Open in Colab

Open in Colab

Assignment 0 - KNN and linear regression practice

Ben Oliver Netid: 790890107

Names of students you worked with on this assignment: N/A

Note: this assignment falls under collaboration Mode 1: Collaborative Assignment – Collaboration Required! Please refer to the syllabus on Canvas for additional information.

Instructions for all assignments can be found here, and is also linked to from the course syllabus.

This is a practice assignment to get you familiar with the assignment format, environment set up, and

Learning Objectives

- Become familiar with the assignment format
- Work on environment set up (Colab is totally fine to start)
- Figure out how to output the notebook and submit it to canvas.
- Practice end-to-end ML processes using KNN
- K-Nearest Neighbours : Our goal is to implement a KNN classifier and apply it to classify the Iris dataset.

KNN

Data processing and visualization

We always have this set of imports at the beginning of our notebooks and set the random seed.

In [11]:
```
import numpy as np
#the output of plotting commands is displayed inline within frontends
%matplotlib inline
```

```
import matplotlib.pyplot as plt

#it is important to set the seed for reproducibility as it initializes the r
np.random.seed(1234)
```

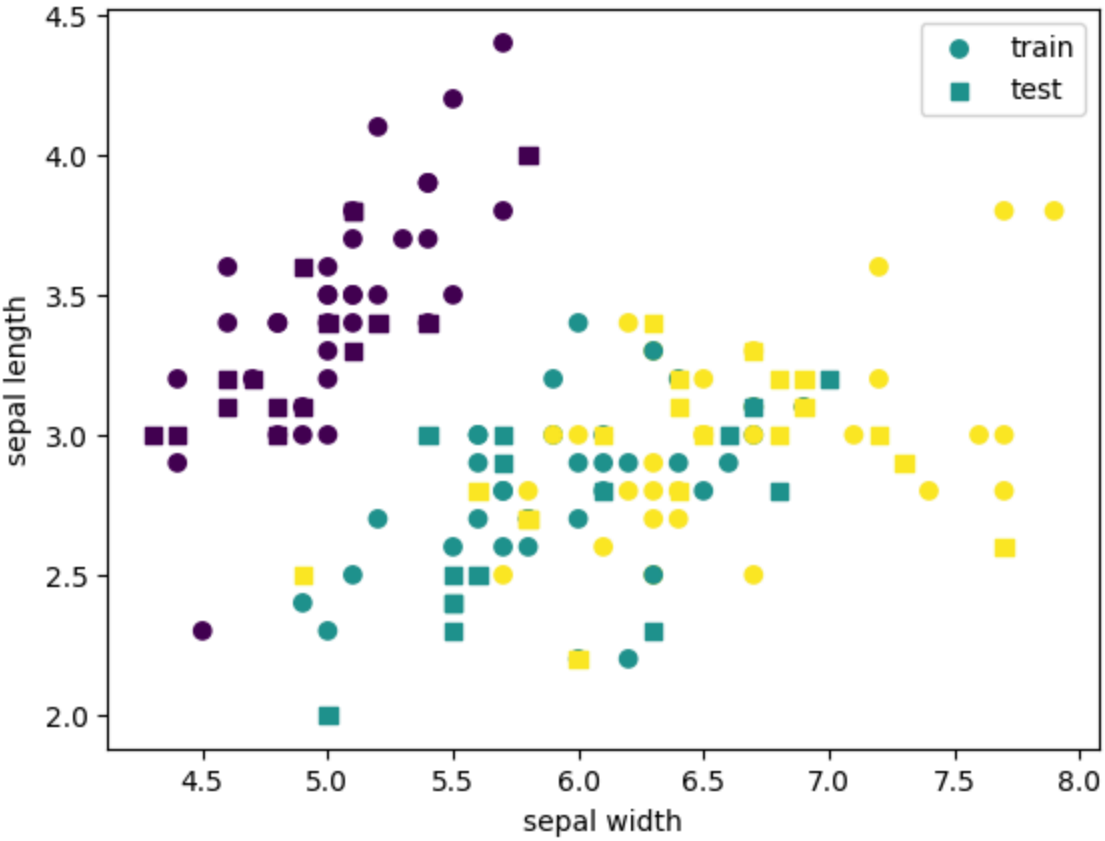
We conveniently load the dataset from the sklearn collection of datasets. To start, we will use the Iris dataset.

In [12]:
```
from sklearn import datasets
#to read more about load_iris() function refer to: https://scikit-learn.org/
dataset = datasets.load_iris()
```

We create the input matrix $X \in \mathbb{R}^{N \times D}$ and the output vector $y \in \{1, \ldots, C\}^N$. Let's only use `sepal length` and `sepal width` for classification, since we know they have high correlation with the class label. We then randomly split the data into train and test and visualize the data.

In [13]:
```
x, y = dataset['data'][:,:2], dataset['target']

#print the feature shape and classes of dataset
(N,D), C = x.shape, np.max(y)+1
print(f'instances (N) \t {N} \n features (D) \t {D} \n classes (C) \t {C}')

inds = np.random.permutation(N)

#split the dataset into train and test
x_train, y_train = x[inds[:100]], y[inds[:100]]
x_test, y_test = x[inds[100:]], y[inds[100:]]

#visualization of the data
plt.scatter(x_train[:,0], x_train[:,1], c=y_train, marker='o', label='train'
plt.scatter(x_test[:,0], x_test[:,1], c=y_test, marker='s', label='test')
plt.legend()
plt.ylabel('sepal length')
plt.xlabel('sepal width')
plt.show()
```

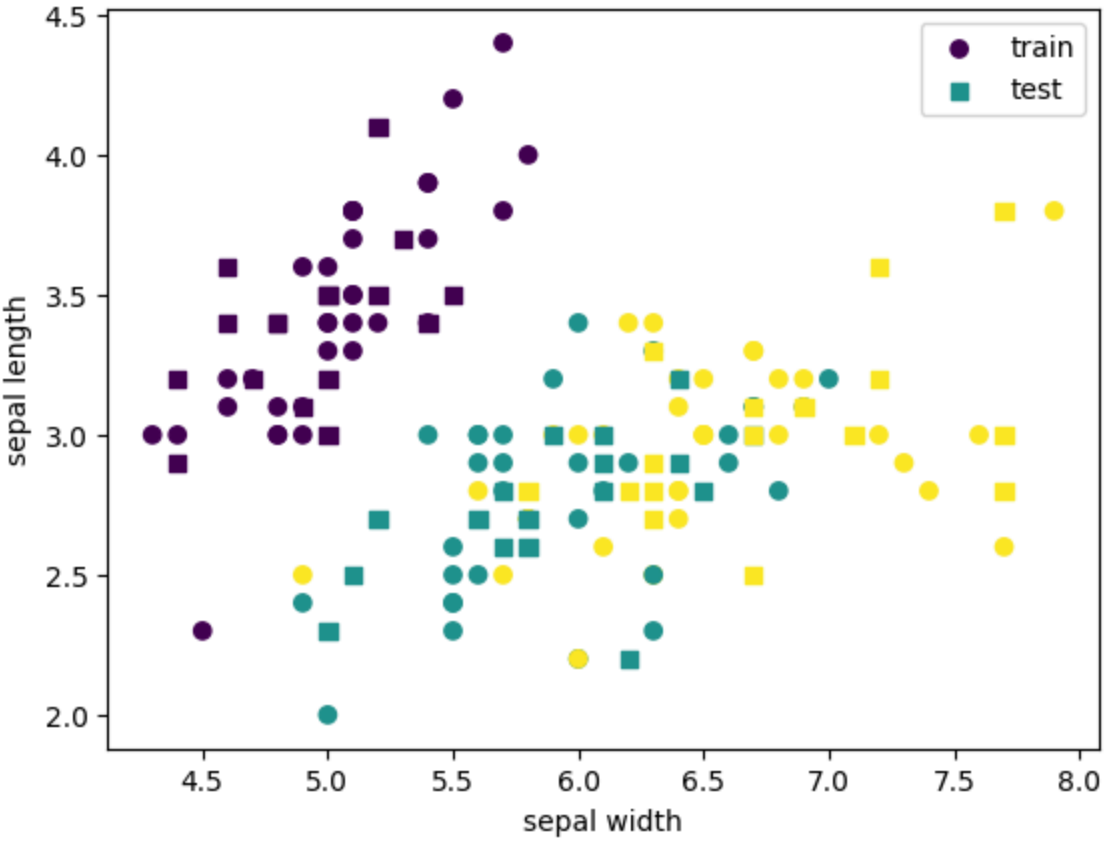
```
instances (N)    150
 features (D)    2
 classes (C)     3
```



[Practice]

Can you replicate this using the `train_test_split` in `sklearn`?

In [15]:
```
from sklearn.model_selection import train_test_split

### Fill in the code below to replicate the figure above using this function
### The exact train and test data points can be different!

# Split the data using train_test_split with the same proportions as above
# Original split: 100 train, 50 test out of 150 total
# That's approximately 67% train, 33% test
x_train_sk, x_test_sk, y_train_sk, y_test_sk = train_test_split(
    x, y, test_size=0.3333, random_state=1234
)

# Visualize the split data
plt.scatter(x_train_sk[:,0], x_train_sk[:,1], c=y_train_sk, marker='o', labe
plt.scatter(x_test_sk[:,0], x_test_sk[:,1], c=y_test_sk, marker='s', label='
plt.legend()
plt.ylabel('sepal length')
plt.xlabel('sepal width')
plt.show()
```



The `KNN` class

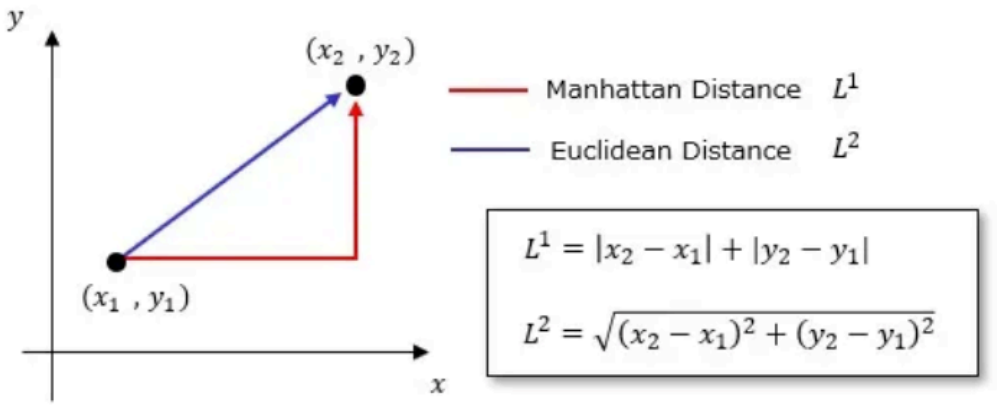
We implement our models as python classes. Two class methods that we usually need to implement are `fit` and `predict`; which respectively perform training by fitting the data, and making prediction on new data. In the `__init__` function, we initialize our model, usually this includes an assignment to *hyper-parameters*.

First though... let's define some distance metrics. I'll give you the Manhattan metric, but you figure out the Euclidean metric yourself!

[Practice]

Can you fill in the euclidean metric?

In [16]:
```
#define the metric we will use to measure similarity
manhattan = lambda x1, x2: np.sum(np.abs(x1 - x2), axis=-1)
euclidean = lambda x1, x2: np.sqrt(np.sum((x1 - x2)**2, axis=-1))
```

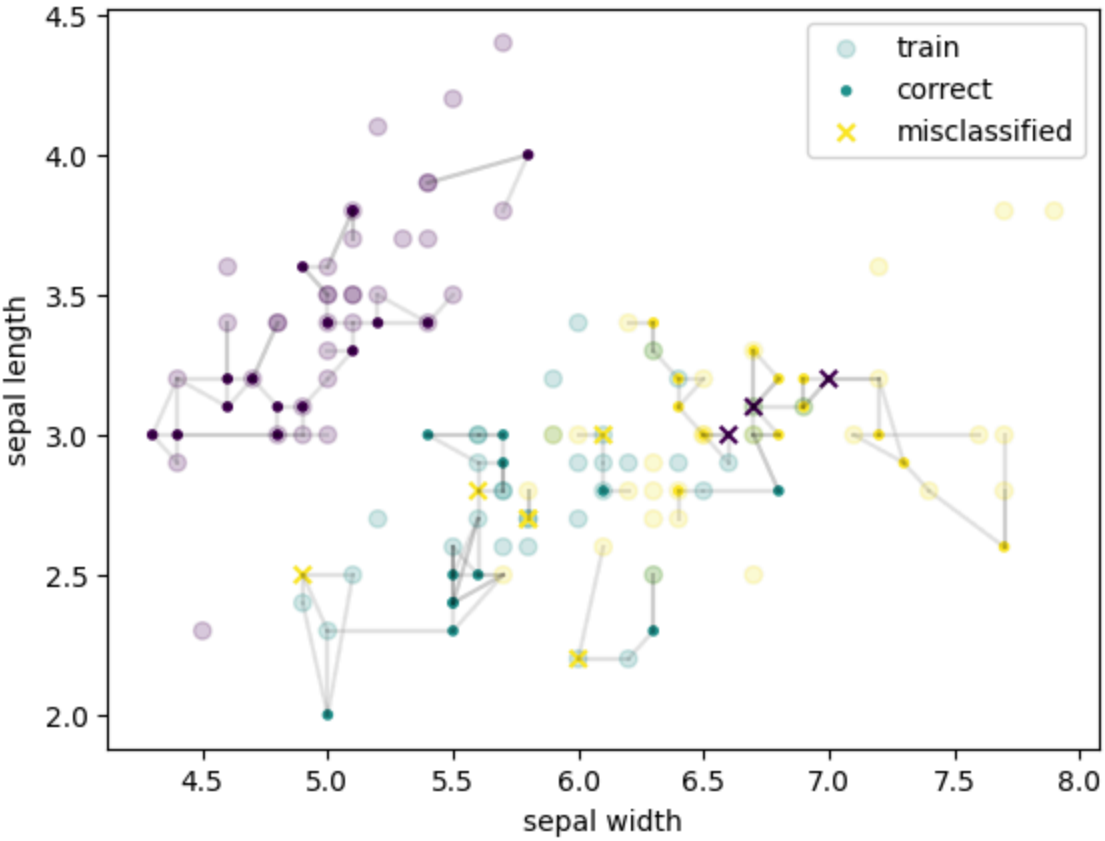


In [17]:
```
class KNN:

    def __init__(self, K=1, dist_fn= manhattan):
        self.dist_fn = dist_fn
        self.K = K
        return

    def fit(self, x, y):
        ''' Store the training data using this method as it is a lazy learne
        self.x = x
        self.y = y
        self.C = np.max(y) + 1
        return self

    def predict(self, x_test):
        ''' Makes a prediction using the stored training data and the test d
        num_test = x_test.shape[0]
        #calculate distance between the training & test samples and returns
        distances = self.dist_fn(self.x[None,:,:], x_test[:,None,:])
        #ith-row of knns stores the indices of k closest training samples to
        knns = np.zeros((num_test, self.K), dtype=int)
        #ith-row of y_prob has the probability distribution over C classes
        y_prob = np.zeros((num_test, self.C))
        for i in range(num_test):
            knns[i,:] = np.argsort(distances[i])[:self.K]
            y_prob[i,:] = np.bincount(self.y[knns[i,:]], minlength=self.C) #
        #y_prob /= np.sum(y_prob, axis=-1, keepdims=True)
        #simply divide by K to get a probability distribution
        y_prob /= self.K
        return y_prob, knns
```

We next `fit` the model(for KNN no learning occurs in training time), and make a prediction on test set(all the computation takes place during testing). We further connect each test node to its closest nearest neighbors in the plot. Here we're using the manhattan metric as the default.

In [18]:
```
model = KNN(K=3)

y_prob, knns = model.fit(x_train, y_train).predict(x_test)
print('knns shape:', knns.shape)
print('y_prob shape:', y_prob.shape)

#To get hard predictions by choosing the class with the maximum probability
y_pred = np.argmax(y_prob,axis=-1)
accuracy = np.sum(y_pred == y_test)/y_test.shape[0]

print(f'accuracy is {accuracy*100:.1f}.')

#boolean array to later slice the indexes of correct and incorrect predictio
correct = y_test == y_pred
incorrect = np.logical_not(correct)

#visualization of the points
plt.scatter(x_train[:,0], x_train[:,1], c=y_train, marker='o', alpha=.2, lab
plt.scatter(x_test[correct,0], x_test[correct,1], marker='.', c=y_pred[corre
plt.scatter(x_test[incorrect,0], x_test[incorrect,1], marker='x', c=y_test[i

#connect each node to k-nearest neighbours in the training set
for i in range(x_test.shape[0]):
    for k in range(model.K):
        hor = x_test[i,0], x_train[knns[i,k],0]
        ver = x_test[i,1], x_train[knns[i,k],1]
        plt.plot(hor, ver, 'k-', alpha=.1)

plt.ylabel('sepal length')
plt.xlabel('sepal width')
plt.legend()
plt.show()
```

```
knns shape: (50, 3)
y_prob shape: (50, 3)
accuracy is 82.0.
```

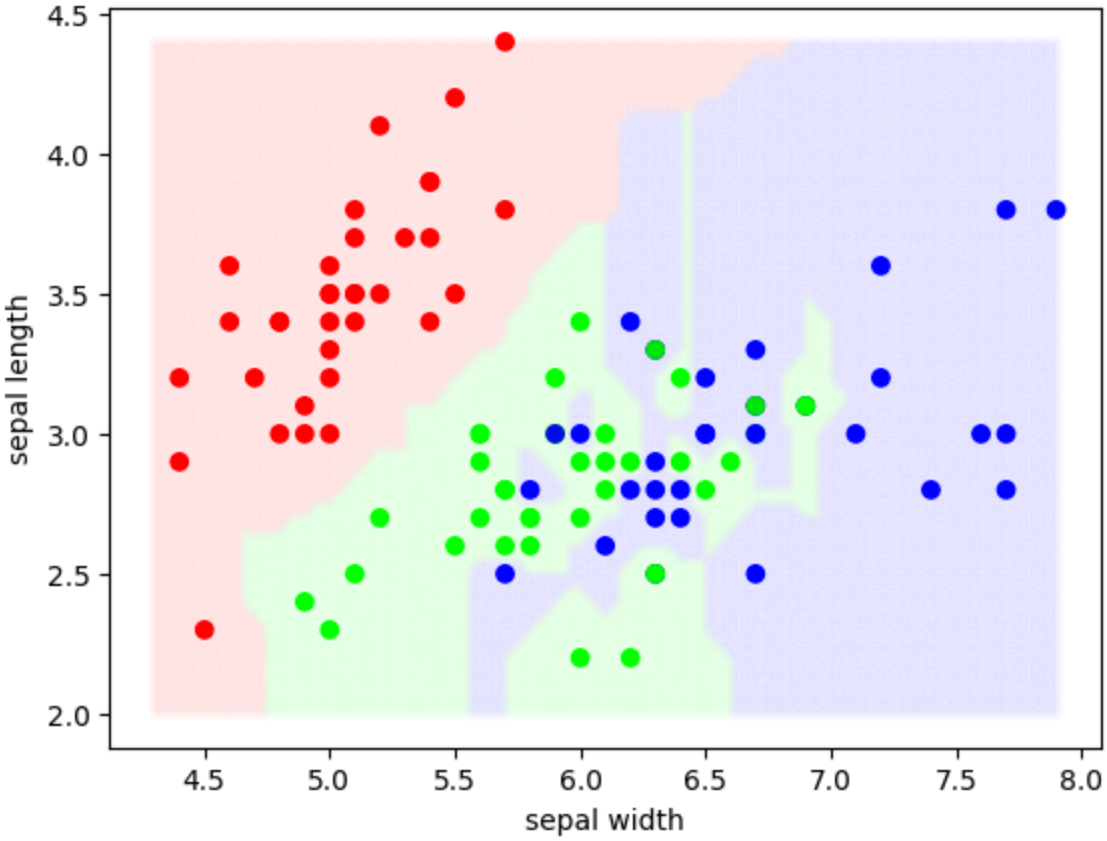


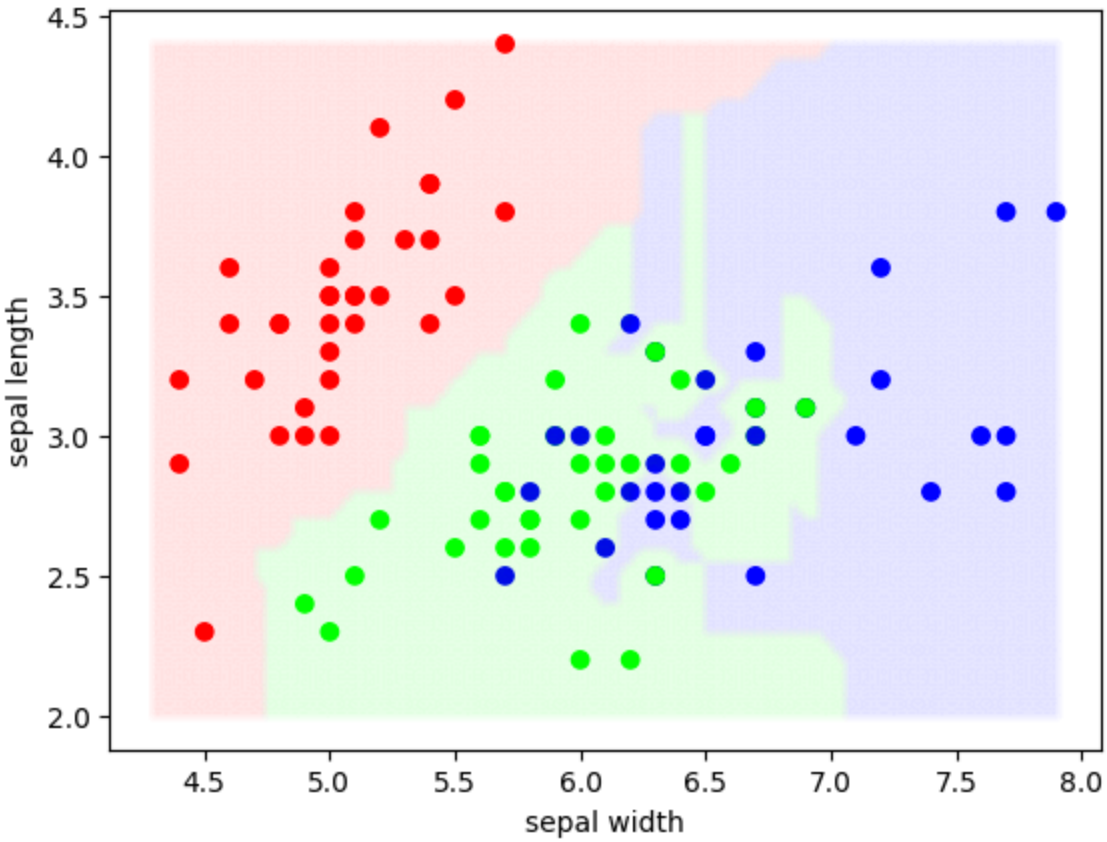
Decision Boundaries

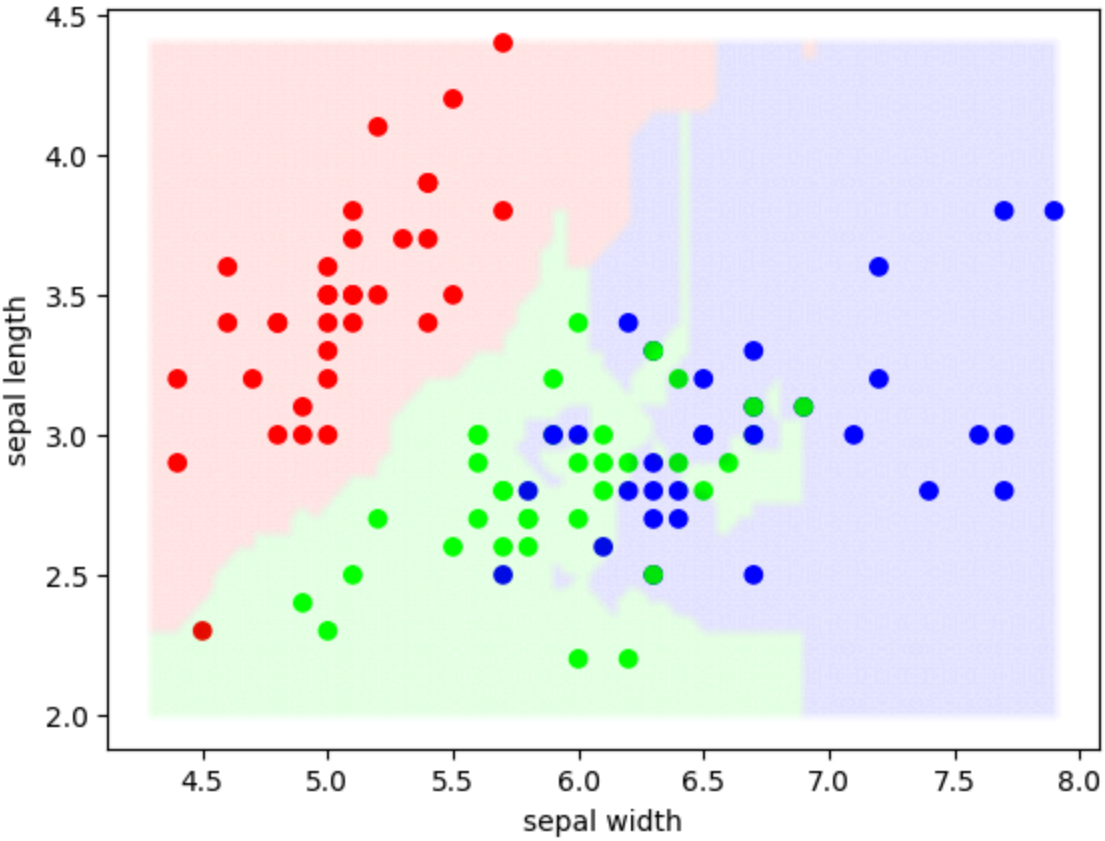
To draw the decision boundary we classify all the points on a 2D grid. The `meshgrid` function creates all the points on the grid by taking discretizations of horizontal and vertical axes.

In [19]:
```
#we can make the grid finer by increasing the number of samples from 200 to
x0v = np.linspace(np.min(x[:,0]), np.max(x[:,0]), 200)
x1v = np.linspace(np.min(x[:,1]), np.max(x[:,1]), 200)

#to features values as a mesh
x0, x1 = np.meshgrid(x0v, x1v)
x_all = np.vstack((x0.ravel(),x1.ravel())).T

for k in range(1,4):
  model = KNN(K=k)

  y_train_prob = np.zeros((y_train.shape[0], C))
  y_train_prob[np.arange(y_train.shape[0]), y_train] = 1

  #to get class probability of all the points in the 2D grid
  y_prob_all, _ = model.fit(x_train, y_train).predict(x_all)

  y_pred_all = np.zeros_like(y_prob_all)
  y_pred_all[np.arange(x_all.shape[0]), np.argmax(y_prob_all, axis=-1)] = 1

  plt.scatter(x_train[:,0], x_train[:,1], c=y_train_prob, marker='o', alpha=
```

```
plt.scatter(x_all[:,0], x_all[:,1], c=y_pred_all, marker='.', alpha=0.01)
plt.ylabel('sepal length')
plt.xlabel('sepal width')
plt.show()
```







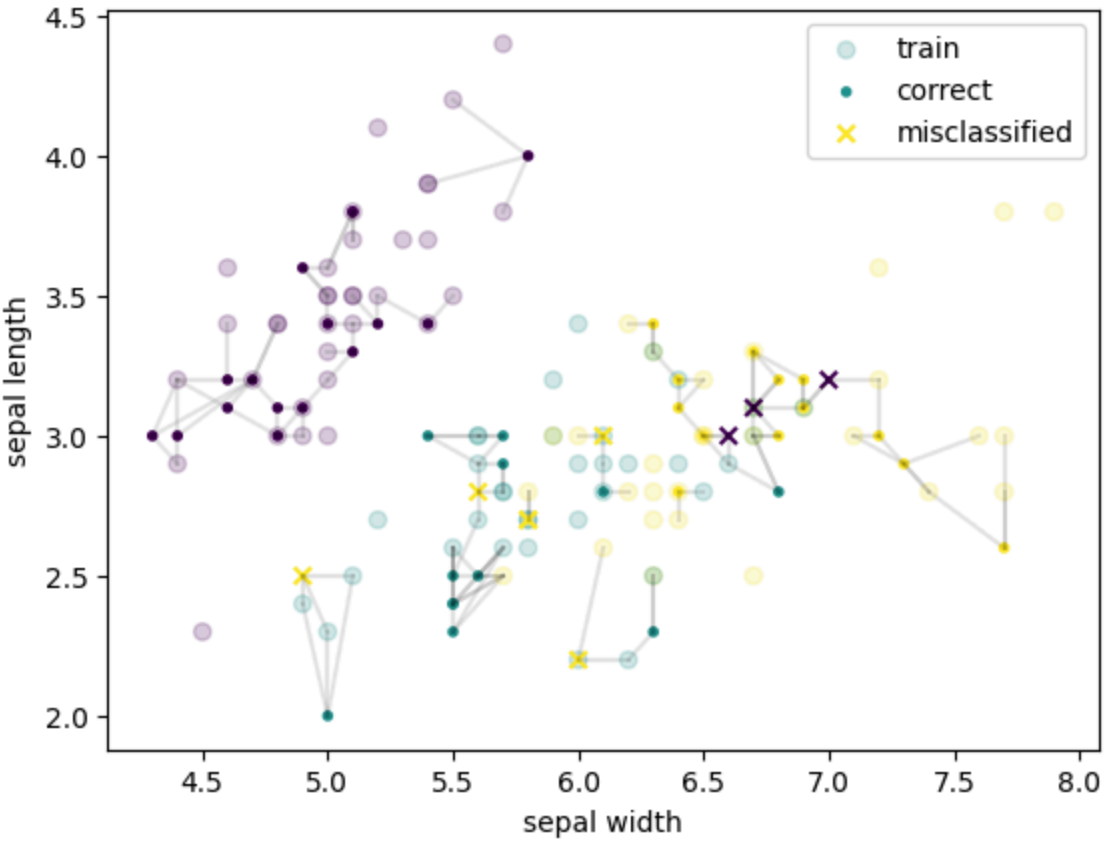
[Practice]

Great! Now can you do that all over again using your euclidean distance metric you defined yourself??? Where do you need to change the code to use that metric??

In [20]:
```
model = KNN(K=3, dist_fn=euclidean)  # change distance to euclidean

y_prob, knns = model.fit(x_train, y_train).predict(x_test)
print('knns shape:', knns.shape)
print('y_prob shape:', y_prob.shape)

#To get hard predictions by choosing the class with the maximum probability
y_pred = np.argmax(y_prob,axis=-1)
accuracy = np.sum(y_pred == y_test)/y_test.shape[0]

print(f'accuracy is {accuracy*100:.1f}.')

#boolean array to later slice the indexes of correct and incorrect predictio
correct = y_test == y_pred
incorrect = np.logical_not(correct)

#visualization of the points
plt.scatter(x_train[:,0], x_train[:,1], c=y_train, marker='o', alpha=.2, lab
plt.scatter(x_test[correct,0], x_test[correct,1], marker='.', c=y_pred[corre
plt.scatter(x_test[incorrect,0], x_test[incorrect,1], marker='x', c=y_test[i

#connect each node to k-nearest neighbours in the training set
for i in range(x_test.shape[0]):
```

```
    for k in range(model.K):
        hor = x_test[i,0], x_train[knns[i,k],0]
        ver = x_test[i,1], x_train[knns[i,k],1]
        plt.plot(hor, ver, 'k-', alpha=.1)

plt.ylabel('sepal length')
plt.xlabel('sepal width')
plt.legend()
plt.show()
```

```
knns shape: (50, 3)
y_prob shape: (50, 3)
accuracy is 82.0.
```



[Practice]

Ok now that we've implemented our own function, let's try it with the sklearn function! Don't forget to use a test size of .3333 (to mirror the same train/test split above). Also, it's fine to use the default distance metric in KNeighborsClassifier.

In [34]:
```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

## Split the data into training and testing data (test_size=.3333)
x_train_sk, x_test_sk, y_train_sk, y_test_sk = train_test_split(
    x, y, test_size=0.3333, random_state=1234
)
```

```
## Instantiate learning model  with  different k's

for n in range(1,21):
    knn_sklearn = KNeighborsClassifier(n_neighbors=n)

    ## Fit the model
    knn_sklearn.fit(x_train_sk, y_train_sk)

    ## Predict the Test set results
    y_pred_sk = knn_sklearn.predict(x_test_sk)

    ## Print the accuracy score (should be above 70%!!!)
    accuracy_sk = accuracy_score(y_test_sk, y_pred_sk)
    print(f'sklearn KNN ({n}) accuracy: {accuracy_sk*100:.1f}%')
```

```
sklearn KNN (1) accuracy: 66.0%
sklearn KNN (2) accuracy: 68.0%
sklearn KNN (3) accuracy: 76.0%
sklearn KNN (4) accuracy: 72.0%
sklearn KNN (5) accuracy: 78.0%
sklearn KNN (6) accuracy: 84.0%
sklearn KNN (7) accuracy: 90.0%
sklearn KNN (8) accuracy: 82.0%
sklearn KNN (9) accuracy: 84.0%
sklearn KNN (10) accuracy: 82.0%
sklearn KNN (11) accuracy: 82.0%
sklearn KNN (12) accuracy: 82.0%
sklearn KNN (13) accuracy: 82.0%
sklearn KNN (14) accuracy: 82.0%
sklearn KNN (15) accuracy: 82.0%
sklearn KNN (16) accuracy: 82.0%
sklearn KNN (17) accuracy: 82.0%
sklearn KNN (18) accuracy: 84.0%
sklearn KNN (19) accuracy: 84.0%
sklearn KNN (20) accuracy: 84.0%
```

Easier huh?!