KNN

In statistics, the k-nearest neighbors algorithm (k-NN) is a non-parametric classification method first developed by Evelyn Fix and Joseph Hodges in 1951. The input consists of the k closest training examples in data set. The output depends on whether k-NN is used for classification or regression:

- In k-NN classification, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If k = 1, then the object is simply assigned to the class of that single nearest neighbor.
- In k-NN regression, the output is the property value for the object. This value is the average of the values of k nearest neighbors.

k-NN is a type of classification where the function is only approximated locally and all computation is deferred until function evaluation. Since this algorithm relies on distance for classification, if the features represent different physical units or come in vastly different scales then **normalizing the training data can improve its accuracy dramatically**.

For **high-dimensional data (e.g., with number of dimensions more than 10) dimension reduction is usually performed prior to applying the k-NN algorithm** in order to avoid the effects of the curse of dimensionality. Dimension reduction could be done using principal component analysis (PCA), linear discriminant analysis (LDA).

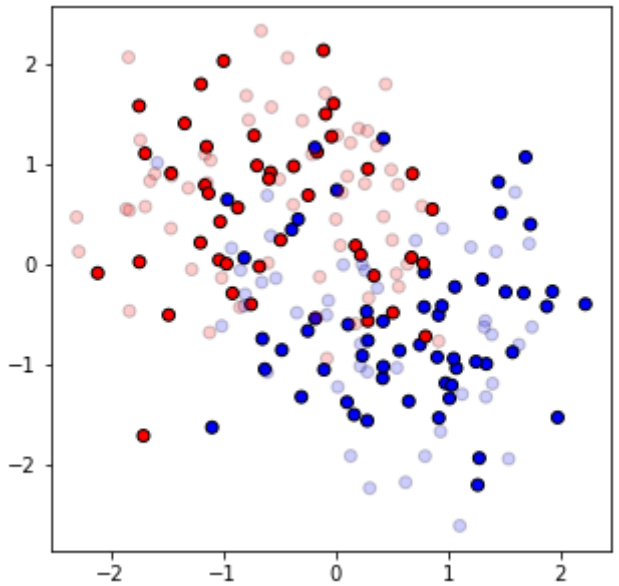
Table of Contents

In [ ]:
```
import ssl
ssl._create_default_https_context = ssl._create_unverified_context
```

In [ ]:
```
%matplotlib inline
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from matplotlib.colors import ListedColormap
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

k-NN - K-Nearest Neighbors for classification

In [ ]:
```
''' define the dataset '''
from sklearn.datasets import make_moons

# Define dataset
X, y = make_moons(n_samples=200, noise=0.4, random_state=0)
X = StandardScaler().fit_transform(X)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.5, random_state=42)

# Plot the training and testing points
plt.figure(figsize=(5, 5))
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
plt.scatter(X_train[:, 0], X_train[:, 1],
            c=y_train, cmap=cm_bright, edgecolors='k')
plt.scatter(X_test[:, 0], X_test[:, 1],
            c=y_test, cmap=cm_bright, edgecolors='k',
            alpha=0.2)
plt.show()
```

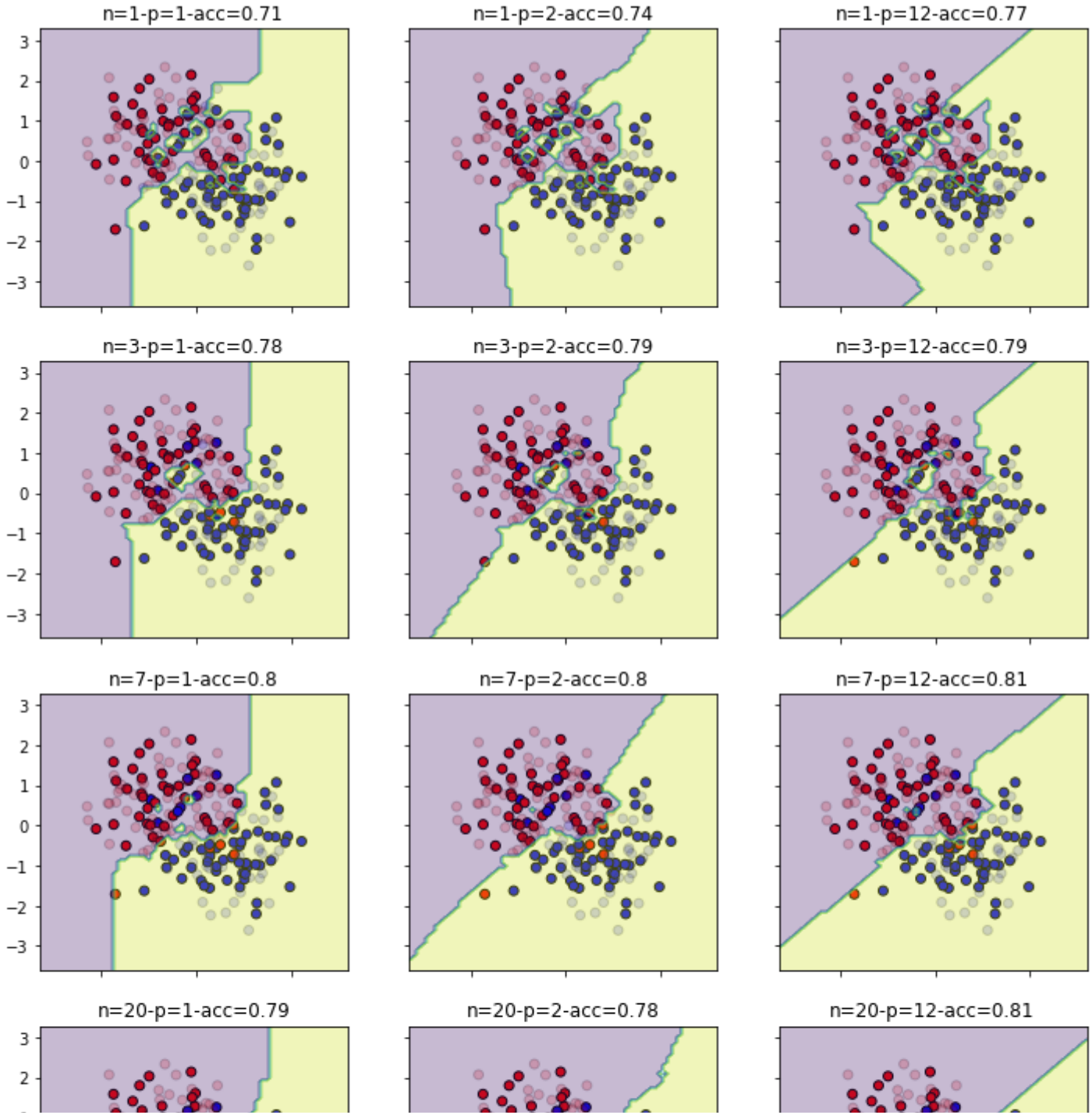


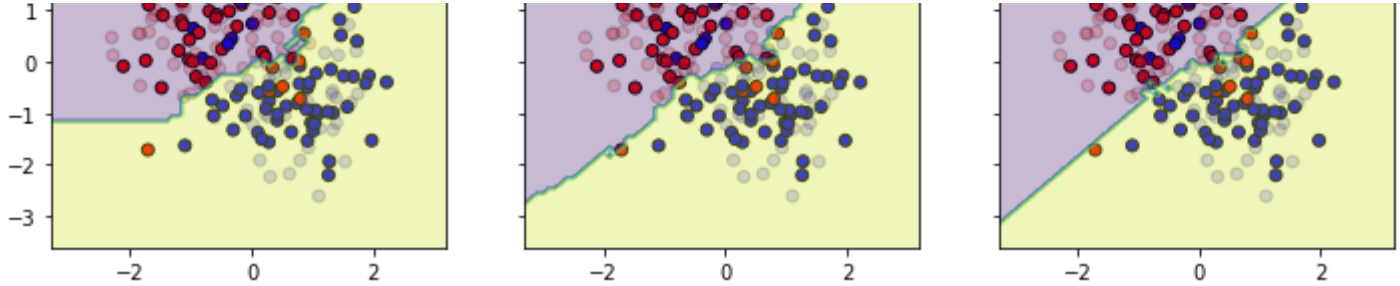
In [ ]:
```
''' Build and use classifier '''
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

clf = KNeighborsClassifier(n_neighbors=7, p=2)
# p: power parameter for the Minkowski metric.
# p = 1 --> manhattan_distance (l1)
# p = 2 --> euclidean_distance (l2)

clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)

# With 2 classes of the same size, accuracy can be used
accuracy_score(y_test, y_pred)
```

Out[ ]:
```
0.8
```

In [ ]:
```
''' Plot the decision regions with different neighbors and distance '''
# change the number of neighbors
n_neighbors = [1, 3, 7, 20]
n_power = [1, 2, 12]
```

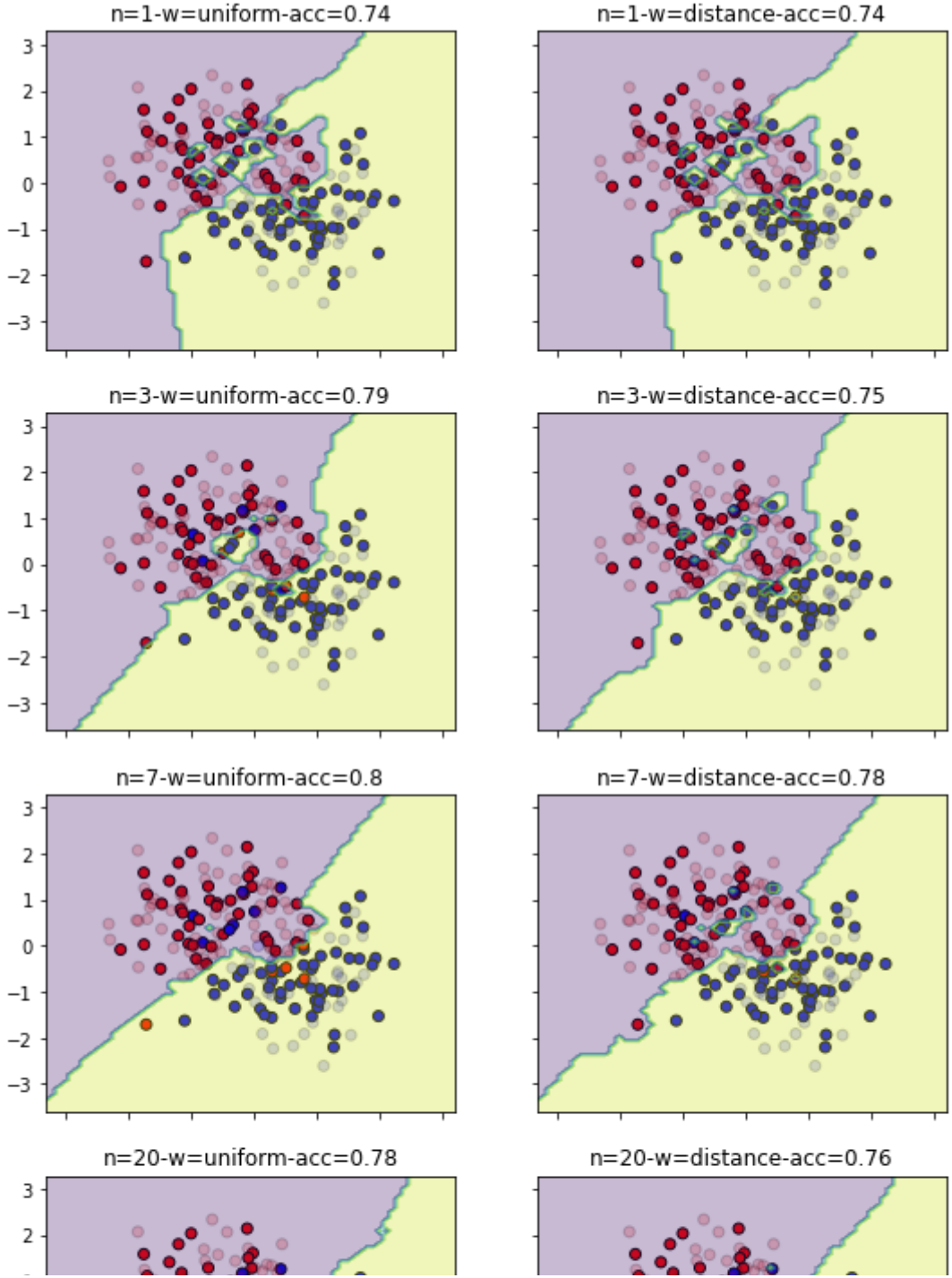
```
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

fig, ax = plt.subplots(len(n_neighbors), len(n_power), sharex='col', sharey='row', figsize=((len(n_power)+1)*3, (len(
cm_bright = ListedColormap(['#FF0000', '#0000FF'])

for i, neighbors in enumerate(n_neighbors):
    for j, power in enumerate(n_power):
        clf = KNeighborsClassifier(n_neighbors=neighbors, p=power)
        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)
        ax[i, j].scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright, edgecolors='k')
        ax[i, j].scatter(X_test[:, 0], X_test[:, 1], c=y_pred, cmap=cm_bright, edgecolors='k', alpha=0.2)

        Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)
        ax[i, j].contourf(xx, yy, Z, alpha=0.3)
        ax[i, j].set_title("n="+str(neighbors)+"-p="+str(power)+"-acc="+str(accuracy_score(y_test, y_pred)))
plt.show()
```

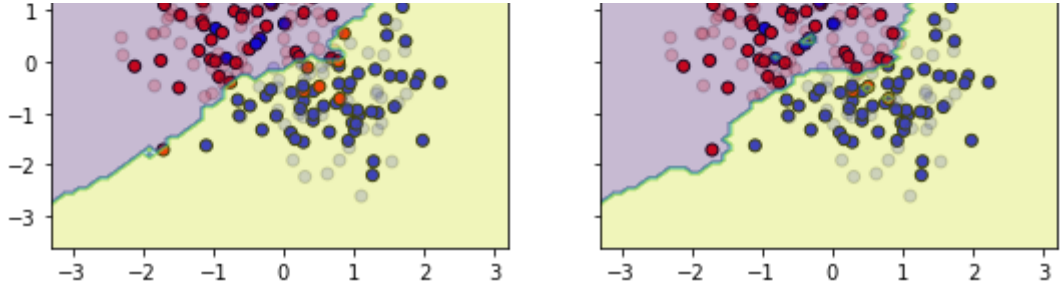

n=1-p=1-acc=0.71
n=1-p=2-acc=0.74
n=1-p=12-acc=0.77
n=3-p=1-acc=0.78
n=3-p=2-acc=0.79
n=3-p=12-acc=0.79
n=7-p=1-acc=0.8
n=7-p=2-acc=0.8
n=7-p=12-acc=0.81
n=20-p=1-acc=0.79
n=20-p=2-acc=0.78
n=20-p=12-acc=0.81



In [ ]:

```
''' Plot the decision regions with different neighbors and weigthed '''
# change the number of neighbors
n_neighbors = [1, 3, 7, 20]
power = 2
n_weights = ['uniform', 'distance']

x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))

fig, ax = plt.subplots(len(n_neighbors), len(n_weights), sharex='col', sharey='row', figsize=((len(n_weights)+1)*3, (
cm_bright = ListedColormap(['#FF0000', '#0000FF'])

for i, neighbors in enumerate(n_neighbors):
    for j, weights in enumerate(n_weights):
        clf = KNeighborsClassifier(n_neighbors=neighbors, p=power, weights=weights)
        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)
        ax[i, j].scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright, edgecolors='k')
        ax[i, j].scatter(X_test[:, 0], X_test[:, 1], c=y_pred, cmap=cm_bright, edgecolors='k', alpha=0.2)

        Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])
        Z = Z.reshape(xx.shape)
        ax[i, j].contourf(xx, yy, Z, alpha=0.3)
        ax[i, j].set_title("n="+str(neighbors)+"-w="+str(weights)+"-acc="+str(accuracy_score(y_test, y_pred)))
plt.show()
```

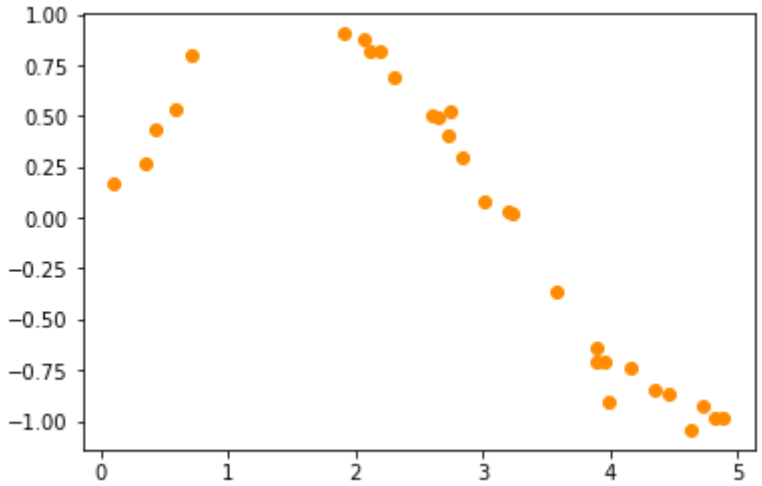

n=1-w=uniform-acc=0.74
n=1-w=distance-acc=0.74
n=3-w=uniform-acc=0.79
n=3-w=distance-acc=0.75
n=7-w=uniform-acc=0.8
n=7-w=distance-acc=0.78
n=20-w=uniform-acc=0.78
n=20-w=distance-acc=0.76



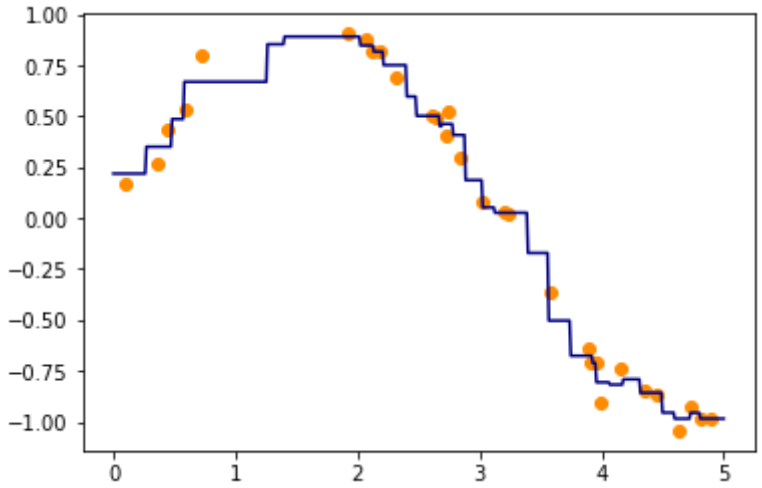
k-NN - K-Nearest Neighbors for regression

In [ ]:

```
from sklearn.datasets import make_regression

# Define dataset
nb_items = 30
np.random.seed(0)
X_train = np.sort(5 * np.random.rand(nb_items, 1), axis=0)
y_train = np.sin(X_train).ravel() + 0.3*(0.5 - np.random.rand(nb_items))*(np.random.rand(nb_items)<0.7)

X_test = np.linspace(0, 5, 500)[:, np.newaxis]

# Plot the training and testing points
plt.scatter(X_train, y_train, color='darkorange', label='data')
plt.show()
```

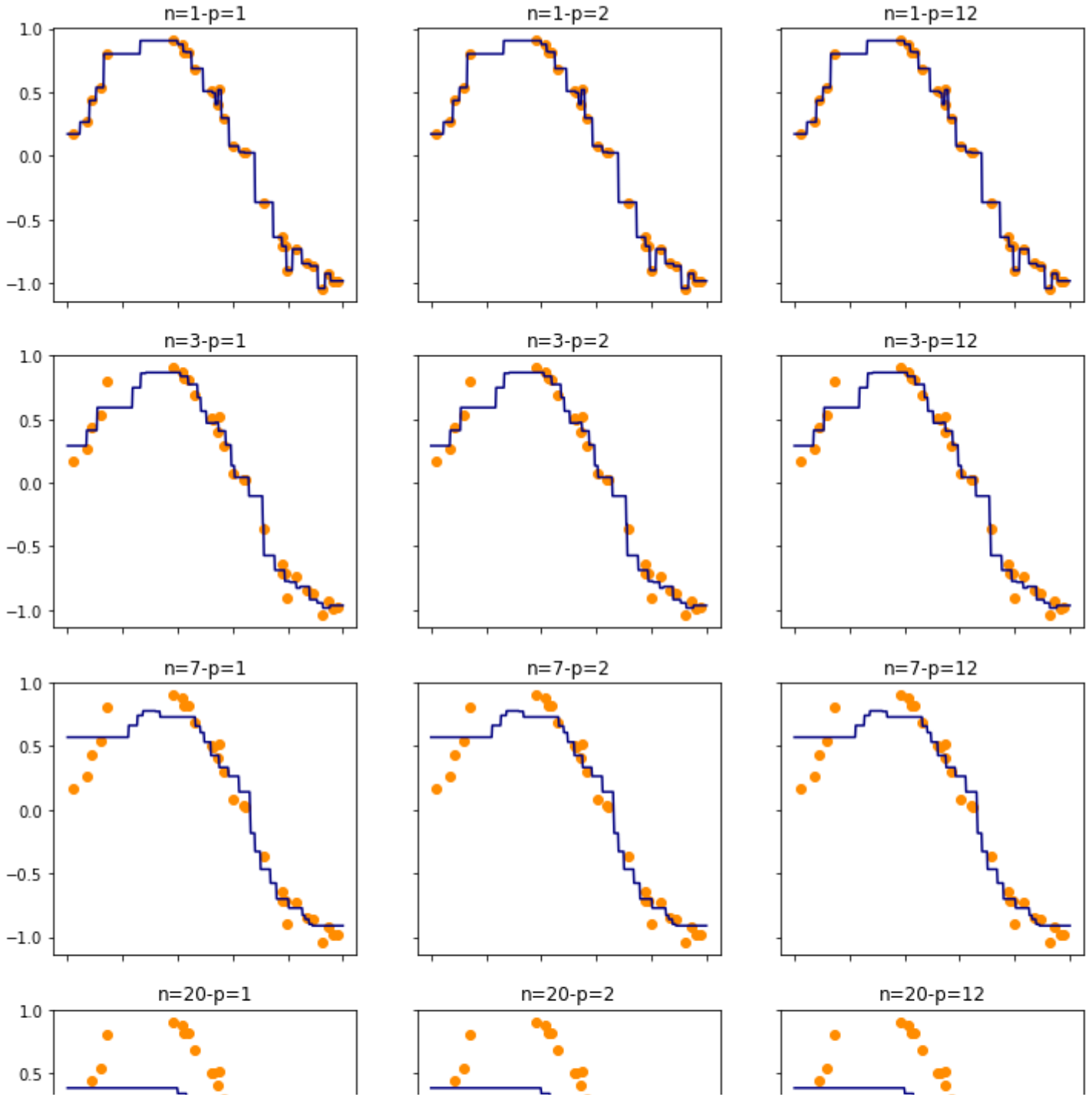


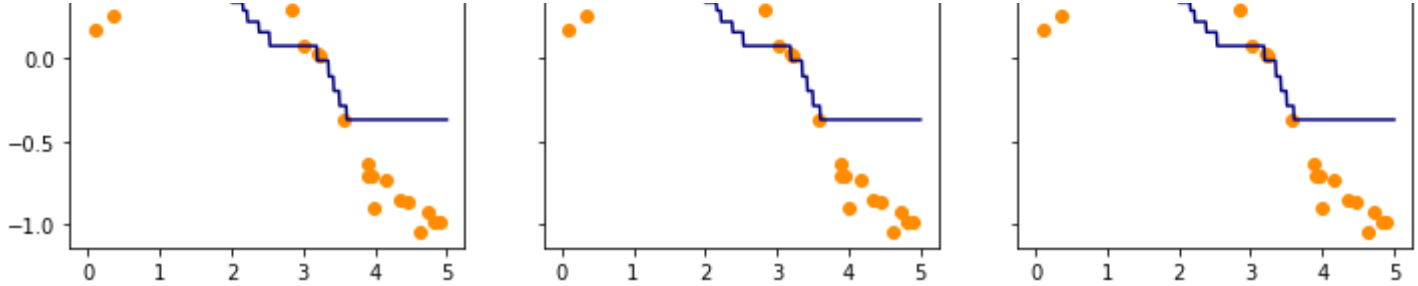
In [ ]:
```
from sklearn.neighbors import KNeighborsRegressor

weights='uniform'
n_neighbors = 2
knn = KNeighborsRegressor(n_neighbors, weights=weights, p=2)
y_ = knn.fit(X_train, y_train).predict(X_test)
y_pred = knn.predict(X_test)

plt.scatter(X_train, y_train, color='darkorange', label='data')
plt.plot(X_test, y_pred, color='navy', label='prediction')
plt.show()
```



In [ ]:
```
n_neighbors = [1, 3, 7, 20]
n_power = [1, 2, 12]

fig, ax = plt.subplots(len(n_neighbors), len(n_power), sharex='col', sharey='row', figsize=((len(n_power)+1)*3, (len(
cm_bright = ListedColormap(['#FF0000', '#0000FF'])

for i, neighbors in enumerate(n_neighbors):
    for j, power in enumerate(n_power):
        clf = KNeighborsRegressor(n_neighbors=neighbors, p=power)
        clf.fit(X_train, y_train)
        y_pred = clf.predict(X_test)
        ax[i, j].scatter(X_train, y_train, color='darkorange', label='data')
        ax[i, j].plot(X_test, y_pred, color='navy', label='prediction')
```

```
        ax[i, j].set_title("n="+str(neighbors)+"-p="+str(power))
plt.show()
```


n=1-p=1
n=1-p=2
n=1-p=12
n=3-p=1
n=3-p=2
n=3-p=12
n=7-p=1
n=7-p=2
n=7-p=12
n=20-p=1
n=20-p=2
n=20-p=12
1.0
0.5
0.0
−0.5
−1.0



kNN from scratch - force brut

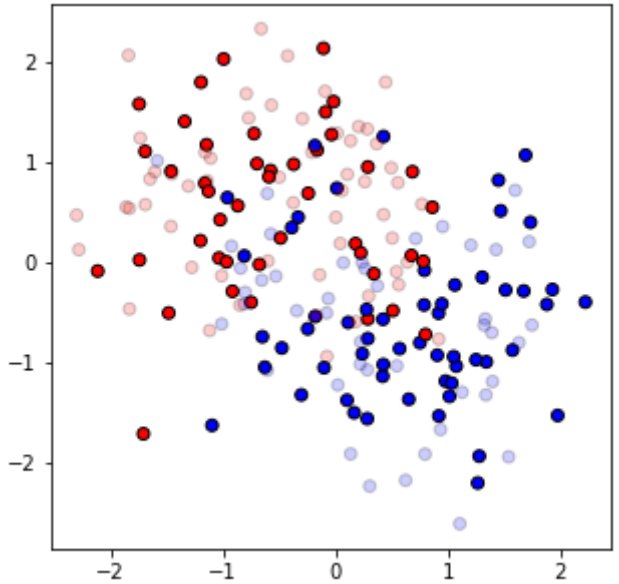
The **naive version** of the algorithm is easy to implement by calculating the distances between the test example and all stored examples, but it is computationally intensive for large training sets. **This is what we will do here.**

Using an **approximate nearest neighbor** search algorithm makes the k-NN faster to compute, even for large data sets. Many nearest neighbor search algorithms have been proposed over the years; they generally seek to reduce the number of distance evaluations actually performed.

Define the dataset

In [ ]:
```
# Define dataset
X, y = make_moons(n_samples=200, noise=0.4, random_state=0)
X = StandardScaler().fit_transform(X)
y = y.reshape(-1, 1)

dataset = np.concatenate((X, y), axis=1)
train, test = train_test_split(dataset, test_size=.5, random_state=42)
```

In [ ]:
```
# Plot the training and testing points
plt.figure(figsize=(5, 5))
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
plt.scatter(train[:, 0], train[:, 1], c=train[:, -1], cmap=cm_bright, edgecolors='k')
plt.scatter(test[:, 0], test[:, 1], c=test[:, -1], cmap=cm_bright, edgecolors='k', alpha=0.2)
plt.show()
```



Step 1: Calculate Euclidean Distance

In [ ]:

```
# calculate the Euclidean distance between two vectors
from math import sqrt

def euclidean_distance(row1, row2):
    distance = 0.0
    for i in range(len(row1)-1):
        distance += (row1[i] - row2[i])**2
    return sqrt(distance)
```

In [ ]:

```
# print distance between the first element of test set to the 10th first element of train
for row in train[:10]:
    distance = euclidean_distance(test[0,:-1], row[:-1])
    print(distance)
```

```
0.00886293919542891
1.045397591464741
1.350070676281334
0.6573564294853871
1.8215091152800857
2.031562341242025
2.2493859974292336
0.4132095942070787
2.328375325366197
1.3205541336786386
```

Step 2: Get Nearest Neighbors

Neighbors for a new piece of data in the dataset are the k closest instances, as defined by our distance measure.

To locate the neighbors for a new piece of data within a dataset we must first calculate the distance between each record in the dataset to the new piece of data. We can do this using our distance function prepared above.

Once distances are calculated, we must sort all of the records in the training dataset by their distance to the new data. We can then select the top k to return as the most similar neighbors.

We can do this by keeping track of the distance for each record in the dataset as a tuple, sort the list of tuples by the distance (in descending order) and then retrieve the neighbors.

In [ ]:

```
# Locate the most similar neighbors
def get_neighbors(train, test_row, num_neighbors):
    distances = list()
    for train_row in train:
        dist = euclidean_distance(test_row, train_row)
        distances.append((train_row, dist))
    distances.sort(key=lambda tup: tup[1])
    neighbors = list()
    for i in range(num_neighbors):
        neighbors.append(distances[i][0])
    return neighbors
```

In [ ]:

```
# Look at the 3 nearest neighbors from the first element of test set
neighbors = get_neighbors(train, test[0], 3)
```

```
for neighbor in neighbors:
    print(neighbor)
```

```
[ 1.51479055 -0.27369259  1.        ]
[ 1.57677596 -0.87449506  1.        ]
[ 0.9482417  -0.41261174  1.        ]
```

Step 3: Make Predictions

The most similar neighbors collected from the training dataset can be used to make predictions.

In the case of classification, we can return the most represented class among the neighbors.

We can achieve this by performing the max() function on the list of output values from the neighbors. Given a list of class values observed in the neighbors, the max() function takes a set of unique class values and calls the count on the list of class values for each class value in the set.

Below is the function named predict_classification() that implements this.

In [ ]:
```
# Make a classification prediction with neighbors
def predict_classification(train, test_row, num_neighbors):
    neighbors = get_neighbors(train, test_row, num_neighbors)
    output_values = [row[-1] for row in neighbors]
    prediction = max(set(output_values), key=output_values.count)
    return prediction
```

In [ ]:
```
for i in range(10):
    prediction = predict_classification(train, test[i], 3)
    print('Expected %d, Got %d.' % (test[i, -1], prediction))
```

```
Expected 1, Got 1.
Expected 0, Got 0.
Expected 0, Got 0.
Expected 0, Got 0.
Expected 1, Got 1.
Expected 1, Got 1.
Expected 0, Got 0.
Expected 0, Got 0.
Expected 1, Got 1.
Expected 0, Got 0.
```

Cost of these approach

- n: number of points in the training dataset
- d: data dimensionality
- k: number of neighbors that we consider for voting

- Training time complexity: O(1)
- Prediction time complexity: O(k *n* d)

The lab for today: use KNN for missing values imputation

The horse colic dataset describes medical characteristics of horses with colic and whether they lived or died.

The data is supplied in two files. The first contains 300 horse colic cases that should be used for training data while the second contains 68 cases that can be used for testing the performance of your method.

The variable that we have tended to try and predict is:

- V23: what eventually happened to the horse? (1 = lived, 2 = died, 3 = was euthanized)
- V24, surgical lesion? retrospectively, was the problem (lesion) surgical? All cases are either operated upon or autopsied so that this value and the lesion type are always known (1 = Yes, 2 = No)
- V25, V26, V27: type of lesion coded on 4 digits
- V28: is pathology data present for this case? (1 = Yes, 2 = No)

The sample contains approx. 30% missing values (indicated by a zero in the data provided). You will need to deal with all types of variables (continuous, discrete, and nominal) as well as the missing values in your method.

The dataset has many missing values for many of the columns where each missing value is marked with a question mark character (“?”).

In [ ]:

```
# load dataset
url = "https://www.i3s.unice.fr/~riveill/dataset/"

def read_dataset(url, name):
    dataframe = pd.read_csv(url+name, header=None, sep=";",
                            na_values='?', error_bad_lines=False)
    mapper = {}
```

```
    for old, new in zip(dataframe.columns, ["V"+str(i+1) for i in dataframe.columns]):
        mapper[old] = new
    dataframe = dataframe.rename(mapper, axis=1)

    dataframe['V1'] = (dataframe['V1']==1.0) # Surgery ?
    dataframe['V2'] = (dataframe['V2']==1) # Adult ?
    dataframe['V3'] = dataframe['V3'].astype('category') # Hospital number
    # dataframe['V4'], rectal temperature
    # dataframe['V5'], heart pulse
    # dataframe['V6'], respirtory rate
    dataframe['V7'] = dataframe['V7'].astype('category') # temperature of extremities
    dataframe['V8'] = dataframe['V8'].astype('category') # peripheral pulse
    dataframe['V9'] = dataframe['V9'].astype('category') # mucous membranes
    dataframe['V10'] = dataframe['V10'].astype('category') # capillary refill time
    dataframe['V11'] = dataframe['V11'].astype('category') # pain
    dataframe['V12'] = dataframe['V12'].astype('category') # peristalsis
    dataframe['V13'] = dataframe['V13'].astype('category') # abdominal distension
    dataframe['V14'] = dataframe['V14'].astype('category') # nasogastric tube
    dataframe['V15'] = dataframe['V15'].astype('category') # nasogastric reflux
    # dataframe['V16'], nasogastric reflux PH
    dataframe['V17'] = dataframe['V17'].astype('category') # rectal examination - feces
    dataframe['V18'] = dataframe['V18'].astype('category') # abdomen
    # dataframe['V19'], packed cell volume
    # dataframe['V20'], total protein
    dataframe['V21'] = dataframe['V21'].astype('category') # abdominocentesis appearance
    # dataframe['V22'], abdomcentesis total protein

    dataframe['V23'] = dataframe['V23'].astype('category')
    dataframe['V24'] = (dataframe['V24']==1.0)
    dataframe[['V25', 'V26', 'V27']] = dataframe[['V25', 'V26', 'V27']].astype('category')
    dataframe['V28'] = (dataframe['V28']==1.0)

    return dataframe

train = read_dataset(url, "horse-colic-train.csv")
test = read_dataset(url, "horse-colic-test.csv")

train.head()
```

```
/tmp/ipykernel_13201/1427134355.py:42: FutureWarning: The error_bad_lines argument has been deprecated and will be re
moved in a future version.


  train = read_dataset(url, "horse-colic-train.csv")
/tmp/ipykernel_13201/1427134355.py:43: FutureWarning: The error_bad_lines argument has been deprecated and will be re
moved in a future version.


  test = read_dataset(url, "horse-colic-test.csv")
```

Out[ ]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	...	V19	V20	V21	V22	V23	V24	V25	V26	V27	V28
0	False	True	530101	38.5	66.0	28.0	3.0	3.0	NaN	2.0	...	45.0	8.4	NaN	NaN	2.0	False	11300	0	0	False
1	True	True	534817	39.2	88.0	20.0	NaN	NaN	4.0	1.0	...	50.0	85.0	2.0	2.0	3.0	False	2208	0	0	False
2	False	True	530334	38.3	40.0	24.0	1.0	1.0	3.0	1.0	...	33.0	6.7	NaN	NaN	1.0	False	0	0	0	True
3	True	False	5290409	39.1	164.0	84.0	4.0	1.0	6.0	2.0	...	48.0	7.2	3.0	5.3	2.0	True	2208	0	0	True
4	False	True	530255	37.3	104.0	35.0	NaN	NaN	6.0	2.0	...	74.0	7.4	NaN	NaN	2.0	False	4300	0	0	False

5 rows × 28 columns

In [ ]:

```
train.dtypes
```

Out[ ]:
```
V1         bool
V2         bool
V3     category
V4      float64
V5      float64
V6      float64
V7     category
V8     category
V9     category
V10    category
V11    category
V12    category
V13    category
V14    category
V15    category
V16     float64
V17    category
V18    category
V19     float64
V20     float64
V21    category
V22     float64
V23    category
V24        bool
V25    category
V26    category
V27    category
V28        bool
dtype: object
```

In [ ]:
```
target = ['V23']
remove = ['V24', 'V25', 'V26', 'V27', 'V28']
features = [c for c in test.columns if c not in target + remove]
types = [train[t].dtype for t in features]

# Drop row if target value is missing
train = train.dropna(axis=0, subset=target)
X_train = train[features]
y_train = train[target]

test = test.dropna(axis=0, subset=target)
X_test = test[features]
y_test = test[target]
```

In [ ]:
```
X_train.head()
```

Out[ ]:

	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	...	V13	V14	V15	V16	V17	V18	V19	V20	V21	V22
0	False	True	530101	38.5	66.0	28.0	3.0	3.0	NaN	2.0	...	4.0	NaN	NaN	NaN	3.0	5.0	45.0	8.4	NaN	NaN
1	True	True	534817	39.2	88.0	20.0	NaN	NaN	4.0	1.0	...	2.0	NaN	NaN	NaN	4.0	2.0	50.0	85.0	2.0	2.0
2	False	True	530334	38.3	40.0	24.0	1.0	1.0	3.0	1.0	...	1.0	NaN	NaN	NaN	1.0	1.0	33.0	6.7	NaN	NaN
3	True	False	5290409	39.1	164.0	84.0	4.0	1.0	6.0	2.0	...	4.0	1.0	2.0	5.0	3.0	NaN	48.0	7.2	3.0	5.3
4	False	True	530255	37.3	104.0	35.0	NaN	NaN	6.0	2.0	...	NaN	NaN	NaN	NaN	NaN	NaN	74.0	7.4	NaN	NaN

5 rows × 22 columns

In [ ]:
```
y_train.head()
```

Out[ ]:

	V23
0	2.0
1	3.0
2	1.0
3	2.0
4	2.0

In [ ]:
```
# summarize the number of rows with missing values for each column
for c in X_train.columns:
    # count number of rows with missing values
    n_miss = X_train[[c]].isnull().sum()
    perc = n_miss / X_train.shape[0] * 100
    print('> %s, Missing: %d (%.1f%%)' % (c, n_miss, perc))
```

```
> V1, Missing: 0 (0.0%)
> V2, Missing: 0 (0.0%)
> V3, Missing: 0 (0.0%)
> V4, Missing: 60 (20.1%)
> V5, Missing: 24 (8.0%)
> V6, Missing: 58 (19.4%)
> V7, Missing: 56 (18.7%)
> V8, Missing: 69 (23.1%)
> V9, Missing: 47 (15.7%)
> V10, Missing: 32 (10.7%)
> V11, Missing: 55 (18.4%)
> V12, Missing: 44 (14.7%)
> V13, Missing: 56 (18.7%)
> V14, Missing: 104 (34.8%)
> V15, Missing: 106 (35.5%)
> V16, Missing: 246 (82.3%)
> V17, Missing: 102 (34.1%)
> V18, Missing: 118 (39.5%)
> V19, Missing: 29 (9.7%)
> V20, Missing: 33 (11.0%)
> V21, Missing: 165 (55.2%)
> V22, Missing: 198 (66.2%)
```

In [ ]:
```
y_train.isnull().sum()
```

Out[ ]:
```
V23    0
dtype: int64
```

Build model

Build a pipeline that:

- imputes the missing values (attention the strategy can be different depending on whether it is numerical or categorical data)
- normalizes / encodes the data (also the work to be done is different and may depend on the next step)
- predicts with a KNN based model (find the right hyper-parameters).

In [ ]:
```
from sklearn.impute import KNNImputer, SimpleImputer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
from sklearn.compose import ColumnTransformer
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.neighbors import KNeighborsClassifier
```

```
from sklearn.model_selection import GridSearchCV

# find col per dtype
categorical_col = np.argwhere((X_train.dtypes.values == "category") | (X_train.dtypes.values == "bool")).reshape(-1)
numerical_col = np.argwhere(X_train.dtypes.values == "float64").reshape(-1)

# define imputers
categorical_imputer = SimpleImputer(strategy="most_frequent")
numerical_imputer = KNNImputer()

imputer = ColumnTransformer(transformers=[
    ("categorical_imputer", categorical_imputer, categorical_col),
    ("numerical_imputer", numerical_imputer, numerical_col),
])

# define preprocessors
categorical_preproc = OneHotEncoder(handle_unknown="ignore", sparse=False)
numerical_preproc = StandardScaler()

preproc = ColumnTransformer(transformers=[
    ("categorical_preproc", categorical_preproc, categorical_col),
    ("numerical_preproc", numerical_preproc, numerical_col),
])

# make pipeline
preproc_pipe = make_pipeline(imputer, preproc)
preproc_X_train = preproc_pipe.fit_transform(X_train)
```

In [ ]:

```
# define classifier
clf = KNeighborsClassifier()

# grid search
param_grid = {
    "algorithm": ["ball_tree", "kd_tree", "brute"],
    "leaf_size": [20, 30, 40, 50, 60],
    "p": [1, 2, 5, 10],
}
grid = GridSearchCV(clf, param_grid, verbose=1, n_jobs=-1)
grid.fit(preproc_X_train, y_train.values.ravel())
grid.best_score_, grid.best_params_
```

```
Fitting 5 folds for each of 60 candidates, totalling 300 fits
```

Out[ ]:
```
(0.6520338983050848, {'algorithm': 'brute', 'leaf_size': 20, 'p': 1})
```

In [ ]:
```
# make complete pipe with imputer, preprocessor and classifier
# the classifier is set with the best parameters from grid search
pipe = make_pipeline(imputer, preproc, clf.set_params(**grid.best_params_))
pipe.fit(X_train, y_train.values.ravel())
pipe
```

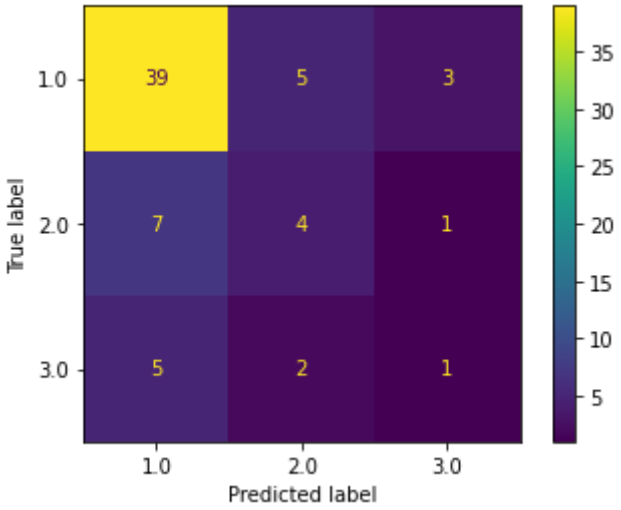
Out[ ]:
```
Pipeline(steps=[('columntransformer-1',
                 ColumnTransformer(transformers=[('categorical_imputer',
                                                  SimpleImputer(strategy='most_frequent'),
                                                  array([ 0,  1,  2,  6,  7,  8,  9, 10, 11, 12, 13, 14, 16, 17, 2
0])),
                                                 ('numerical_imputer',
                                                  KNNImputer(),
                                                  array([ 3,  4,  5, 15, 18, 19, 21]))])),
                ('columntransformer-2',
                 ColumnTransformer(transformers=[('categorical_preproc',
                                                  OneHotEncoder(handle_unknown='ignore',
                                                                sparse=False),
                                                  array([ 0,  1,  2,  6,  7,  8,  9, 10, 11, 12, 13, 14, 16, 17, 2
0])),
                                                 ('numerical_preproc',
                                                  StandardScaler(),
                                                  array([ 3,  4,  5, 15, 18, 19, 21]))])),
                ('kneighborsclassifier',
                 KNeighborsClassifier(algorithm='brute', leaf_size=20, p=1))])
```

Evaluate your model

- Plot the confusion matrix
- print the classification report
- find the previous values from the confusion matrix (put the formulas in a commented cell)

In [ ]:
```
from sklearn.metrics import confusion_matrix, classification_report, ConfusionMatrixDisplay
y_pred = pipe.predict(X_test)
ConfusionMatrixDisplay.from_predictions(y_test, y_pred)
```

Out[ ]:
```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f89b4f61340>
```



In [ ]:

```
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

         1.0       0.76      0.83      0.80        47
         2.0       0.36      0.33      0.35        12
         3.0       0.20      0.12      0.15         8

    accuracy                           0.66        67
   macro avg       0.44      0.43      0.43        67
weighted avg       0.63      0.66      0.64        67
```

In [ ]:

```
cm = confusion_matrix(y_test, y_pred)
print(f"Confusion matrix{cm}")


# precision
print("\n--> precision: TP/PP")
precision = lambda i: round(cm[i,i] / cm[:,i].sum(), 2)
for i in range(3):
    print(f"precision for class {i}: {precision(i)}")

# recall
print("\n--> recall: TP/PP")
```

```
recall = lambda i: round(cm[i,i] / cm[i].sum(), 2)
for i in range(3):
    print(f"recall for class {i}: {recall(i)}")

# f1-score
print("\n--> f1-score: 2 * (precision * recall) / (precision + recall)")
f_one = lambda i: round(2 * (precision(i) * recall(i)) / (precision(i) + recall(i)), 2)
for i in range(3):
    print(f"f1-score for class {i}: {f_one(i)}")

# support
print("\n--> support: cardinal of class `i`")
support = lambda i: cm[i].sum()
for i in range(3):
    print(f"support for class {i}: {support(i)}")
print(f"total support: {cm.sum()}")

# accuracy
print("\n--> accuracy: TP/(nb observation)")
accuracy = lambda i: round(sum([cm[i,i] for i in range(3)]) / cm.sum(), 2)
print(f"accuracy: {accuracy(i)}")

# macro metrics
print("\n--> macro metrics: unweighted average of class metrics")
macro = lambda metric : round(np.mean([metric(i) for i in range(3)]), 2)
print(f"macro precision: {macro(precision)}")
print(f"macro recall: {macro(recall)}")
print(f"macro f_one: {macro(f_one)}")

# weighted metrics
print("\n--> weighted metrics: average of class metrics, weighted by class cardinal")
weighted = lambda metric : round(np.average([metric(i) for i in range(3)], weights=[support(i) for i in range(3)]), 2
print(f"weighted precision: {weighted(precision)}")
print(f"weighted recall: {weighted(recall)}")
print(f"weighted f_one: {weighted(f_one)}")
```

```
[[39  5  3]
 [ 7  4  1]
 [ 5  2  1]]

--> precision: TP/PP
precision for class 0: 0.76
precision for class 1: 0.36
precision for class 2: 0.2

--> recall: TP/PP
recall for class 0: 0.83
recall for class 1: 0.33
recall for class 2: 0.12

--> f1-score: 2 * (precision * recall) / (precision + recall)
f1-score for class 0: 0.79
f1-score for class 1: 0.34
f1-score for class 2: 0.15

--> support: cardinal of class `i`
support for class 0: 47
support for class 1: 12
support for class 2: 8
total support: 67

--> accuracy: TP/(nb observation)
accuracy: 0.66

--> macro metrics: unweighted average of class metrics
macro precision: 0.44
macro recall: 0.43
macro f_one: 0.43

--> weighted metrics: average of class metrics, weighted by class cardinal
weighted precision: 0.62
weighted recall: 0.66
weighted f_one: 0.63
```

Approximate Nearest Neighbors

Try to understand ANN

- KNN (K-Nearest Neighbors) is Dead!
- Comprehensive Guide To Approximate Nearest Neighbors Algorithms
- Approximate Nearest Neighbor Search in High Dimensions

In [ ]: