearn.preprocessing.LabelEncoder()
        le.fit(df[col])
        df[col] = le.transform(df[col])
        encoder[col] = le
    return df, encoder
```

```
mushrooms_encoded_df, encoder = encode_labels(mushrooms_df)
```

In [ ]:
```
mushrooms_encoded_df
```

Out[ ]:

	class	cap-shape	cap-surface	cap-color	bruises	odor	gill-attachment	gill-spacing	gill-size	gill-color	...	stalk-surface-below-ring
0	1	5	2	4	1	6	1	0	1	4	...	2
1	0	5	2	9	1	0	1	0	0	4	...	2
2	0	0	2	8	1	3	1	0	0	5	...	2
3	1	5	3	8	1	6	1	0	1	5	...	2
4	0	5	2	3	0	5	1	1	0	4	...	2
...	...	...	...	...	...	...	...	...	...	...	...	...
8119	0	3	2	4	0	5	0	0	0	11	...	2
8120	0	5	2	4	0	5	0	0	0	11	...	2
8121	0	2	2	4	0	5	0	0	0	5	...	2
8122	1	3	3	4	0	8	1	0	1	0	...	1
8123	0	5	2	4	0	5	0	0	0	11	...	2

8124 rows × 23 columns

Dataset Splitting

Before we start with the implementation of our k-nearest neighbors algorithm we need to prepare our dataset for the "training" and testing.

First, we divide the dataset into attributes (often called features) and classes (often called targets). Keeping attributes and classes separately is a common practice in many implementations. This should simplify the implementation and make the code understandable.

In [ ]:
```
X_df = mushrooms_encoded_df.drop('class', axis=1)  # attributes
y_df = mushrooms_encoded_df['class']  # classes
X_array = X_df.to_numpy()
y_array = y_df.to_numpy()
```

And this is how it looks like.

In [ ]:
```
print('X =', X_array)
print('y =', y_array)
```

```
X = [[5 2 4 ... 2 3 5]
 [5 2 9 ... 3 2 1]
 [0 2 8 ... 3 2 3]
 ...
 [2 2 4 ... 0 1 2]
 [3 3 4 ... 7 4 2]
 [5 2 4 ... 4 1 2]]
y = [1 0 0 ... 0 1 0]
```

Next, we need to split the attributes and classes into training sets and test sets.

Exercise:

Implement the holdout splitting method with shuffling.

In [ ]:
```
def train_test_split(X, y, test_size=0.2):
    """
    Shuffles the dataset and splits it into training and test sets.

    :param X
        attributes
    :param y
        classes
    :param test_size
        float between 0.0 and 1.0 representing the proportion of the dataset t
    :return
        train-test splits (X-train, X-test, y-train, y-test)
    """
    # shuffling in unison
    data_length = len(X)
    shuffler = np.random.permutation(data_length)
    X = X[shuffler]
    y = y[shuffler]

    # preparing for splitting
    train_size = 1 - test_size
    test_length = round(data_length * test_size)
    train_length = round(data_length * train_size)

    # splitting
    X_train = X[:train_length]
    y_train = y[:train_length]
    X_test = X[-test_length:]
    y_test = y[-test_length:]

    return X_train, X_test, y_train, y_test
```

Let's split the dataset into training and validation/test set with 67:33 split.

In [ ]:
```
X_train, X_test, y_train, y_test = train_test_split(X_array, y_array, 0.33)
```

In [ ]:
```
print('X_train =', X_train)
print('y_train =', y_train)
print('X_test =', X_test)
print('y_test =', y_test)
```

```
X_train = [[5 3 4 ... 3 4 0]
 [5 3 2 ... 7 4 0]
 [2 3 3 ... 1 4 4]
 ...
 [2 0 4 ... 3 4 0]
 [2 3 3 ... 2 5 0]
 [5 2 2 ... 7 4 0]]
y_train = [0 1 1 ... 0 0 1]
X_test = [[2 2 3 ... 1 3 5]
 [2 2 4 ... 4 4 2]
 [2 0 4 ... 3 5 0]
 ...
 [2 2 4 ... 3 4 2]
 [5 0 4 ... 3 5 0]
 [3 2 4 ... 7 4 4]]
y_test = [1 0 0 ... 0 0 1]
```

A quick sanity check...

In [ ]:
```
assert len(X_train) == len(y_train)
assert len(y_train) == 5443
assert len(X_test) == len(y_test)
assert len(y_test) == 2681
```

Algorithm

The k-nearest neighbors algorithm doesn't require a training step. The class of an unseen sample is deduced by comparison with samples of known class.

Exercise:

Implement the k-nearest neighbors algorithm.

In [ ]:
```
from tqdm import tqdm


def euclidean_distance(x1, x2):
    Sum = sum([(x1[i] - x2[i])**2 for i in range(len(x1))])
    distance = Sum ** 0.5
    return distance


class Neighbor:
    def __init__(self, index, distance):
        self.index: int = int(index) # index in original array (X_pred)
        self.distance: float = distance

    def __lt__(self, other):
        return self.distance < other.distance

    def __str__(self):
        return str(round(self.distance))


def most_frequent(predictions):
    # positive for most ones, negative for most zeros (only works for binary y
```

```
    sign = sum([1 if prediction == 1 else -1 for prediction in predictions])
    return 1 if sign > 0 else 0
```

In [ ]:
```
def knn(X_true, y_true, X_pred, k=5):
    """
    k-nearest neighbors classifier.

    :param X_true
        attributes of the groung truth (training set)
    :param y_true
        classes of the groung truth (training set)
    :param X_pred
        attributes of samples to be classified
    :param k
        number of neighbors to use
    :return
        predicted classes
    """
    y_pred = []
    neighbor: Neighbor
    # loop through samples to be classified
    for i, sample in enumerate(tqdm(X_pred)):
        # compare sample to all other known data points (neighbors)
        knn_x = []
        for j, data in enumerate(X_true):
            if i == j: continue
            distance = euclidean_distance(sample, data)

            # add neighbor to knn_x if close enough
            neighbor = Neighbor(j, distance)
            knn_x.append(neighbor)
            knn_x.sort()
            if len(knn_x) > k:
                knn_x = knn_x[:-1]

        # classify the sample given its neighbors
        knn_y = []
        for neighbor in knn_x:
            knn_y.append(y_true[neighbor.index])
        prediction = most_frequent(knn_y)
        y_pred.append(prediction)
    return y_pred
```

In [ ]:
```
y_hat = knn(X_train, y_train, X_test, k=5)
```

```
100%|██████████| 2681/2681 [02:41<00:00, 16.65it/s]
```

First ten predictions of the test set.

In [ ]:
```
y_hat[:10]
```

Out[ ]:
```
[1, 0, 0, 1, 0, 0, 0, 1, 0, 0]
```

Evaluation

Now we would like to assess how well our classifier performs.

Exercise:

Implement a function for calculating the accuracy of your predictions given the ground truth and predictions.

In [ ]:
```
def evaluate(y_true, y_pred):
    """
    Function calculating the accuracy of the model on the given data.

    :param y_true
        true classes
    :paaram y
        predicted classes
    :return
        accuracy
    """
    data_points = len(y_true)
    hits = sum([1 if (true == pred) else 0 for (true, pred) in zip(y_true, y_p
    accuracy = hits / data_points
    return accuracy
```

In [ ]:
```
accuracy = evaluate(y_test, y_hat)
print('accuracy =', accuracy)
```

```
accuracy = 0.9981350242446848
```

How many items where misclassified?

In [ ]:
```
print('misclassified =', sum(abs(y_hat - y_test)))
```

```
misclassified = 5
```

How balanced is our test set?

In [ ]:
```
np.bincount(y_test)
```

Out[ ]:
```
array([1436, 1245])
```

If it's balanced, we don't have to be worried about objectivity of the accuracy metric.

Congratulations! At this point, hopefully, you have successufuly implemented a k-nearest neighbors algorithm able to classify unseen samples with high accuracy.

✌️