Artificial Intelligence and Machine Learning

Unit II

Intro to Supervised Learning and k Nearest Neighbours (k-NN)

Iacopo Masi

In [1]:
```
from IPython.core.display import HTML

HTML("""
<style>
.output_png {
    display: table-cell;
    text-align: center;
    vertical-align: middle;
}
img[alt=regression] { width: "50%"; }
</style>
""")
```

Out[1]:

My own latex definitions

In [2]:
```python
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline
plt.style.use('seaborn-whitegrid')

font = {'family' : 'Times',
        'weight' : 'bold',
        'size'   : 12}

matplotlib.rc('font', **font)


# Aux functions

def plot_grid(Xs, Ys, axs=None):
    ''' Aux function to plot a grid'''
    t = np.arange(Xs.size) # define progression of int for indexing colormap
    if axs:
        axs.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        axs.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        axs.axis('scaled') # axis scaled
    else:
        plt.plot(0, 0, marker='*', color='r', linestyle='none') #plot origin
        plt.scatter(Xs,Ys, c=t, cmap='jet', marker='.') # scatter x vs y
        plt.axis('scaled') # axis scaled

def linear_map(A, Xs, Ys):
    '''Map src points with A'''
    # [NxN,NxN] -> NxNx2 # add 3-rd axis, like adding another layer
    src = np.stack((Xs,Ys), axis=Xs.ndim)
    # flatten first two dimension
    # (NN)x2
    src_r = src.reshape(-1,src.shape[-1]) #ask reshape to keep last dimension and adjust the rest
    # 2x2 @ 2x(NN)
    dst = A @ src_r.T # 2xNN
    #(NN)x2 and then reshape as NxNx2
    dst = (dst.T).reshape(src.shape)
    # Access X and Y
    return dst[...,0], dst[...,1]


def plot_points(ax, Xs, Ys, col='red', unit=None, linestyle='solid'):
    '''Plots points'''
    ax.set_aspect('equal')
    ax.grid(True, which='both')
    ax.axhline(y=0, color='gray', linestyle="--")
    ax.axvline(x=0, color='gray',  linestyle="--")
    ax.plot(Xs, Ys, color=col)
    if unit is None:
        plotVectors(ax, [[0,1],[1,0]], ['gray']*2, alpha=1, linestyle=linestyle)
    else:
        plotVectors(ax, unit, [col]*2, alpha=1, linestyle=linestyle)

def plotVectors(ax, vecs, cols, alpha=1, linestyle='solid'):
    '''Plot set of vectors.'''
    for i in range(len(vecs)):
        x = np.concatenate([[0,0], vecs[i]])
        ax.quiver([x[0]],
                  [x[1]],
                  [x[2]],
                  [x[3]],
                  angles='xy', scale_units='xy', scale=1, color=cols[i],
                  alpha=alpha, linestyle=linestyle, linewidth=2)
```

```
/var/folders/rt/lg7n4lt1489270pz_18qn1_c0000gp/T/ipykernel_2847/1496334134.py:5: MatplotlibDeprecationWarning:
The seaborn styles shipped by Matplotlib are deprecated since 3.6, as they no longer correspond to the styles s
hipped by seaborn. However, they will remain available as 'seaborn-v0_8-<style>'. Alternatively, directly use t
he seaborn API instead.
  plt.style.use('seaborn-whitegrid')
```

Recap previous lecture

- Recap probability and Gaussian Distribution
- Maximum Likelihood Estimator (MLE)
- MLE with a single Gaussian
- Gaussian Mixture Model (GMM)
- Applications: Anomaly Detection and Data Generation

Today's lecture

Moving to Supervised Learning (Good news is that is easier)

Non-parametric model

1. k-Nearest Neighbours (Today)

2. Decision Trees (Coming up)

This lecture material is taken from

- Cimi Book - Chapter 01
- CSC411: Introduction to Machine Learning
- Cornell ML course
- Prof. Olga Veksler ML course

Unsupervised Learning

Objective and Motivation: The goal of unsupervised learning is to find **hidden patterns** in unlabeled data.

$$\underbrace{\{\mathbf{x}_i\}_{i=1}^N}_{\text{known}} \sim \underbrace{\mathcal{D}}_{\text{unknown}} \tag{1}$$

- Unlike in supervised learning, any data points is not paired with a label.

- As you can see the unsupervised learning problem is ill-posed (which hidden patterns?) and in principle more difficult than supervised learning.

- Unsupervised learning can be thought of as "finding structure" in the data.

Supervised Learning

Objective and Motivation: The goal of supervised learning is to find **relations and association** between pairs of datapoints paired with a label.

$$\underbrace{\{\mathbf{x}_i, y_i\}_{i=1}^N}_{\text{known}} \sim \underbrace{\mathcal{D}}_{\text{unknown}}$$

- Any data points $\mathbf{x} \in \mathbb{R}^d$ is paired with a label $y \in \mathbb{R}^1$.
- We want to learn a function h parametrized by θ that given $\mathbf{x}$ predicts y or else $h_\theta : \mathbf{x} \to y$
- Note that the hypothesis (the model) can be either **parametric** or **non-parametric**.

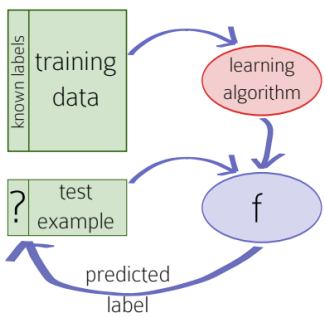


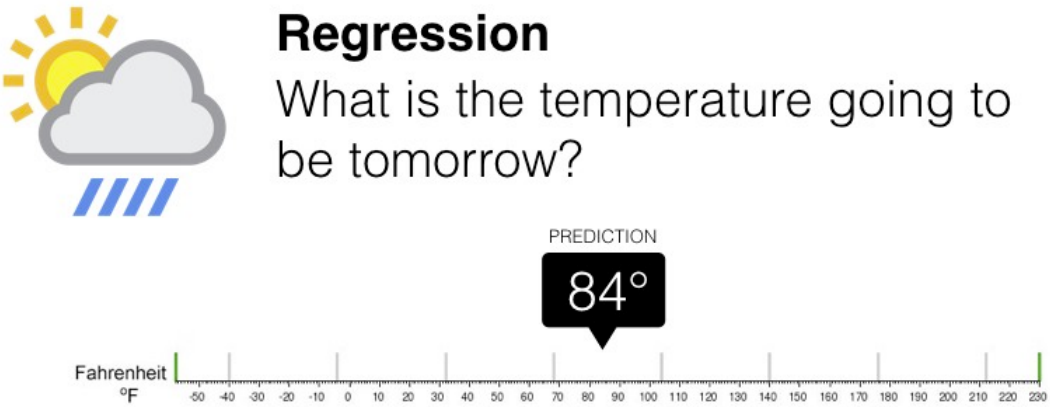
Figure 1.1: The general supervised approach to machine learning: a learning algorithm reads in training data and computes a learned function f. This function can then automatically label future text examples.

Classification vs Regression

- The general settings are very similar. We want to learn a function h parametrized by θ that given $\mathbf{x}$ predicts y or else $h_\theta : \mathbf{x} \to y$
- **Regression:**
  - y is a **continuous, real-value** so $y \in \mathbb{R}^d$ (e.g. predict the interest rate given as input some conditions of the financial market)
- **Classification**:
  - y is a **categorical variable**. For example classify images of `hotdog vs non-hotdog` is same as `0 vs 1`. In this case we do binary classification if the categories are 2.
  - y is a **categorical variable** with more than $N > 2$ then it is **multi-class classification**.

Classification vs Regression

We will see soon that they are **two side of the same coin**



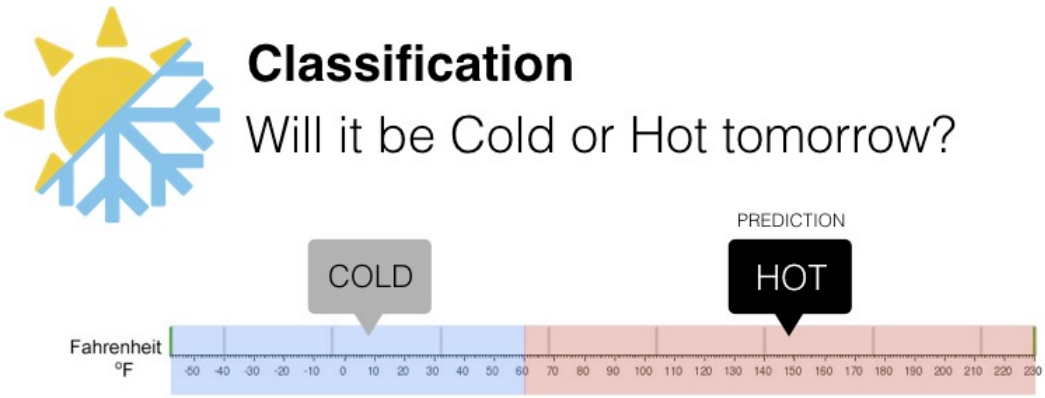


image credit

Parametric vs Non-Parametric

- If parametric, then the model information is "squeezed" into a set of **parameters** θ
  - (e.g. $\theta = \boldsymbol{\mu}, \boldsymbol{\Sigma}$ in the case of a single Gaussian)
- If non-parametric, then the way the model classifies/regresses a value is different and it is **NOT encoded** in some form of parameters θ.

Parametric vs Non-Parametric

- In a **parametric model**, the number of parameters is fixed with respect to the training size. We try to "squeeze" the information of the training set into the parameters useful for the task at hand (i.e. predicting y).
- In a **non-parametric model**, the (effective) number of parameters can grow with the training size.
  - Sometime we refer to non-parametric models as **instance-based learning**.
  - The reason for this name is because **learning amounts to simply storing training data.**
  - Test Instances are classified using `similar` training instances, but what is `similar`?

[Bishop: Pages 120-127]

Non-Parametric Models

- Non-parametric models assume that the data distribution cannot be defined in terms of such a finite set of parameters θ. But they can often be defined by assuming an infinite dimensional θ.
  - The amount of information that θ can capture about the data $\mathcal{D}$ can grow as the amount of data grows.
  - This makes them **more flexible and the advantage is that there is no training basically to do**!
  - Weakness is that **more expensive to store** *(what if we have a billions of images?)*

Two types of Non-Parametric Models

- k Nearest-Neighbours
- Decision Trees

before moving to parametric models for the rest of the course.

Instance-based Learning assumptions

such as k Nearest-Neighbours

Embodies often sensible underlying assumptions:

- Output varies smoothly with input
- The data occupies sub-space of high-dimensional input space

Formalizing the supervised learning problem

Cimi - Chapter 01

Formalizing the notion of **learning** relationships between the data $\mathbf{x}$ and the labels y.

- The performance of the learning algorithm should be measured on $\text{unseen “test” data.}$
- The way in which we measure performance should depend on the problem we are trying to solve (regression vs classification)
- There should be a **strong relationship between the data** that our algorithm sees at training time and the data it sees at test time **(i.i.d. assumptions)**

You will note that is **much better defined for supervised learning.**

Learning = a) Lower ⬇ the cost $\mathcal{L}$ in training **AND** b) ⬇ also in test

In order to accomplish a), let's assume we have a **loss or cost** function:

$$\mathcal{L}(\underbrace{\hat{y}}_{pred.}, \underbrace{y}_{gt}) \quad \text{where} \quad \hat{y} = h_\theta(\mathbf{x})$$

- The job of $\mathcal{L}$ is to tell us how **“bad” a system's prediction is, in comparison to the truth.**
- In particular, if y is what is defined as **ground-truth value or label** and $\hat{y}$ is the machine prediction.
- You can view $\mathcal{L}$ as a measure of the system error over the training set.

Loss function

- Note that the **loss function** is something that you must decide on based on the goals of learning.
- There are multiple loss functions depending on what we want to achieve.

Regression loss can be squared error

$$\mathcal{L}(\hat{y}, y) = (\hat{y} - y)^2$$

Regression loss can be absolute error

$$\mathcal{L}(\hat{y}, y) = \ |\hat{y} - y|$$

Binary Classification loss can be zero/one loss:

$$\mathcal{L}(y, \hat{y}) = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{otherwise} \end{cases}$$

Probabilistic Modeling of Learning

Objective and Motivation: The goal of supervised learning is to find **relations and associations** between pairs of datapoints paired with a label.

$$\underbrace{\{\mathbf{x}_i, y_i\}_{i=1}^N}_{\text{known}} \sim \underbrace{\mathcal{D}}_{\text{unknown}}$$

- There is a probability distribution $\mathcal{D}$ over **input/output pairs** $p(\mathbf{x}, y)$
- This is often called the data generating distribution.

A useful way to think about $\mathcal{D}$ is that it gives high probability to reasonable $(\mathbf{x}, y)$ pairs, and low probability to unreasonable $(\mathbf{x}, y)$ pairs.

- In case we do not have any prior on the shape of $\mathcal{D}$ (i.e. is it Gaussian?), we can treat each training sample:

  $$\mathbf{x}_i, y_i \sim p(\mathbf{x}, y)$$

  as a random sample from this distribution.
- The set of $\{\mathbf{x}_i\}_i^N$ is said **training data**.
- We only see samples from

  $$\mathbf{x}_i, y_i \sim p(\mathbf{x}, y)$$

Sample space Ω

- **Quiz:** with is the sample space for $\mathbf{x}_i \sim p(\mathbf{x}, y)$ if $\mathbf{x}$ is a `2x2` image that takes values in `[0,1,2]` with class labels `[hot, cold]` $

- Sample space is the set of all possible outcomes of $\mathbf{x}$ times the possible values of y

- An image $\mathbf{x}$ `HxW` that takes values `P=[0,1,2]` can be combined in $|P|^{(H \times W)}$ ways

- So all possible images are $3^4 = 81$ images that can take either label `hot` or `cold` thus

- `81x2` is the sample space of this toy problem joint distribution
- **What is the sample space of an IPhone camera for a binary classification problem?**

Probabilistic Modeling of Learning

Based on this training data, we need to **induce** a function f that maps new inputs $\mathbf{x}$ to corresponding prediction $\hat{y}$.

The key property that f should obey is that it should do well (as measured by $\mathcal{L}$) on future examples that are also drawn from $\mathcal{D}$.

Formally, it's **expected loss** ϵ **over** $\mathcal{D}$ **should be as small as possible**:

$$\epsilon \triangleq \mathbb{E}_{(x,y)\sim\mathcal{D}}[\mathcal{L}(y, f(x))] = \sum_{(x,y)} \underbrace{\mathcal{D}(x,y)}_{\text{weight/prob}} \underbrace{\mathcal{L}(y, f(x))}_{\text{function}}$$

Empirical Risk Minimization

- The difficulty in minimizing our expected loss is that we don't know what $\mathcal{D}$ is!
- All we have access to is some training data sampled from it!
- The training data consists of N-many input/output pairs, $(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)$.

Empirical Risk Minimization

$$\begin{aligned}
\mathbb{E}_{(x,y)\sim D}[\mathcal{L}(y, f(x))] &= \textstyle\sum_{(x,y)\in D}[D(x,y)\mathcal{L}(y, f(x))] && \text{def. of expect. of function } \mathbb{E}[f(\mathbf{x})] = \sum p(\mathbf{x}) f(\mathbf{x}) \\
&= \textstyle\sum_{n=1}^{M}\left[D\left(x_n, y_n\right)\mathcal{L}\left(y_n, f\left(x_n\right)\right)\right] && D \text{ is discrete and finite, } M = \text{all possible samples} \\
&\approx \textstyle\sum_{n=1}^{N}\left[\frac{1}{N}\mathcal{L}\left(y_n, f\left(x_n\right)\right)\right] && \textbf{law of large numbers}\text{ , } N = \text{number of training points} \\
&= \tfrac{1}{N}\textstyle\sum_{n=1}^{N}\mathcal{L}\left(y_n, f\left(x_n\right)\right) && \text{rearranging terms}
\end{aligned}$$

Empirical Risk Minimization (ERM)

- Minimizes the average loss function over the training set with equal probability to each sample
- If you take weighted average instead, it is a way to saying this sample is more probably or put more emphasis on this sample.
- Works under the **weak law of large numbers** and approximiation vanishes when $N \mapsto \infty$

$$\epsilon = \frac{1}{N}\sum_{n=1}^{N}\mathcal{L}\left(y_n, f\left(x_n\right)\right)$$

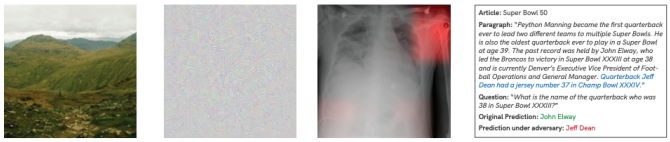
Watch out: Spurious Correlations

- In a famous--if possibly apocryphal--example from the 1970s, the US Government wanted to **train a classifier to distinguish between US tanks and Russian tanks.**
- They collected a training and test set, and managed to **build a classifier with nearly 100% accuracy on that data.**
- **But when this classifier was run in the "real world", it failed miserably.**

Watch out: Spurious Correlations

It had not, in fact, learned to distinguish between US tanks and Russian tanks, **but rather just between clear photos and blurry photos.**

In this case, there was a **bias** in the training data (due to how the training data was collected) that caused the learning algorithm to learn something other than what we were hoping for.



Task for DNN	Caption image	Recognise object	Recognise pneumonia	Answer question
Problem	Describes green hillside as grazing sheep	Hallucinates teapot if certain patterns are present	Fails on scans from new hospitals	Changes answer if irrelevant information is added
Shortcut	Uses background to recognise primary object	Uses features irrecognisable to humans	Looks at hospital token, not lung	Only looks at last sentence and ignores context

Taken from this article

ERM and Its Limits: Bias-Variance Trade-off

$$\epsilon = \frac{1}{N} \sum_{n=1}^{N} \mathcal{L}\left(y_n, f\left(x_n\right)\right)$$

The error of a ML system can be always decomposed into **two parts**:

- **Bias Error**
- **Variance Error**

Bias-Variance Trade-off

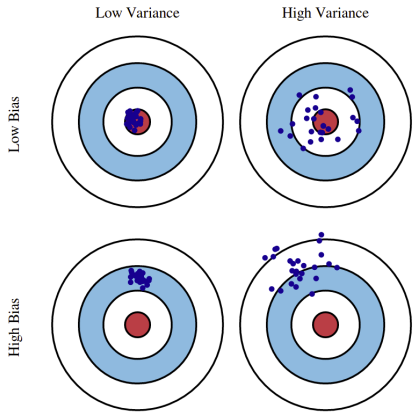
BIAS-Variance Trade-off

- The **bias error** is produced by weak assumptions in the learning algorithm
  - **High bias** can cause an algorithm to **miss the relevant relations between features and target outputs**
  - Problem know as `underfitting`. Solution: increase the complexity/expressiveness of your ML algorithm!

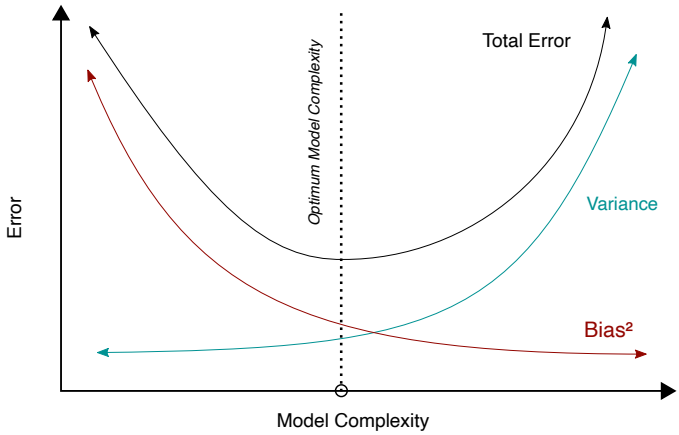
Bias-VARIANCE Trade-off

- The **variance** is an error produced by an **oversensitivity to small fluctuations in the training set**
  - High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs
  - Problem know as `overfitting`. Solution: decrease the model complexity or add strong regularization.

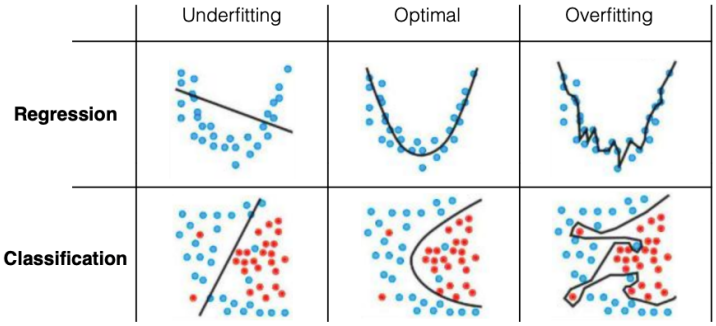
Bias-Variance Tradeoff as Dartboard



Error in function of model complexity



Over or Under Fitting

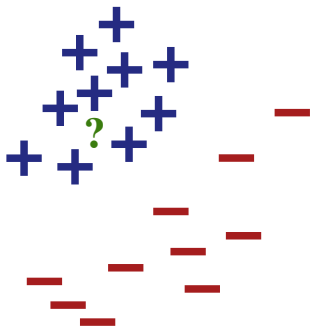


k-Nearest Neighbours

Non-parametric models

Quiz: how do you classify the "? question mark"? Plus or a minus?

- Quick tip on what happens in the machine: the position `x,y` encodes the features; the `+/-` encodes the class labels
- We want to achieve: $(x, y) \mapsto \pm?$
- **Binary Classification**



Assumptions of a ML algorithm (No free lunch theorem)

- There is **NO universal method** that will work for **all the data**
- It is important to know the assumptions of each method so that you can apply it under the right settings in your problem

Assumptions of k-NN

- One reason why you might have thought that is that you believe that
  the label for an example should be similar to the label of nearby points.
- Another way of saying the same is: **The output varies smoothly wrt to the input**
- This is an example of a new form of **inductive bias** or else **assumption that you make before the data arrives**.
- The **nearest neighbor (NN)** classifier is built upon this insight.

Nearest Neighbour

- At training time, we simply **store the entire training set along with labels**
- At test time, we get a test example $\hat{\mathbf{v}}$.
- To predict its label, we find the training example $\mathbf{x}$ that is **most similar** to $\hat{\mathbf{v}}$.

What's "Most Similar"?

We have to formalize **"most similar"**:

- We have the notion of **point in a vector-space.**
- We have the notion of **distance between data points** (we can assume euclidean for now aka ℓ_2)

Most similar $\rightarrow$ **minimum distance in the vector space**: we find the training example $\mathbf{x}_i$ such that

$$i^* = \arg\min_{i \in \mathcal{X}} d(\mathbf{x}, \hat{\mathbf{v}}).$$

- Since $\mathbf{x}_i$ is a training example, it has a corresponding label, y.
- We predict that the label of $\hat{\mathbf{v}}$ is also y.

Nearest Neighbour

Input is the datapoint $\mathbf{v}$; $\mathcal{D} = \{(\mathbf{x}_1, y_1) \ldots, (\mathbf{x}_n, y_n)\}$ is the training set

Algorithm:

1. Find example $(\mathbf{x}^*, t^*)$ *(from $\mathcal{D}$)* closest to the test instance $\mathbf{v}$. That is:

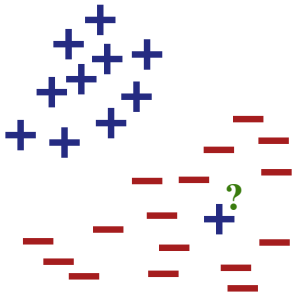
$$\mathbf{x}^* = \underset{\mathbf{x}_i \in \mathcal{D}}{\operatorname{argmin}}\, \mathrm{d}(\mathbf{x}_i, \mathbf{v})$$

where $\mathrm{d}(\cdot, \cdot)$ is a suitable distance metric.

1. Output $y = t^*$

Another quiz is coming and we will be using NN to classify!

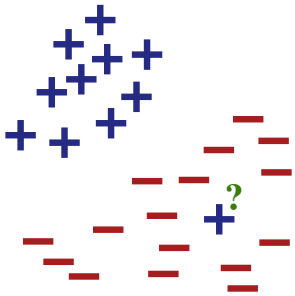
Quiz: Let us use NN to classify!



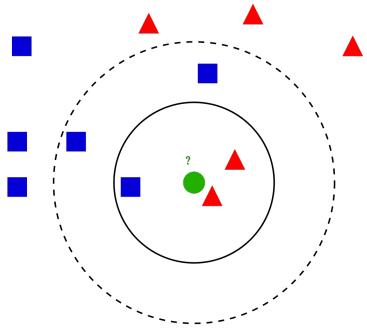
Since the nearest neighbor algorithm only looks at the *single* nearest neighbor, **it cannot consider the `preponderance of evidence` that this point should probably actually be a negative example.** It will make an unnecessary error.

NN is sensitive to class-label noise

- We can improve by performing **smoothing**.
- Why just select a single **neighbour**?
- Indeed we see that if we relax and look at its 3 NN we have **2 minus vs 1 plus.**
- By $\underline{\text{voting}}$ **over the k-NN**, we get the correct class.



Ambigous cases based on the distance and neighbours



The set of k-Nearest Neighbours - Definition

- Denote the set of the k nearest neighbors of $\mathbf{x}$ as $S_{\mathbf{x}}$. Formally S_{x} is defined as $S_{\mathrm{x}} \subseteq D$ s.t. $|S_{\mathrm{x}}| = k$ and $\forall\,(\mathbf{x}', y') \in D \setminus S_{\mathrm{x}}$,

$$\operatorname{dist}(\mathbf{x}, \mathbf{x}') \geq \max_{(\mathbf{x}'', y'') \in S_{\mathrm{x}}} \operatorname{dist}(\mathbf{x}, \mathbf{x}'')$$

- (i.e. every point in D but not in S_{x} is at least as far away from $\mathbf{x}$ as the furthest point in $S_{\mathbf{x}}$).
- We can then define the classifier $h(\cdot)$ as a function returning the most common label in S_{x}:

$$h(\mathbf{x}) = \operatorname{mode}(\{y'' : (\mathbf{x}'', y'') \in S_{\mathbf{x}}\})$$

k-Nearest Neighbour

Input is the datapoint $\mathbf{v}$; $\mathcal{D} = \{(\mathbf{x}_1, y_1) \ldots, (\mathbf{x}_n, y_n)\}$ is the training set

Algorithm:

1. We find first the set of k Nearest Neighbour to $\mathbf{v}$ called S_{v}.

   Formally S_{v} is defined as $S_{\mathrm{v}} \subseteq \mathcal{D}$ s.t. $|S_{\mathrm{v}}| = k$ and $\forall\,(\mathbf{x}', y') \in D \backslash S_{\mathrm{v}}$,

$$\mathrm{d}(\mathbf{v}, \mathbf{x}') \geq \max_{(\mathbf{x}'', y'') \in S_{\mathrm{v}}} \operatorname{dist}(\mathbf{v}, \mathbf{x}'')$$

(i.e. every point in D but not in S_{v} is at least as far away from $\mathbf{v}$ as the furthest point in $S_{\mathbf{v}}$).

```
Technically this part is implemented simply as sorting and choosing top-K in a brute force approach
```

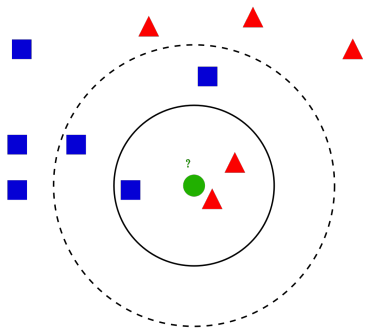
1. Assign the label to $\mathbf{v}$ as most frequent element in the labels of S_{v} That is:

$$y^* = \operatorname{mode}\left(\{y'' : (\mathbf{x}'', y'') \in S_{\mathbf{v}}\}\right) \qquad \text{for classification}$$

the **mode** of a distribution recovers the most frequent element.

🛠️ **Hyper-param: How do we choose k?**

The impact of hyper-parameter k



Which distance metric to use?

There is a family of distances: Minkowski distance

The k-nearest neighbor classifier fundamentally relies on a **distance metric**. The better that metric reflects label similarity, the better the classified will be. The most common choice is one over the family of distances defined by the **Minkowski distance**:

$$\text{dist}(\mathbf{x}, \mathbf{z}) = \left(\sum_{r=1}^{d} |x_r - z_r|^p \right)^{1/p}$$

This distance definition is pretty general and contains many well-known distances as special cases. Can you identify the following candidates?

1. $p = 1:$
2. $p = 2:$
3. $p \to \infty$

Cosine distance

If you want to measure only the angle you can use the **cosine distance**.

What is a good distance here to perform classification (color is the class) ?

- [ ] L1 distance (Manhattan)
- [ ] L2 distance (Euclidean)
- [ ] L_inf distance (Max of abs value)
- [ ] Cosine distance
- [ ] Mahalanobis distance

What is a good distance here to perform classification (color is the class) ?

- [ ] L1 distance (Manhattan)
- [ ] L2 distance (Euclidean)
- [ ] L_inf distance (Max of abs value)
- [x] **Cosine distance** (features of the same classes are almost on the same line)
- [x] **Mahalanobis distance** (but requires to estimate covariance matrices)

The impact of hyper-parameter k

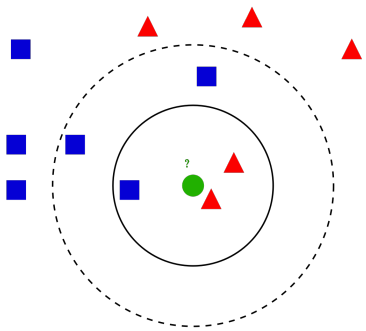
1. Assign the label to $\mathbf{v}$ as most frequent element in the labels of S_{v}. That is:

$$y^* = \text{mode}\left(\{y'' : (\mathbf{x}'', y'') \in S_{\mathbf{v}}\}\right) \qquad \text{for classification}$$

the **mode** of a distribution recovers the most frequent element.

🛠 **Hyper-param: How do we choose k?**

The impact of hyper-parameter k



How do we choose k?

- $k = 1$ is the NN algorithm and may suffer of **overfitting**. You just based your decision on too few informations.
- $k \gg 1$ you may end up with good performance (you smooth out the result, i.e. impose regularization)
- if you have binary classification, k better be **odd** (why is that?)

- Now let's think what happens when $k = |\mathcal{D}|$? i.e. k is equal to the number of datapoints!?

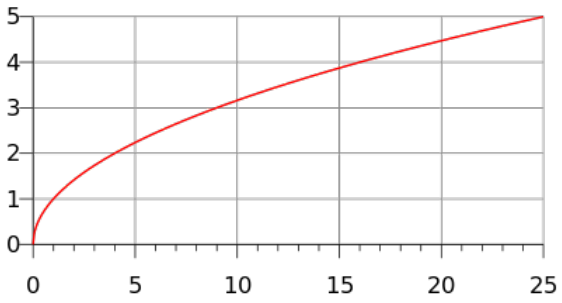
- if $k = |\mathcal{D}|$ then we **underfit** and always **predict the majority class in $\mathcal{D}$.**

Best constant predictor

- if $k = |\mathcal{D}|$ then we underfit and always **predict the majority class in $\mathcal{D}$.**
- You can think of the best constant predictor as a lazy algorithm that just counts the frequency of labels in the data and, without EVEN looking at the input, predicts the most frequent label.

Yes, but how do we choose k?

- We can use **cross-validation** to find k
- Rule of thumb is `k < sqrt(N)`, where N is the number of training examples (idea: if N is small, k should be small but; if N is large K should be increased but not too much; sqrt function increases as N when N is small; for large N the value is kept much lower than N).

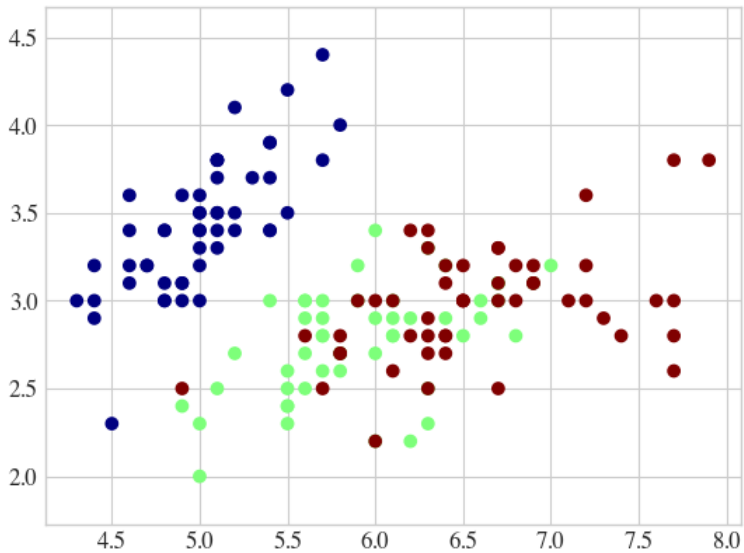


k-NN decision boundary and demo

In [3]:
```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import neighbors, datasets
# import some data to play with
iris = datasets.load_iris()
# we only take the first two features. We could avoid this ugly
# slicing by using a two-dim dataset
X = iris.data[:, :2] # Nx2
y = iris.target      # Nx1
```

The data

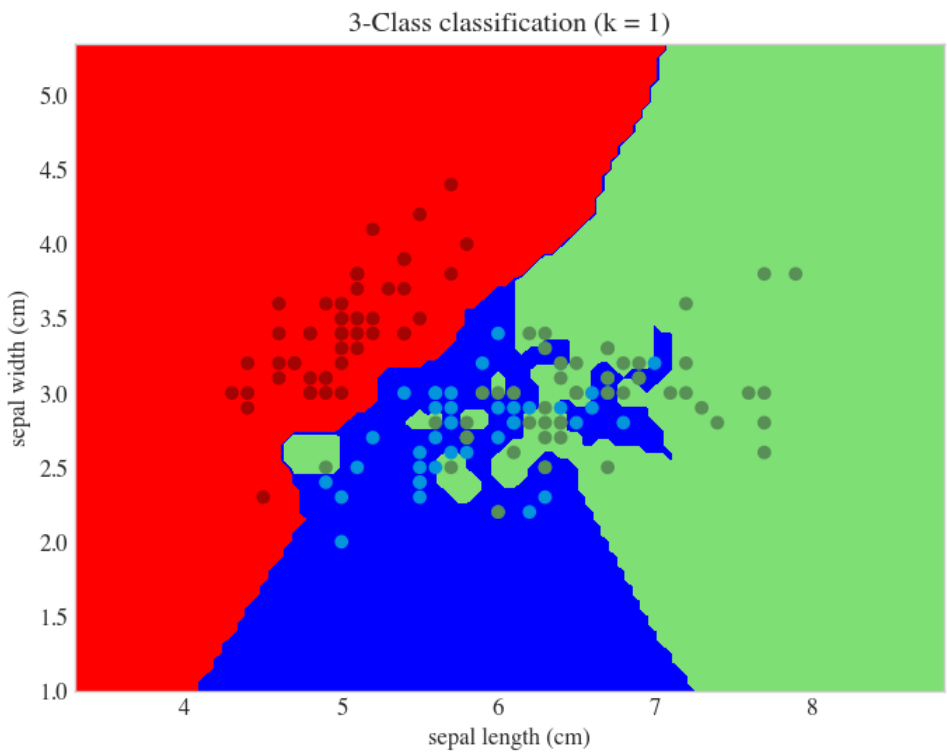
In [4]:
```
plt.scatter(*X.T, c=y, cmap='jet');
plt.axis('equal');
```

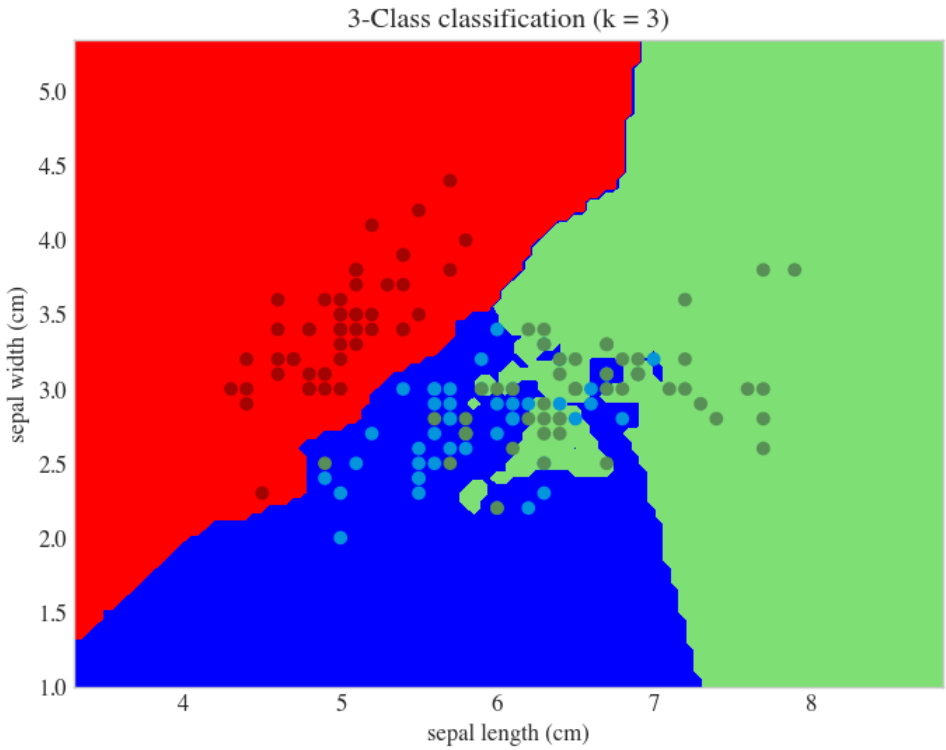


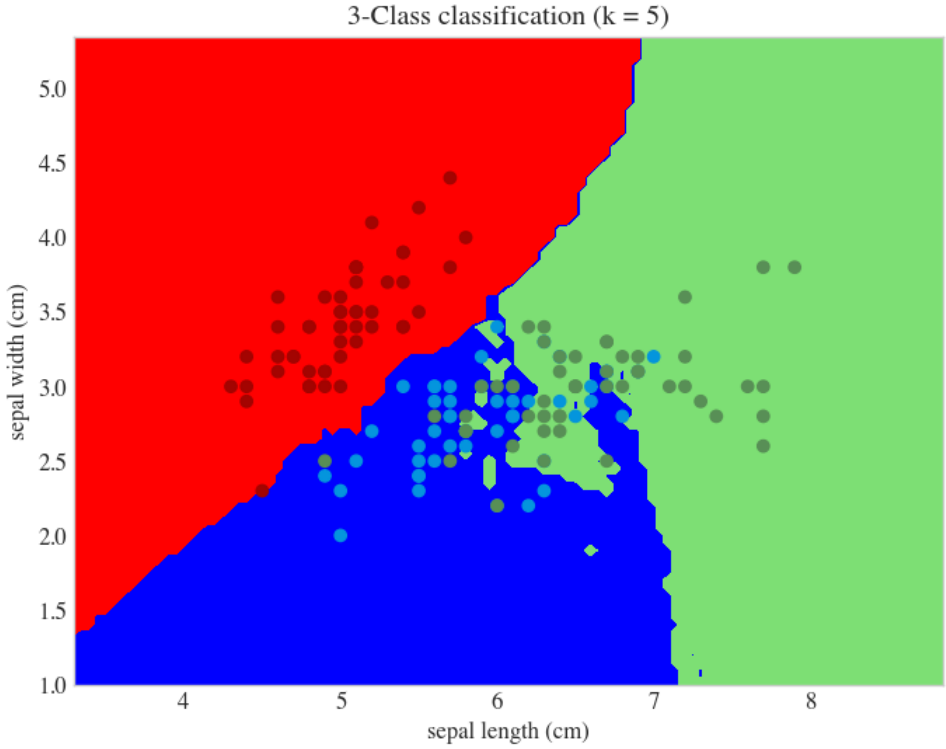
Decision boundaries by varying k

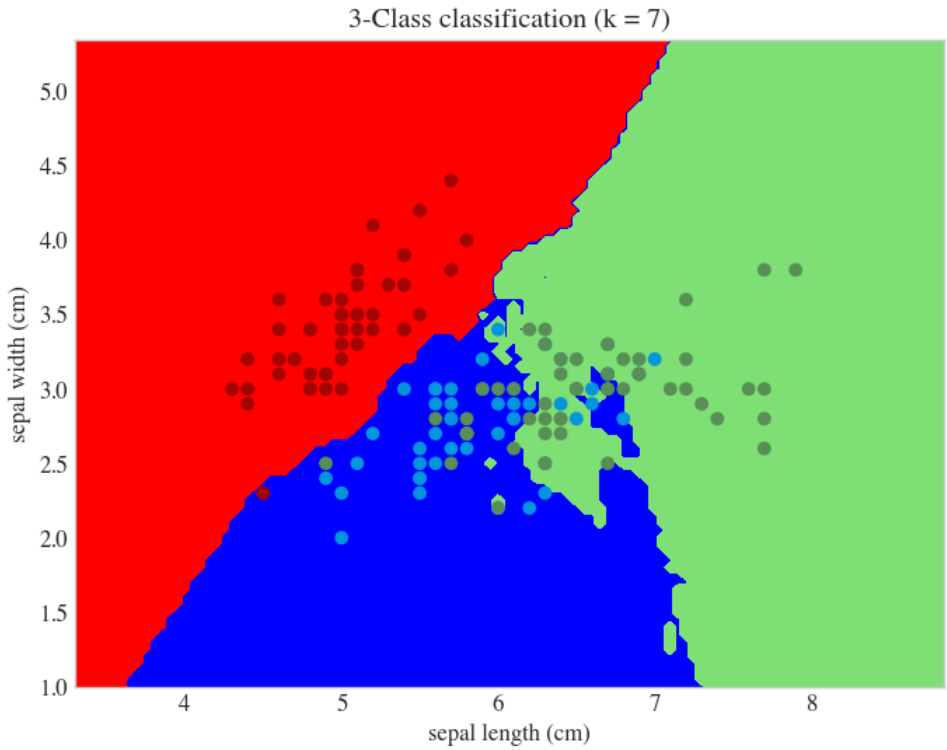
In [5]:

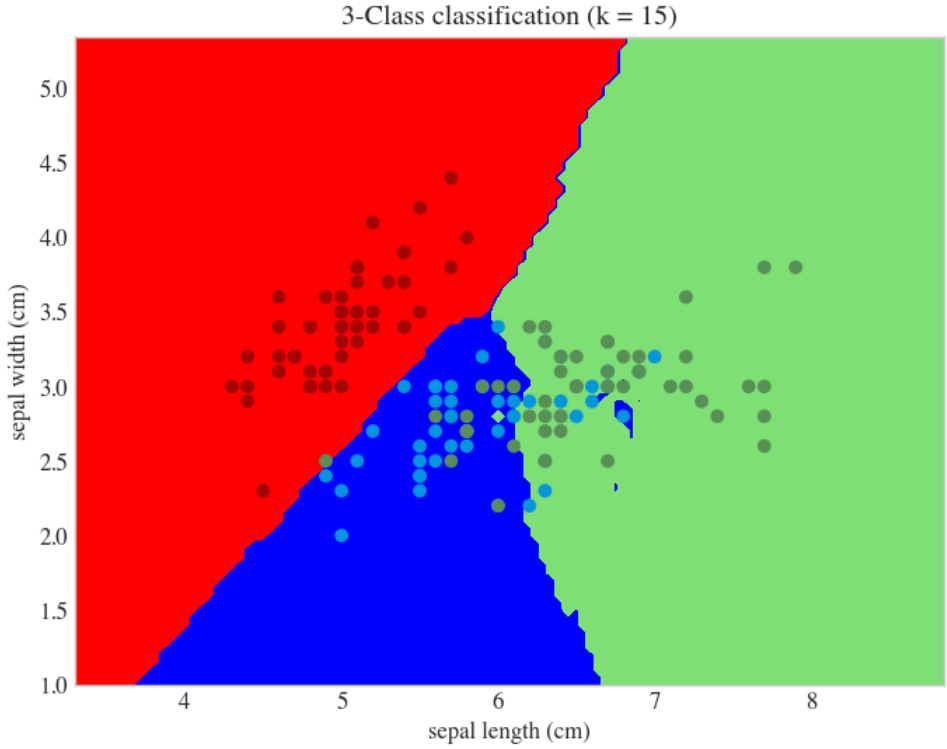
```
from matplotlib.colors import ListedColormap
from sklearn import neighbors, datasets

#####################################################
h = 0.05  # step size in the mesh
#####################################################

# Create color maps
cm = ListedColormap(["#a30401", "#0495dd", "#588f56"])
cm_bright = ListedColormap(["#FF0000", "#0000FF", "#7ddf74"])


for n_neighbors in [1, 3, 5, 7, 15, 21, 27, 39, X.shape[0]]:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = neighbors.KNeighborsClassifier(
        n_neighbors, weights="uniform", algorithm='brute', p=2)
    clf.fit(X, y)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    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, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure(figsize=(8, 6))
    plt.contourf(xx, yy, Z, cmap=cm_bright)

    # Plot also the training points
    plt.scatter(
        x=X[:, 0],
        y=X[:, 1],
        c=y,
        cmap=cm,
    )
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title(
        "3-Class classification (k = %i)" % (n_neighbors)
    )
    plt.xlabel(iris.feature_names[0])
    plt.ylabel(iris.feature_names[1])
plt.show()
```

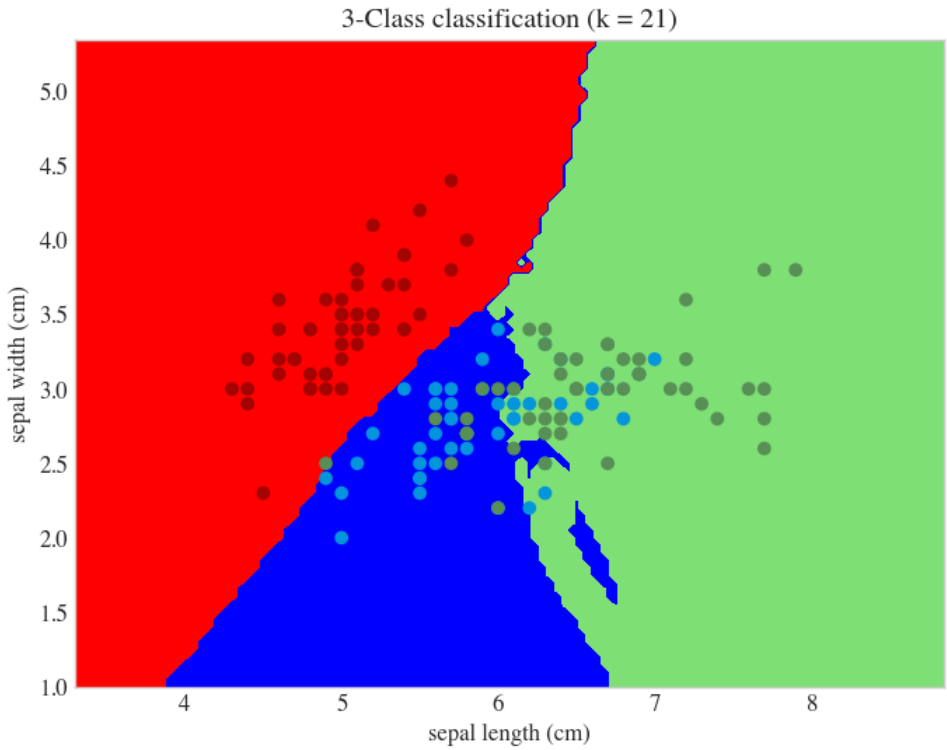


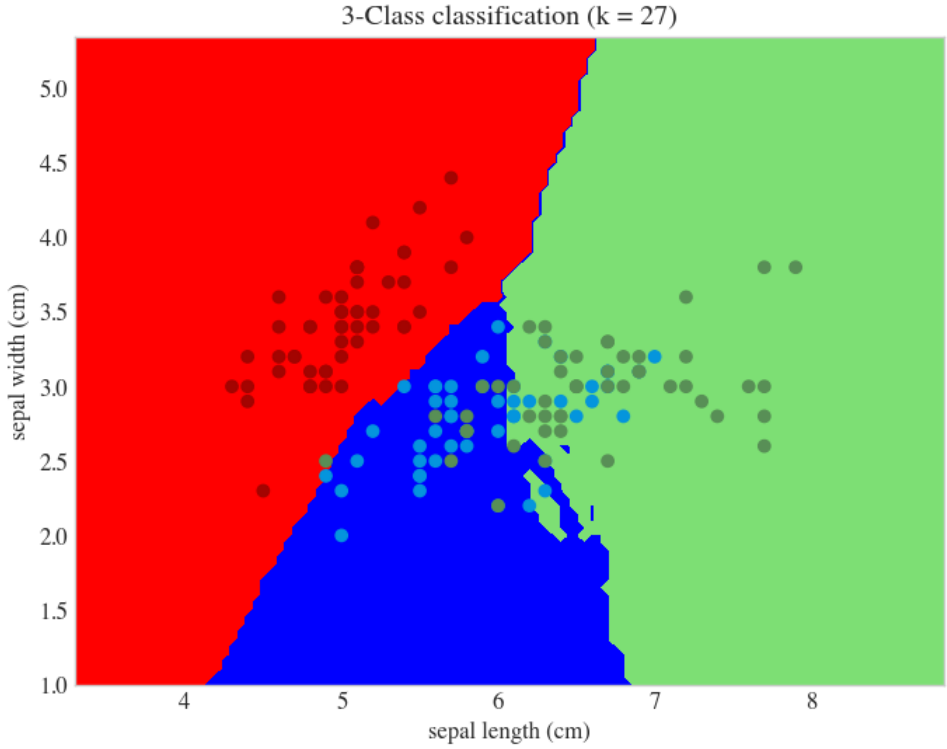

3-Class classification (k = 3)
sepal width (cm)
sepal length (cm)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
4
5
6
7
8

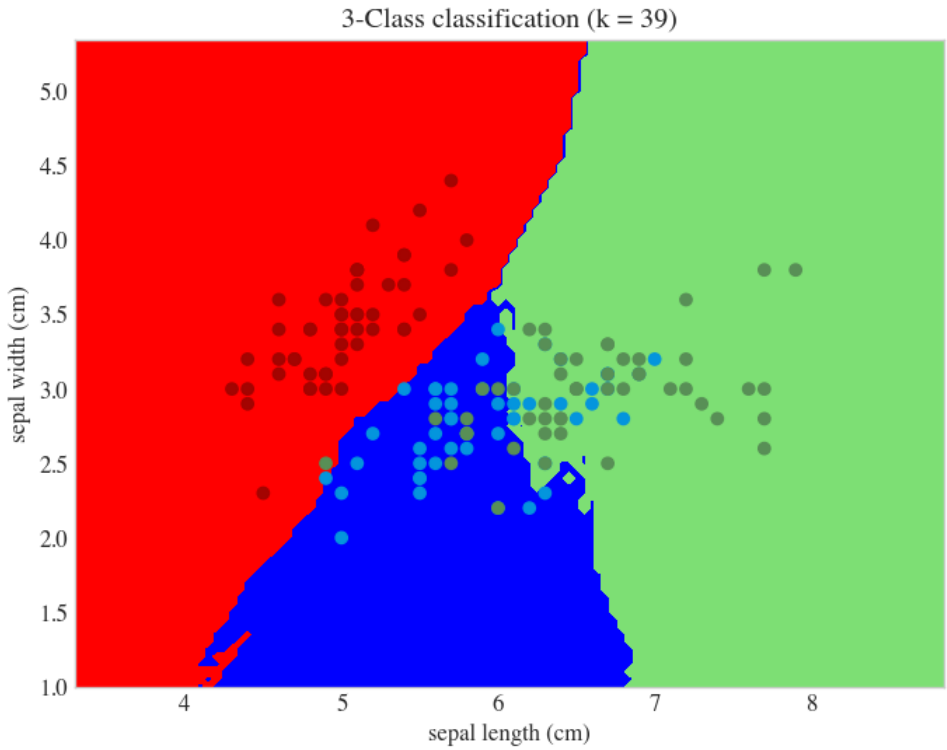

3-Class classification (k = 5)
sepal width (cm)
sepal length (cm)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
4
5
6
7
8

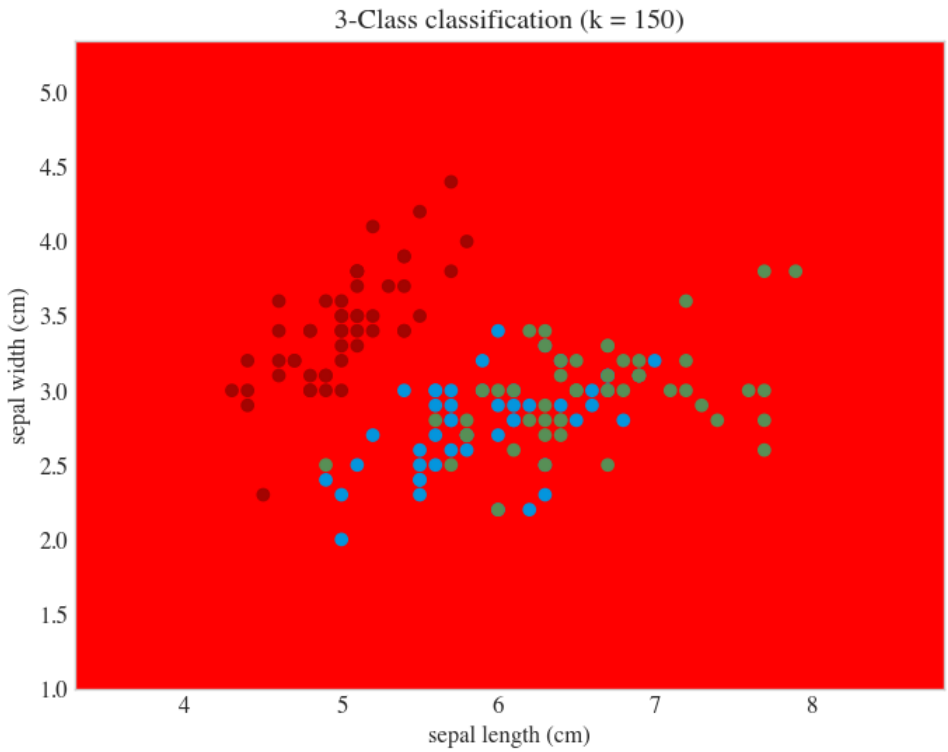

3-Class classification (k = 7)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
sepal width (cm)
4
5
6
7
8
sepal length (cm)


3-Class classification (k = 15)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
sepal width (cm)
4
5
6
7
8
sepal length (cm)


3-Class classification (k = 21)
sepal width (cm)
sepal length (cm)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
4
5
6
7
8

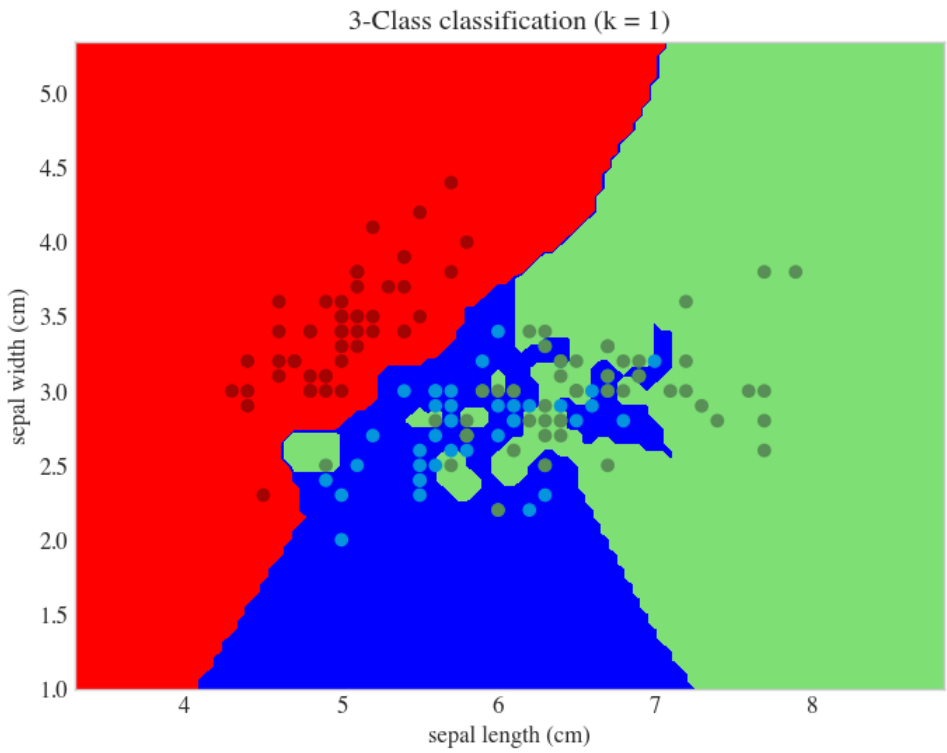

3-Class classification (k = 27)
sepal width (cm)
sepal length (cm)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
4
5
6
7
8

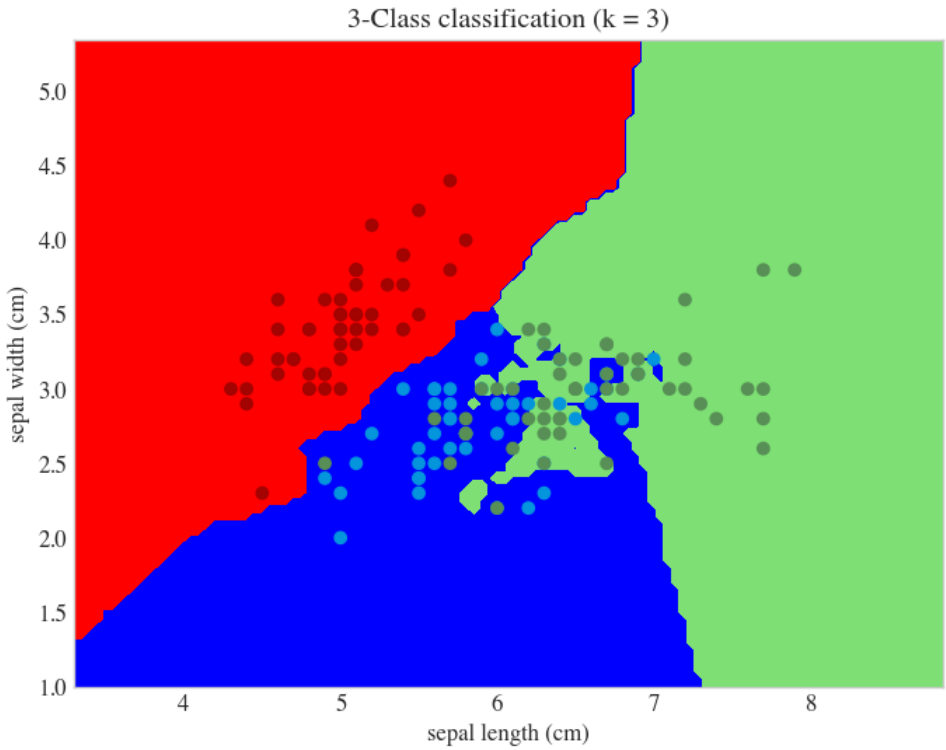

3-Class classification (k = 39)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
sepal width (cm)
4
5
6
7
8
sepal length (cm)

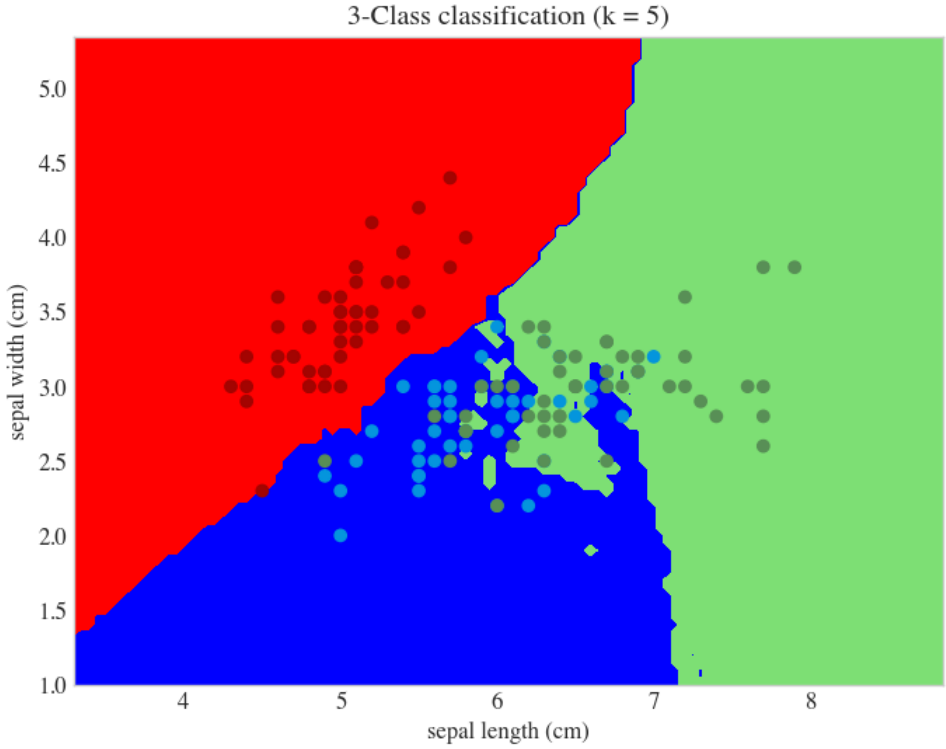

3-Class classification (k = 150)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
sepal width (cm)
4
5
6
7
8
sepal length (cm)

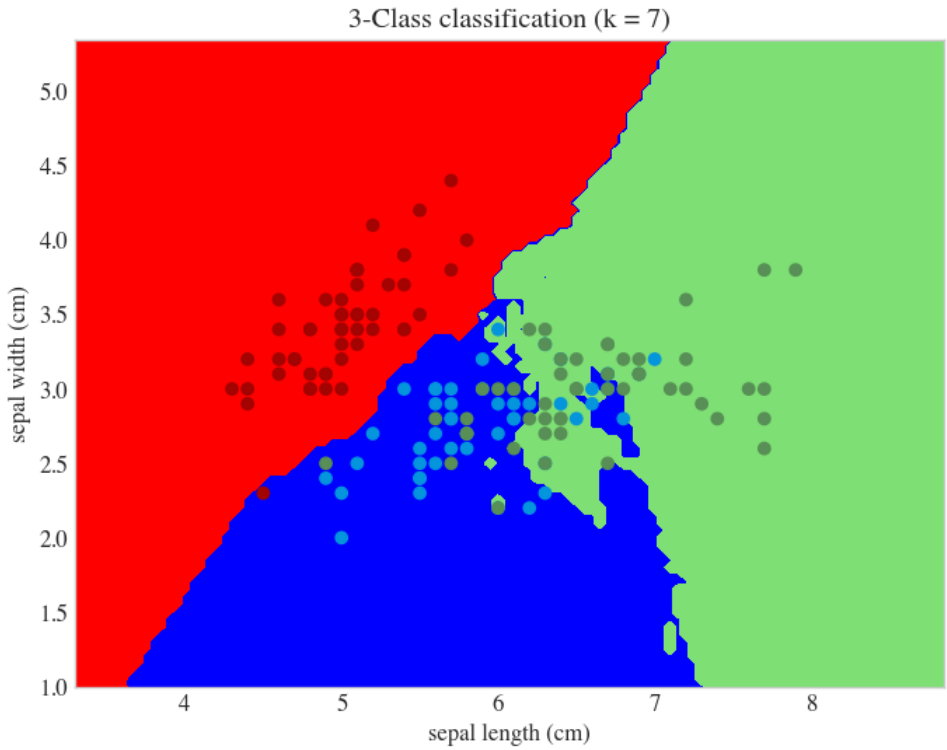
In [6]:

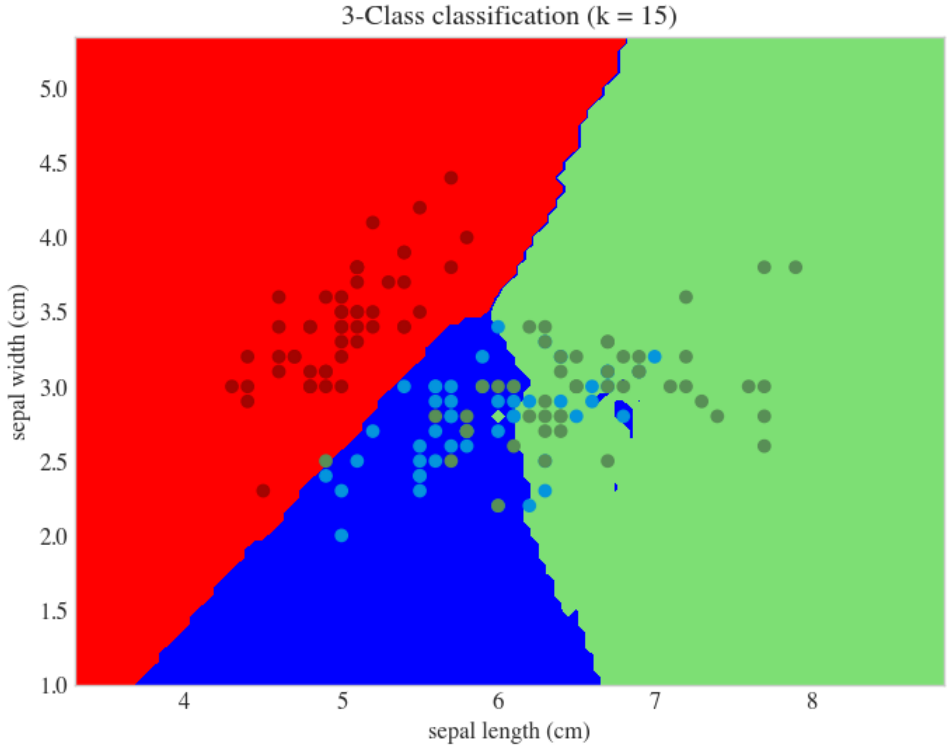
```
from matplotlib.colors import ListedColormap
from sklearn import neighbors, datasets

#####################################################
h = 0.05  # step size in the mesh
#####################################################

# Create color maps
cm = ListedColormap(["#a30401", "#0495dd", "#588f56"])
cm_bright = ListedColormap(["#FF0000", "#0000FF", "#7ddf74"])


for n_neighbors in [1, 3, 5, 7, 15, 21, 27, 39, X.shape[0]]:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = neighbors.KNeighborsClassifier(
        n_neighbors, weights="uniform", algorithm='brute', p=2)
    clf.fit(X, y)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    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, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure(figsize=(8, 6))
    plt.contourf(xx, yy, Z, cmap=cm_bright)

    # Plot also the training points
    plt.scatter(
        x=X[:, 0],
        y=X[:, 1],
        c=y,
        cmap=cm,
    )
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title(
        "3-Class classification (k = %i)" % (n_neighbors)
    )
    plt.xlabel(iris.feature_names[0])
    plt.ylabel(iris.feature_names[1])

plt.show()
```

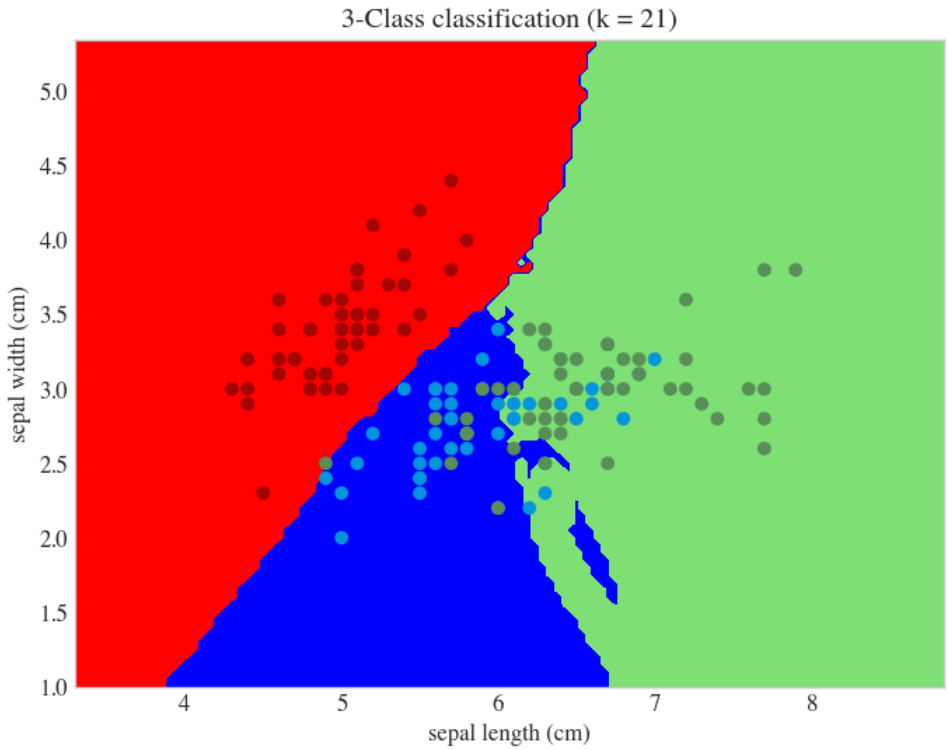


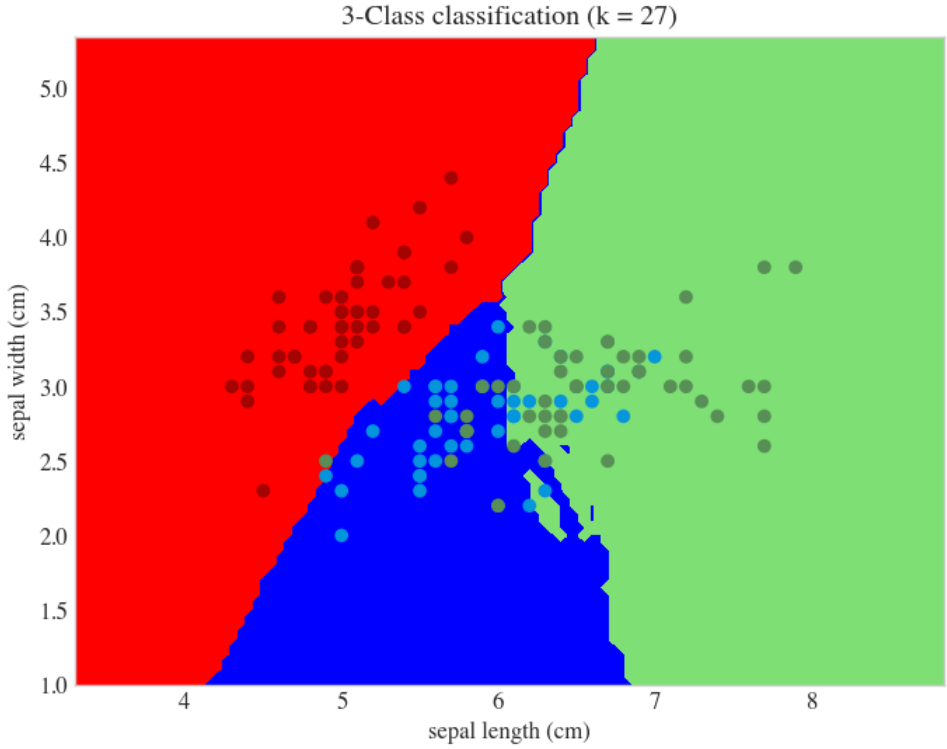

3-Class classification (k = 3)
sepal width (cm)
sepal length (cm)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
4
5
6
7
8

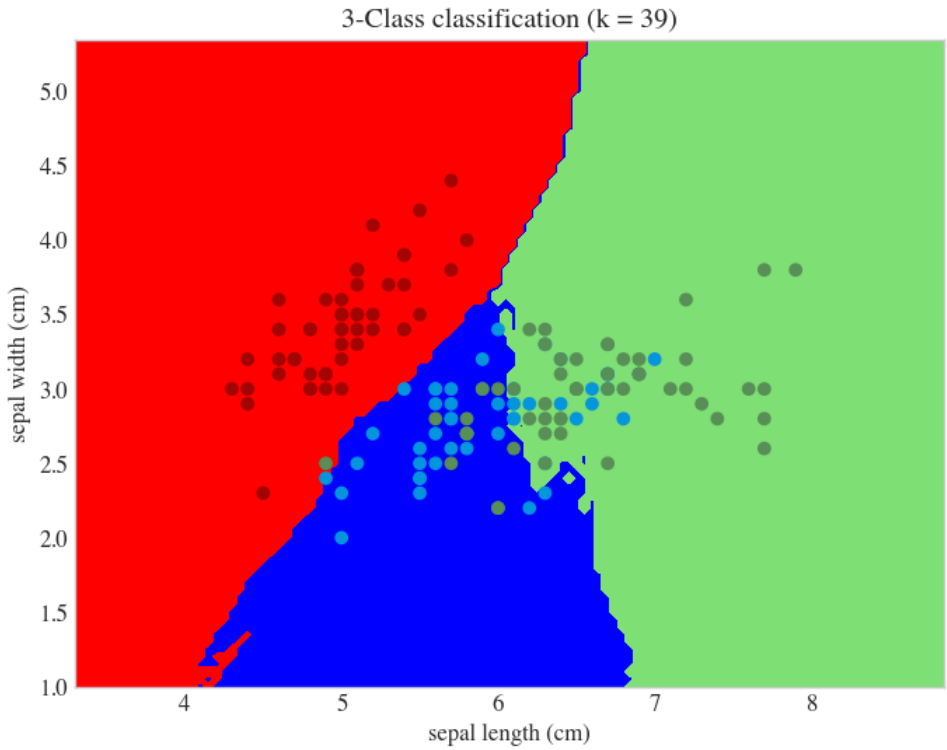

3-Class classification (k = 5)
sepal width (cm)
sepal length (cm)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
4
5
6
7
8

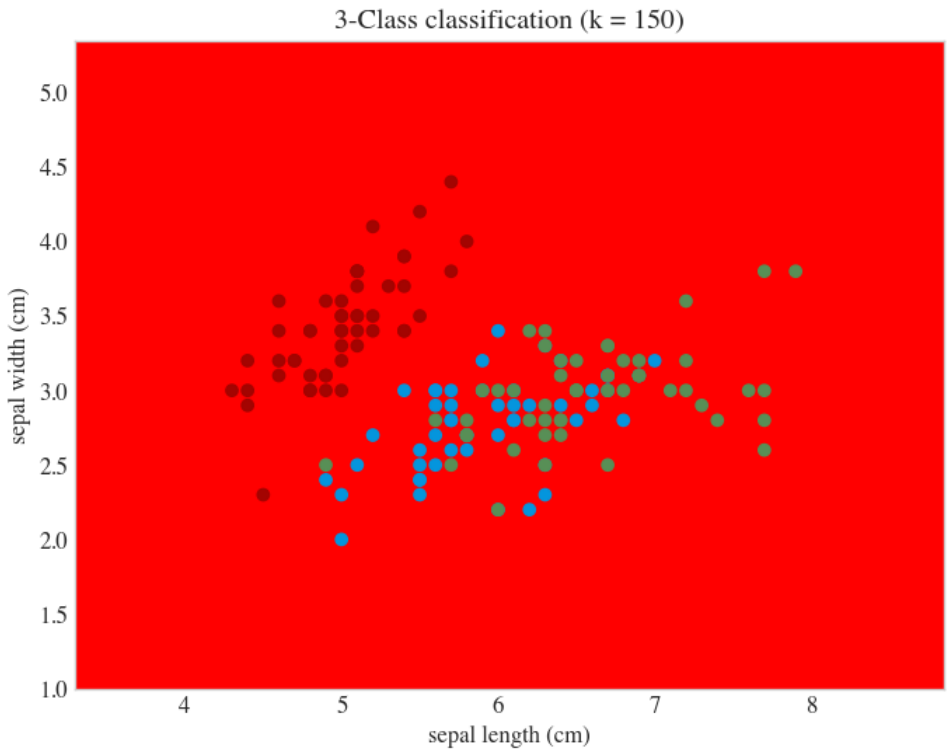

3-Class classification (k = 7)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
sepal width (cm)
4
5
6
7
8
sepal length (cm)


3-Class classification (k = 15)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
sepal width (cm)
4
5
6
7
8
sepal length (cm)


3-Class classification (k = 21)
sepal width (cm)
sepal length (cm)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
4
5
6
7
8


3-Class classification (k = 27)
sepal width (cm)
sepal length (cm)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
4
5
6
7
8

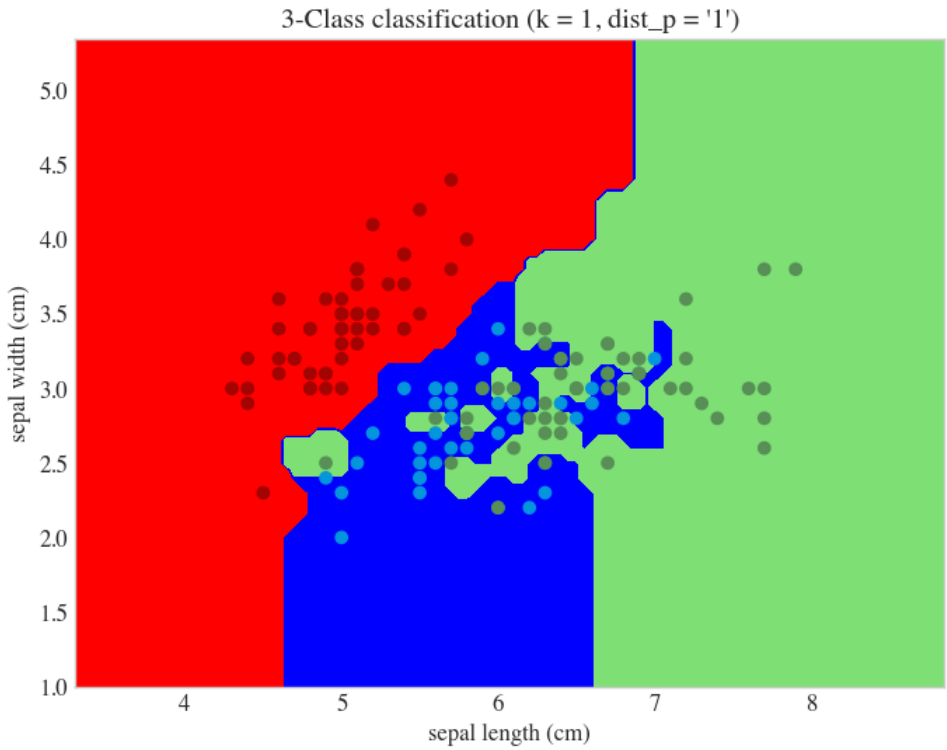


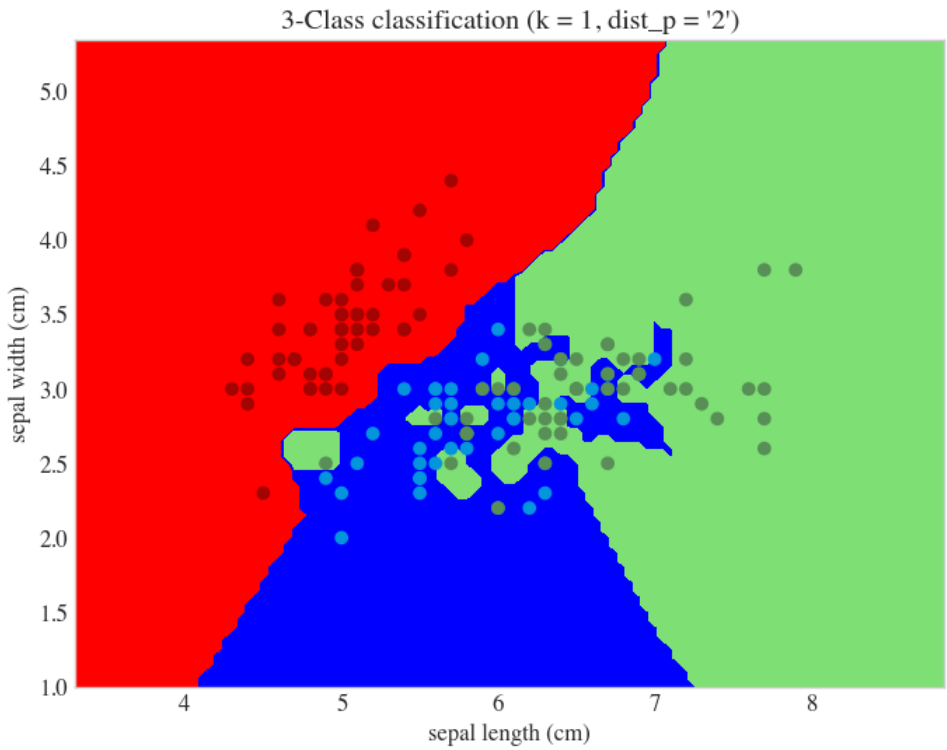


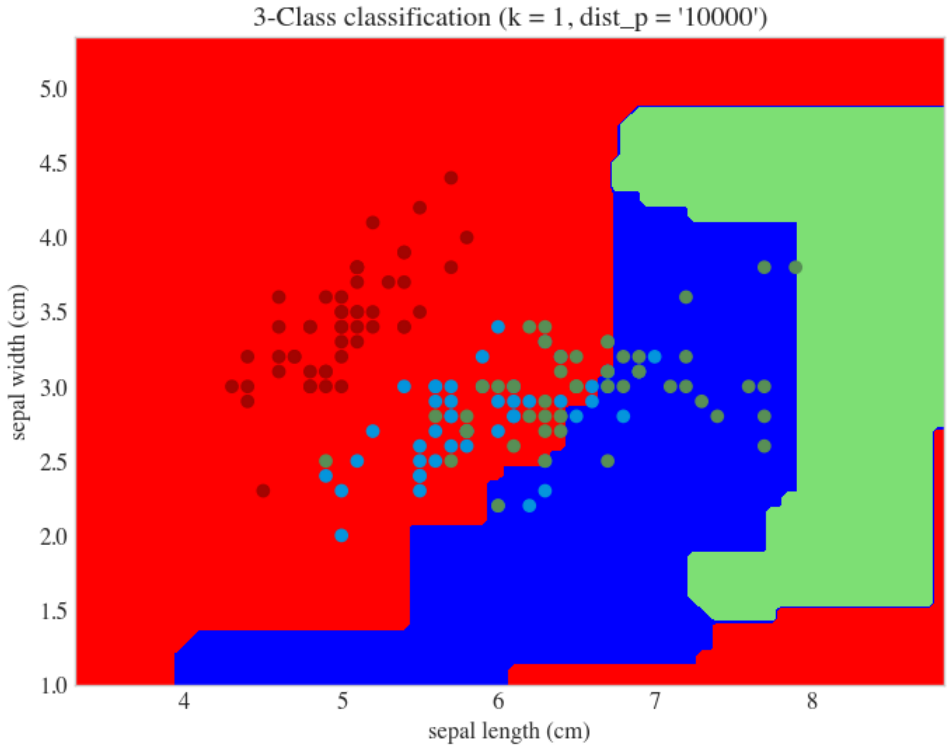
Decision boundaries by varying distance

In [7]:

```
from matplotlib.colors import ListedColormap
from sklearn import neighbors, datasets

#####################################################
h = 0.05  # step size in the mesh
n_neighbors = 1
#####################################################

# Create color maps
cm = ListedColormap(["#a30401", "#0495dd", "#588f56"])
cm_bright = ListedColormap(["#FF0000", "#0000FF", "#7ddf74"])


for dist in [1, 2, 10000]:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = neighbors.KNeighborsClassifier(
        n_neighbors, weights="uniform", algorithm='brute', p=dist)
    clf.fit(X, y)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    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, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure(figsize=(8, 6))
    plt.contourf(xx, yy, Z, cmap=cm_bright)

    # Plot also the training points
    plt.scatter(
        x=X[:, 0],
        y=X[:, 1],
        c=y,
        cmap=cm,
    )
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title(
        "3-Class classification (k = %i, dist_p = '%i')" % (n_neighbors, dist)
    )
    plt.xlabel(iris.feature_names[0])
    plt.ylabel(iris.feature_names[1])

plt.show()
```

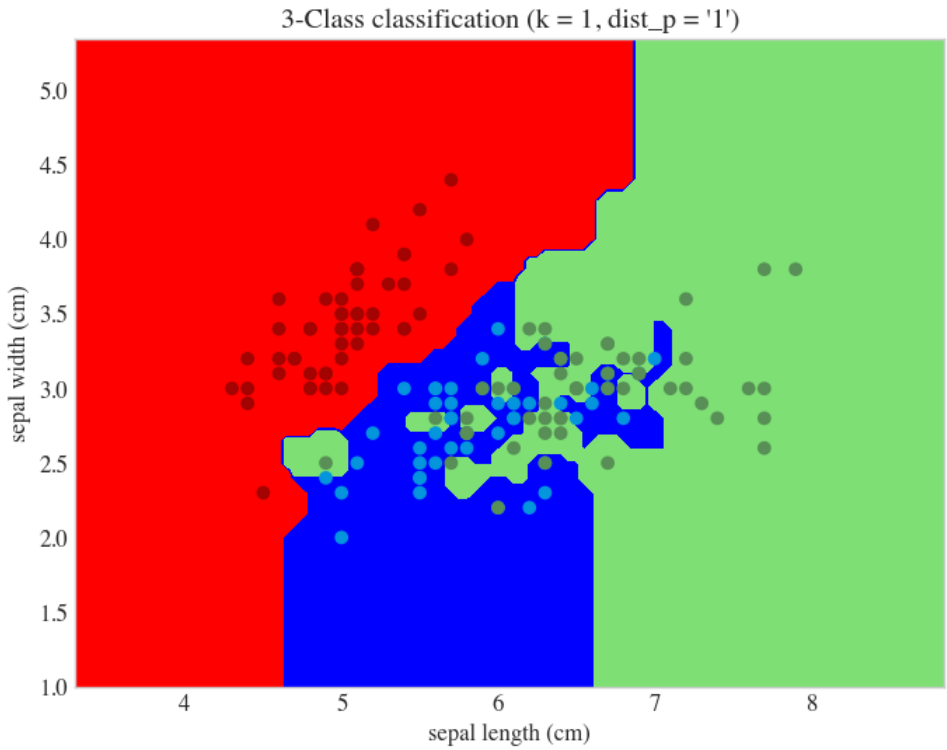


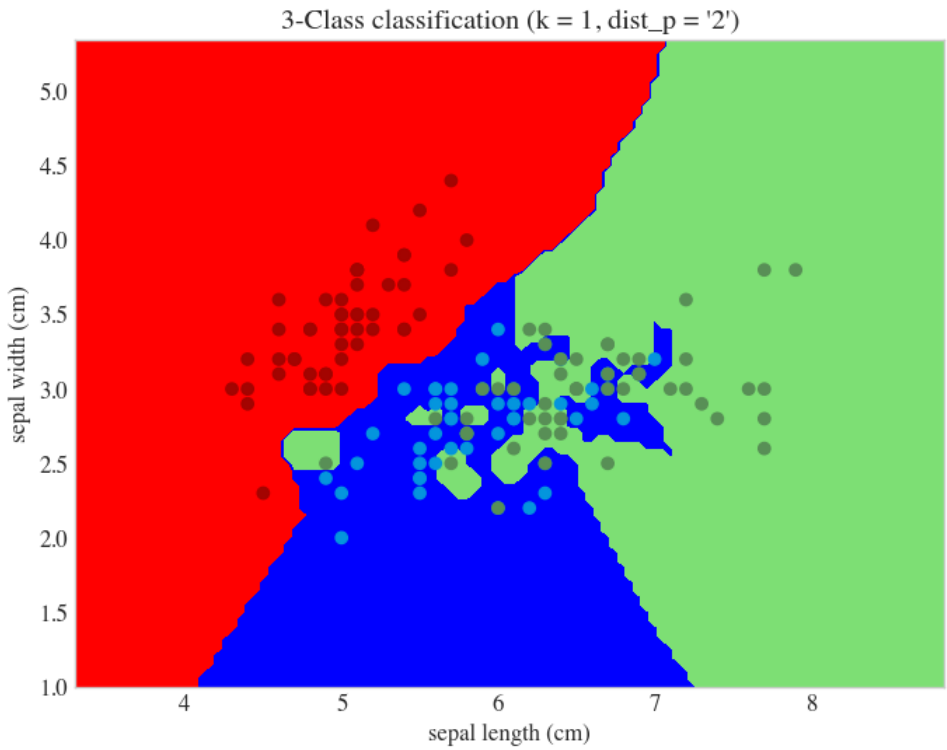

3-Class classification (k = 1, dist_p = '2')
sepal width (cm)
sepal length (cm)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
4
5
6
7
8

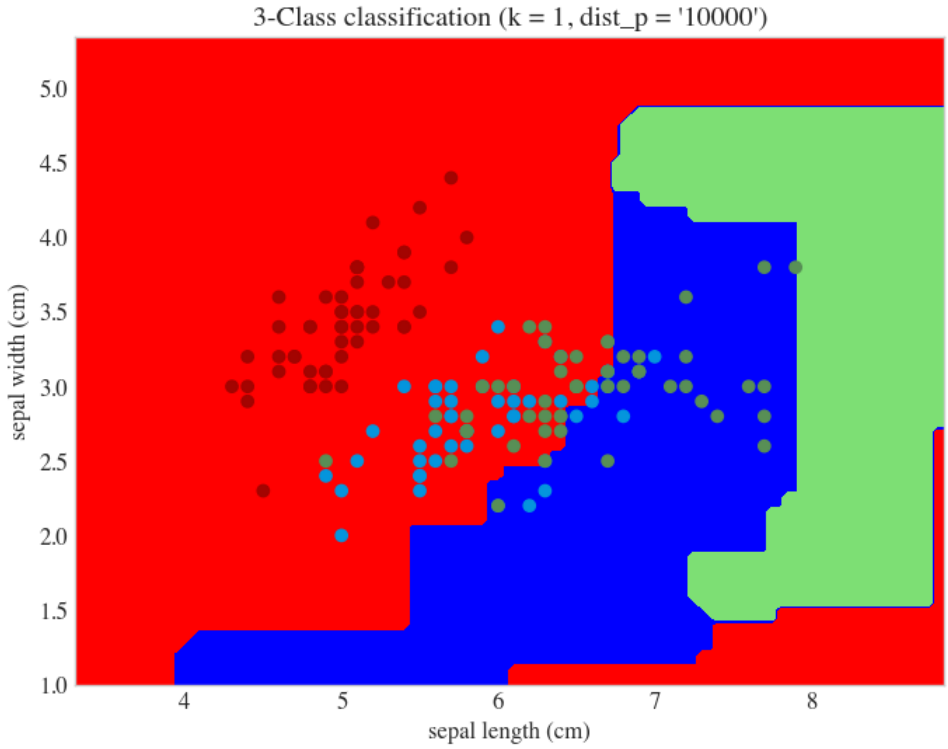

3-Class classification (k = 1, dist_p = '10000')
sepal width (cm)
sepal length (cm)
5.0
4.5
4.0
3.5
3.0
2.5
2.0
1.5
1.0
4
5
6
7
8

In [8]:

```
from matplotlib.colors import ListedColormap
from sklearn import neighbors, datasets

#####################################################
h = 0.05  # step size in the mesh
n_neighbors = 1
#####################################################

# Create color maps
cm = ListedColormap(["#a30401", "#0495dd", "#588f56"])
cm_bright = ListedColormap(["#FF0000", "#0000FF", "#7ddf74"])


for dist in [1, 2,10000]:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = neighbors.KNeighborsClassifier(
        n_neighbors, weights="uniform", algorithm='brute', p=dist)
    clf.fit(X, y)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    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, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure(figsize=(8, 6))
    plt.contourf(xx, yy, Z, cmap=cm_bright)

    # Plot also the training points
    plt.scatter(
        x=X[:, 0],
        y=X[:, 1],
        c=y,
        cmap=cm,
    )
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title(
        "3-Class classification (k = %i, dist_p = '%i')" % (n_neighbors, dist)
    )
    plt.xlabel(iris.feature_names[0])
    plt.ylabel(iris.feature_names[1])

plt.show()
```





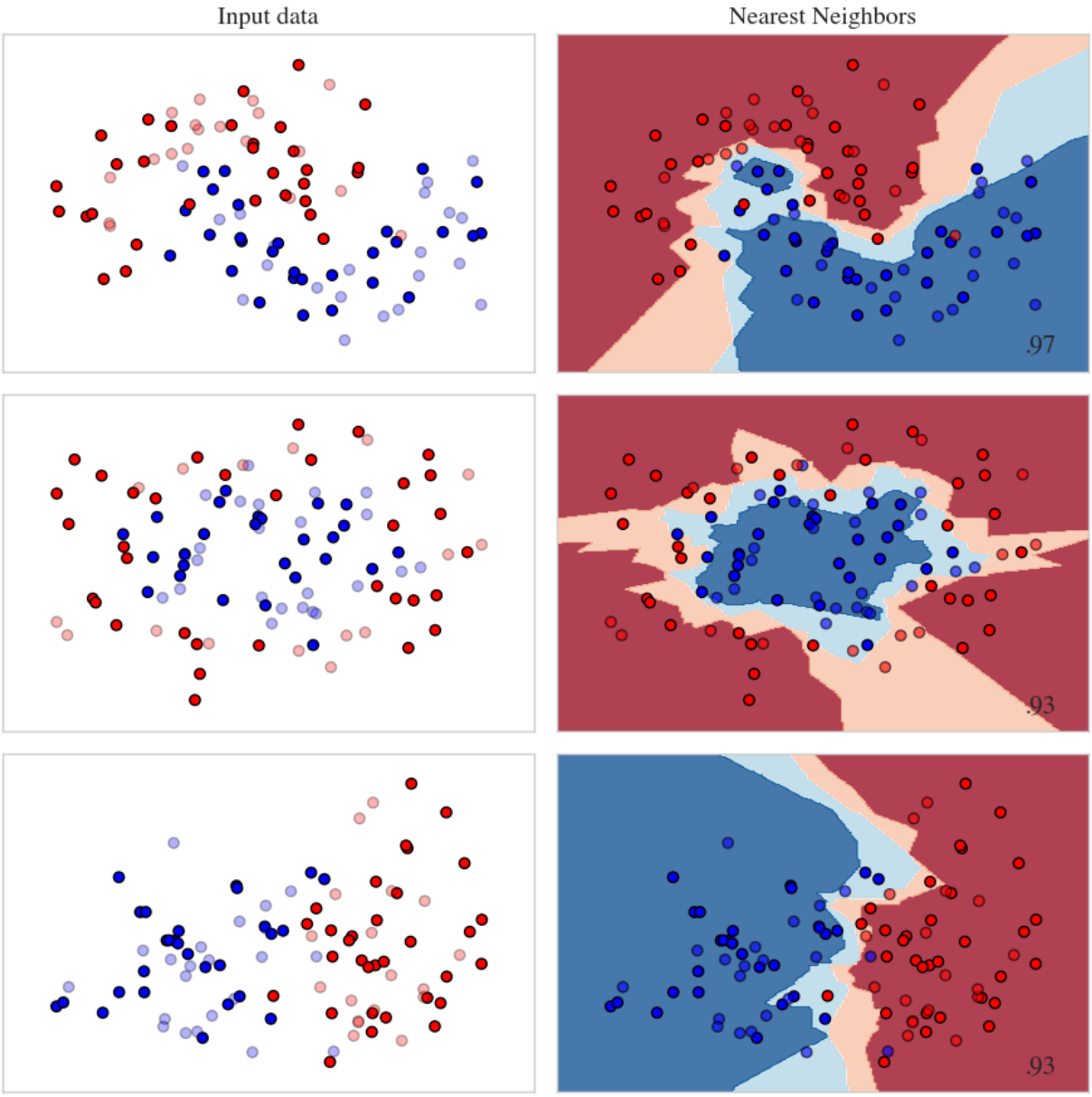


k-NN has irregular and non-linear decision boundaries

In [9]:

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neighbors import KNeighborsClassifier


h = 0.02  # step size in the mesh

names = [
    "Nearest Neighbors",
]

classifiers = [
    KNeighborsClassifier(3),
]

X, y = make_classification(
    n_features=2, n_redundant=0, n_informative=2, random_state=1, n_clusters_per_class=1
)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

datasets = [
    make_moons(noise=0.3, random_state=0),
    make_circles(noise=0.2, factor=0.5, random_state=1),
    linearly_separable,
]

figure = plt.figure(figsize=(9, 9))
i = 1
# iterate over datasets
for ds_cnt, ds in enumerate(datasets):
    # preprocess dataset, split into training and test part
    X, y = ds
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.4, random_state=42
    )

    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # just plot the dataset first
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(["#FF0000", "#0000FF"])
    ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
    if ds_cnt == 0:
        ax.set_title("Input data")
    # Plot the training points
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright, edgecolors="k")
    # Plot the testing points
    ax.scatter(
        X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.3, edgecolors="k"
    )
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())
    i += 1

    # iterate over classifiers
    for name, clf in zip(names, classifiers):
        ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
        clf.fit(X_train, y_train)
        score = clf.score(X_test, y_test)

        # Plot the decision boundary. For that, we will assign a color to each
        # point in the mesh [x_min, x_max]x[y_min, y_max].
        if hasattr(clf, "decision_function"):
            Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
        else:
            Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        ax.contourf(xx, yy, Z, cmap=cm, alpha=0.8)

        # Plot the training points
        ax.scatter(
            X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright, edgecolors="k"
        )
        # Plot the testing points
        ax.scatter(
            X_test[:, 0],
            X_test[:, 1]
```

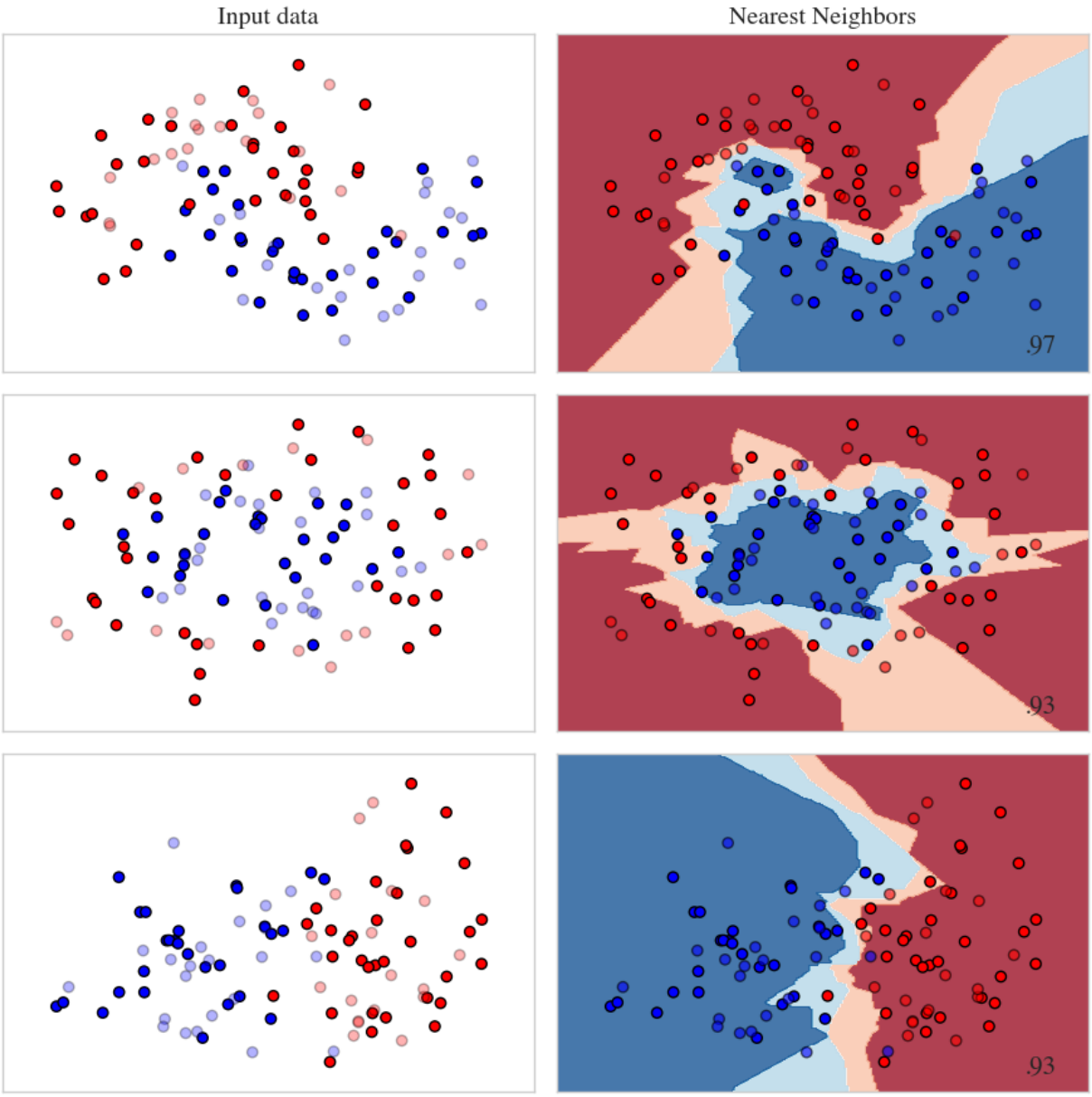
```
            x_test[:, 1],
            c=y_test,
            cmap=cm_bright,
            edgecolors="k",
            alpha=0.6,
        )

        ax.set_xlim(xx.min(), xx.max())
        ax.set_ylim(yy.min(), yy.max())
        ax.set_xticks(())
        ax.set_yticks(())
        if ds_cnt == 0:
            ax.set_title(name)
        ax.text(
            xx.max() - 0.3,
            yy.min() + 0.3,
            ("%.2f" % score).lstrip("0"),
            size=15,
            horizontalalignment="right",
        )
        i += 1

plt.tight_layout()
plt.show()
```



In [10]:

```python
import numpy as np
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_moons, make_circles, make_classification
from sklearn.neighbors import KNeighborsClassifier


h = 0.02  # step size in the mesh

names = [
    "Nearest Neighbors",
]

classifiers = [
    KNeighborsClassifier(3),
]

X, y = make_classification(
    n_features=2, n_redundant=0, n_informative=2, random_state=1, n_clusters_per_class=1
)
rng = np.random.RandomState(2)
X += 2 * rng.uniform(size=X.shape)
linearly_separable = (X, y)

datasets = [
    make_moons(noise=0.3, random_state=0),
    make_circles(noise=0.2, factor=0.5, random_state=1),
    linearly_separable,
]

figure = plt.figure(figsize=(9, 9))
i = 1
# iterate over datasets
for ds_cnt, ds in enumerate(datasets):
    # preprocess dataset, split into training and test part
    X, y = ds
    X = StandardScaler().fit_transform(X)
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.4, random_state=42
    )

    x_min, x_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5
    y_min, y_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # just plot the dataset first
    cm = plt.cm.RdBu
    cm_bright = ListedColormap(["#FF0000", "#0000FF"])
    ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
    if ds_cnt == 0:
        ax.set_title("Input data")
    # Plot the training points
    ax.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright, edgecolors="k")
    # Plot the testing points
    ax.scatter(
        X_test[:, 0], X_test[:, 1], c=y_test, cmap=cm_bright, alpha=0.3, edgecolors="k"
    )
    ax.set_xlim(xx.min(), xx.max())
    ax.set_ylim(yy.min(), yy.max())
    ax.set_xticks(())
    ax.set_yticks(())
    i += 1

    # iterate over classifiers
    for name, clf in zip(names, classifiers):
        ax = plt.subplot(len(datasets), len(classifiers) + 1, i)
        clf.fit(X_train, y_train)
        score = clf.score(X_test, y_test)

        # Plot the decision boundary. For that, we will assign a color to each
        # point in the mesh [x_min, x_max]x[y_min, y_max].
        if hasattr(clf, "decision_function"):
            Z = clf.decision_function(np.c_[xx.ravel(), yy.ravel()])
        else:
            Z = clf.predict_proba(np.c_[xx.ravel(), yy.ravel()])[:, 1]

        # Put the result into a color plot
        Z = Z.reshape(xx.shape)
        ax.contourf(xx, yy, Z, cmap=cm, alpha=0.8)

        # Plot the training points
        ax.scatter(
            X_train[:, 0], X_train[:, 1], c=y_train, cmap=cm_bright, edgecolors="k"
        )
        # Plot the testing points
        ax.scatter(
            X_test[:, 0],
            X_test[:, 1]
```

```
            x_test[:, 1],
            c=y_test,
            cmap=cm_bright,
            edgecolors="k",
            alpha=0.6,
        )

        ax.set_xlim(xx.min(), xx.max())
        ax.set_ylim(yy.min(), yy.max())
        ax.set_xticks(())
        ax.set_yticks(())
        if ds_cnt == 0:
            ax.set_title(name)
        ax.text(
            xx.max() - 0.3,
            yy.min() + 0.3,
            ("%.2f" % score).lstrip("0"),
            size=15,
            horizontalalignment="right",
        )
        i += 1

plt.tight_layout()
plt.show()
```



k-NN Issues and Remedies

Hint: Achilles' heel is the distance

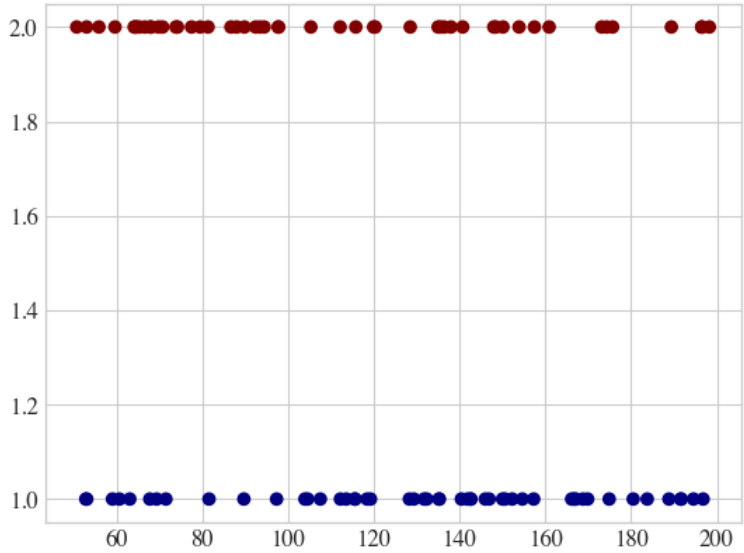
First issue: the feature range matters!

- So far we assumed we use **Euclidean distance** to find the nearest neighbor but:

$$d(\mathbf{x}, \mathbf{v}) = \sqrt{\sum_{i=1}^{D} (x_i - v_i)^2} \qquad \text{with } \mathbf{x}, \mathbf{v} \in \mathbb{R}^D$$

- Euclidean distance treats each feature as **equally important** (each axis in the vector)
- However **some features (dimensions) may be much more discriminative than other features**

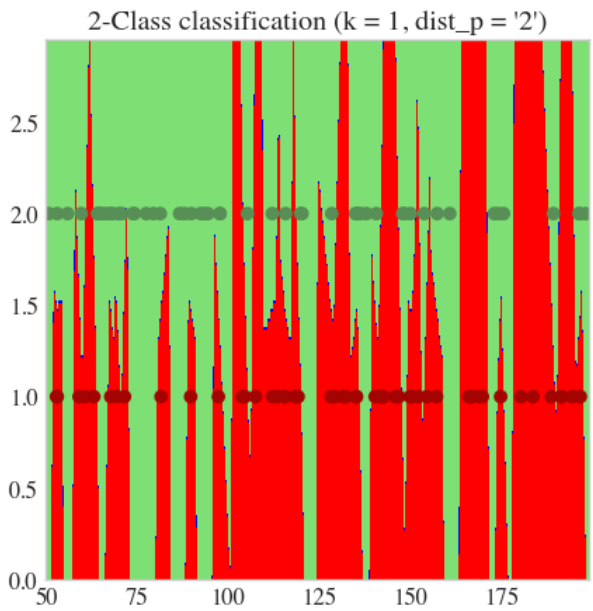
In [11]:
```
np.random.seed(0)
N_samples = 50
# samples points for class 1
X_1 = np.random.uniform(50, 200, N_samples)
X_1 = np.vstack((X_1, (1,)*N_samples))
# samples points for class 2
X_2 = np.random.uniform(50, 200, N_samples)
X_2 = np.vstack((X_2, (2,)*N_samples))
X = np.concatenate((X_1, X_2))
# data
X = np.concatenate((X_1, X_2), axis=1)
# labels
labels = X[1, ...]
# Plot also the training points
plt.scatter(
    x=X[0, ...],
    y=X[1, ...],
    c=labels,
    cmap='jet',
)
# Code below wants Nx2
X = X.T
```



Let's try to see what happens with k=1 so we make sure we fit well (overfit) the data

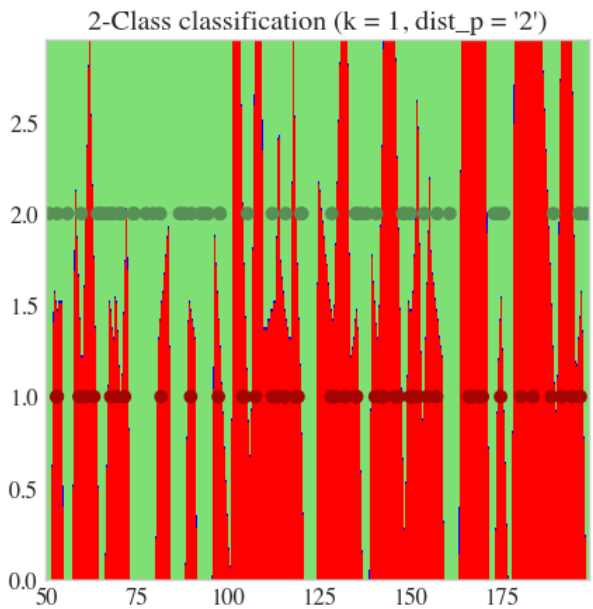
In [12]:

```
from matplotlib.colors import ListedColormap
from sklearn import neighbors, datasets

#####################################################
h = 0.05  # step size in the mesh
n_neighbors = 1
#####################################################

# Create color maps
cm = ListedColormap(["#a30401", "#0495dd", "#588f56"])
cm_bright = ListedColormap(["#FF0000", "#0000FF", "#7ddf74"])


for dist in [2]:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = neighbors.KNeighborsClassifier(
        n_neighbors, weights="uniform", algorithm='brute', p=dist)
    clf.fit(X, labels)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    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, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure(figsize=(5, 5))
    plt.contourf(xx, yy, Z, cmap=cm_bright)

    # Plot also the training points
    plt.scatter(
        x=X[:, 0],
        y=X[:, 1],
        c=labels,
        cmap=cm,
    )
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title(
        "2-Class classification (k = %i, dist_p = '%i')" % (n_neighbors, dist)
    )
plt.show()
```

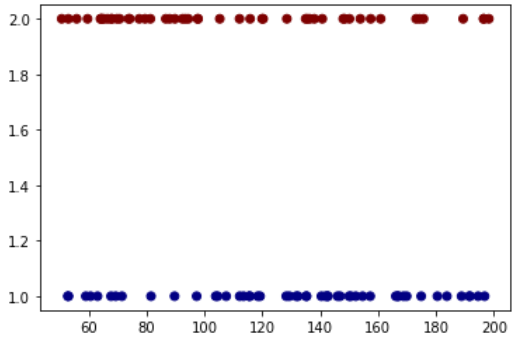


In [13]:

```
from matplotlib.colors import ListedColormap
from sklearn import neighbors, datasets

#####################################################
h = 0.05  # step size in the mesh
n_neighbors = 1
#####################################################

# Create color maps
cm = ListedColormap(["#a30401", "#0495dd", "#588f56"])
cm_bright = ListedColormap(["#FF0000", "#0000FF", "#7ddf74"])


for dist in [2]:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = neighbors.KNeighborsClassifier(
        n_neighbors, weights="uniform", algorithm='brute', p=dist)
    clf.fit(X, labels)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    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, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure(figsize=(5, 5))
    plt.contourf(xx, yy, Z, cmap=cm_bright)

    # Plot also the training points
    plt.scatter(
        x=X[:, 0],
        y=X[:, 1],
        c=labels,
        cmap=cm,
    )
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title(
        "2-Class classification (k = %i, dist_p = '%i')" % (n_neighbors, dist)
    )
plt.show()
```

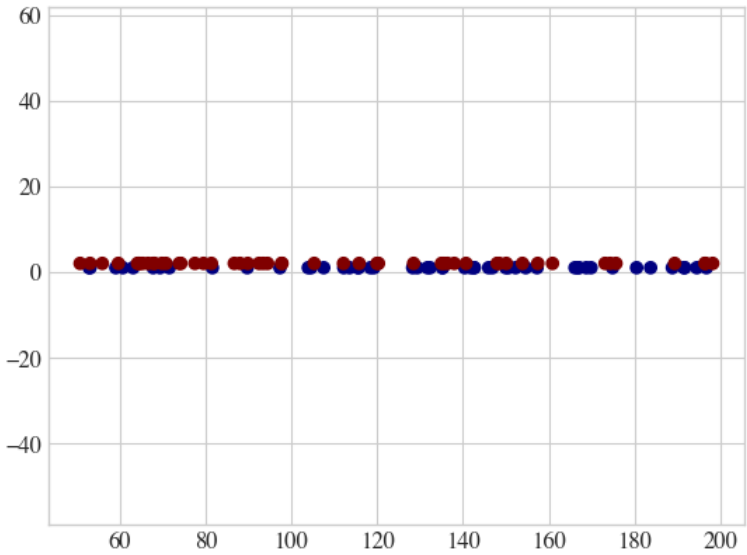


Who believes this is a good separation?



Let's look at the data with axis "equal"

In [14]:
```
# Plot also the training points
plt.scatter(
    x=X[...,0],
    y=X[...,1],
    c=labels,
    cmap='jet',
);
plt.axis('equal');
```



The previous decision boundary is really non-sense

- Feature on **first axis** has **no label information**, but its scale is **large**
- Feature on **second axis is discriminative** but its scale is **small**

The large scale of irrelevant feature dominates in the distance computation.

Feature 1 (on the x-axis of the plot)

Values around ~150

{{_ = plt.hist(X[..., 0], bins=20, range=(X.min(), X.max())) }}

Feature 2 (on the y-axis of the plot)

Values around 0 and 2

{{_ = plt.hist(X[..., 1], bins=20, range=(X.min(),5)) }}

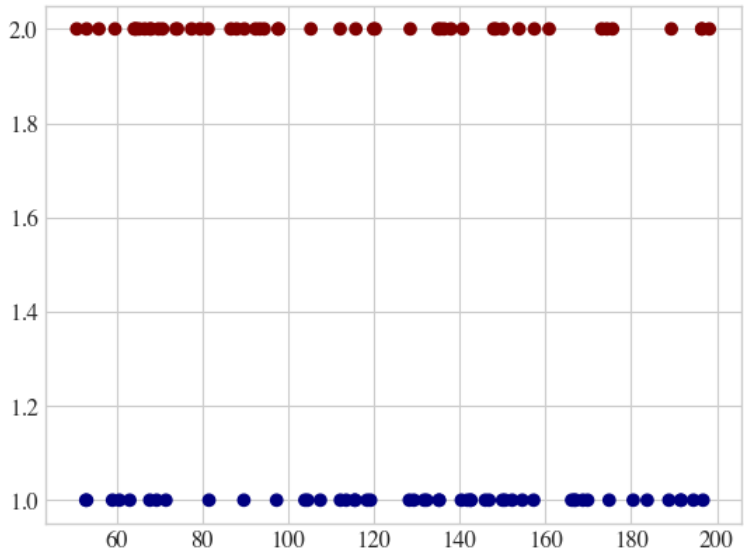
Idea: Normalize features to be on the same scale

Min-Max Normalization

$\forall$ dimension, **linearly** scales the features to be in the range $[0, 1]$

$$\mathbf{x}' = \frac{\mathbf{x} - \mathbf{x}_{\text{min}}}{\mathbf{x}_{\text{max}} - \mathbf{x}_{\text{min}}}$$

In [15]:
```
# Plot also the training points
plt.scatter(
    x=X[:, 0],
    y=X[:, 1],
    c=labels,
    cmap='jet',
)
# MinMax Normalization
maxX, minX = X.max(axis=0), X.min(axis=0)
X_old = np.copy(X)
X = (X - minX)/(maxX - minX)
```

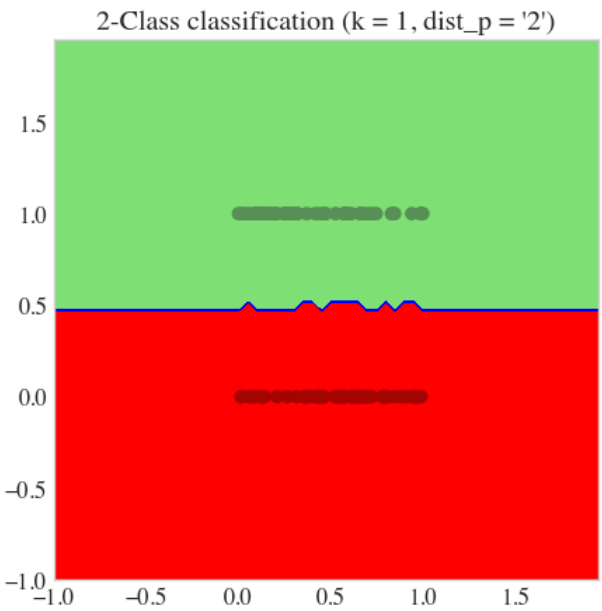


After (Focus on the Axis range)

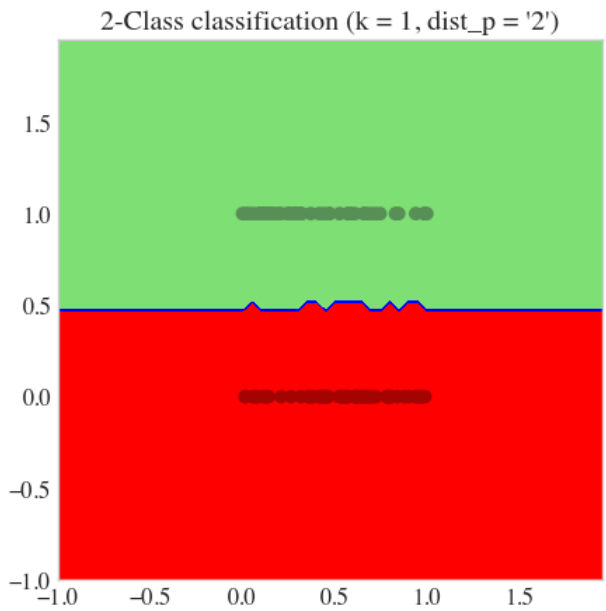
{{_=plt.scatter(x=X[:, 0],y=X[:, 1],c=labels, cmap='jet');plt.axis('equal');}}

In [16]:

```
from matplotlib.colors import ListedColormap
from sklearn import neighbors, datasets

#####################################################
h = 0.05  # step size in the mesh
n_neighbors = 1
#####################################################

# Create color maps
cm = ListedColormap(["#a30401", "#0495dd", "#588f56"])
cm_bright = ListedColormap(["#FF0000", "#0000FF", "#7ddf74"])


for dist in [2]:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = neighbors.KNeighborsClassifier(
        n_neighbors, weights="uniform", algorithm='brute', p=dist)
    clf.fit(X, labels)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    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, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure(figsize=(5, 5))
    plt.contourf(xx, yy, Z, cmap=cm_bright)

    # Plot also the training points
    plt.scatter(
        x=X[:, 0],
        y=X[:, 1],
        c=labels,
        cmap=cm,
    )
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title(
        "2-Class classification (k = %i, dist_p = '%i')" % (n_neighbors, dist)
    )
plt.show()
```



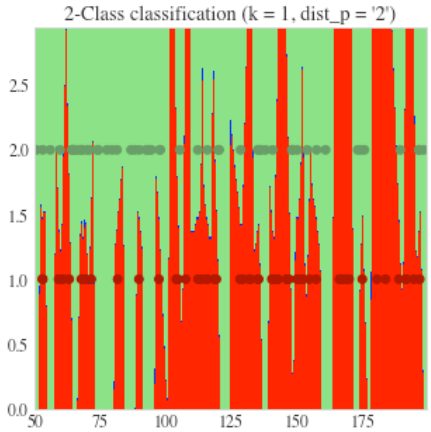
In [17]:
```
from matplotlib.colors import ListedColormap
from sklearn import neighbors, datasets

#####################################################
h = 0.05  # step size in the mesh
n_neighbors = 1
#####################################################

# Create color maps
cm = ListedColormap(["#a30401", "#0495dd", "#588f56"])
cm_bright = ListedColormap(["#FF0000", "#0000FF", "#7ddf74"])


for dist in [2]:
    # we create an instance of Neighbours Classifier and fit the data.
    clf = neighbors.KNeighborsClassifier(
        n_neighbors, weights="uniform", algorithm='brute', p=dist)
    clf.fit(X, labels)

    # Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
    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, h),
                         np.arange(y_min, y_max, h))
    Z = clf.predict(np.c_[xx.ravel(), yy.ravel()])

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.figure(figsize=(5, 5))
    plt.contourf(xx, yy, Z, cmap=cm_bright)

    # Plot also the training points
    plt.scatter(
        x=X[:, 0],
        y=X[:, 1],
        c=labels,
        cmap=cm,
    )
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
    plt.title(
        "2-Class classification (k = %i, dist_p = '%i')" % (n_neighbors, dist)
    )
plt.show()
```



Now I like it much more! Split along 0.5 on feature 2!

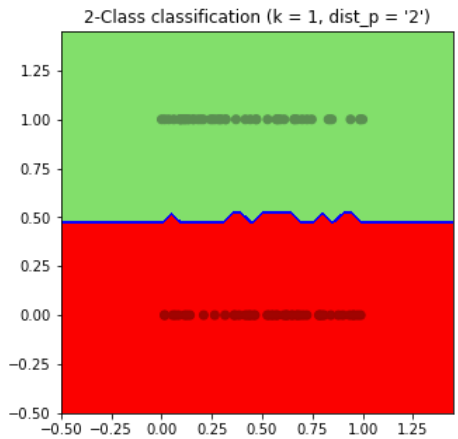
Take away: decision boundaries give you a way to assess learning

Very Fragmented decision boundaries $\longrightarrow$ probably overfit, model too specific



Regular decision boundaries $\longrightarrow$ probably underfit, model too generic.

In this case is a good solution since problem was easy.



Standardization

- Assumption that $\mathbf{X}$ is sampled from multi-variate Gaussian distribution
- **Center all the data in the center (origin)**. **Compute mean and remove it.**
- **Rescale all axis so that the standard deviation is one.**

$$\mathbf{X}' \leftarrow \frac{\mathbf{X} - \mu}{\sigma}$$

Feature Normalization in High-Dimensional Space $D \gg 1$

- Feature normalization does not help in high dimensional spaces if **most features are irrelevant**
- Assume that the dimensions (axis) are split in two set: those of **irrelevant features** $\mathcal{S}_{ir}$ and those of **good features** $\mathcal{S}_{gd}$.
- $D = \mid \mathcal{S}_{ir} \mid + \mid \mathcal{S}_{gd} \mid$
- if $\mid \mathcal{S}_{ir} \mid \gg \mid \mathcal{S}_{gd} \mid$, then **we have problems with k-NN**

$$d(\mathbf{x}, \mathbf{v}) = \sqrt{\sum_{i \in \mathcal{S}_{ir}} \underbrace{(x_i - v_i)^2}_{\text{dominate}} + \sum_{j \in \mathcal{S}_{gd}} \underbrace{(x_j - v_j)^2}_{\text{good yet not used}}} \qquad \text{with } \mathbf{x}, \mathbf{v} \in \mathbb{R}^D$$

- Eliminate some features or learn which features are important (**Metric Learning**)

Feature Weighting

We can try to learn a vector $\mathbf{w} = [w_1, \ldots, w_D]$ that scales each dimension in the Euclidean distance.

$$d(\mathbf{x}, \mathbf{v}) = \sqrt{\sum_{i=1}^{D} w_i (x_i - v_i)^2} \qquad \text{with } \mathbf{x}, \mathbf{v} \in \mathbb{R}^D$$

- Can use our prior knowledge about which features are more important
- Can learn the weights w_i using cross-validation (to be covered later)

Note: `sklearn` offers off-the-shelf normalization with the class `StandardScaler()`

Do not reinvent the wheel

k-NN and data embedded in a High Dimensional Space

k-NN and data embedded in a High Dimensional Space

- In high-dimension, points are concentrated at the **boundary of the space (hypercube)**
- If the entire space is the **Unit Hyper-cube** points will probably be at the edge. Why? #### What is the probability of a point to lie in the center in 1 dimension?

```
| eps | ------------- | eps |
0                           1
        length = 1
```

- Assuming points are sampled uniformly, prob. to be at the center is $p = (1 - 2\epsilon)$
- So if ϵ is small $p \approx 0.9$. In 1-D, for sure it is at the center.

k-NN and data embedded in a High Dimensional Space

- In high-dimension, points are concentrated at the **boundary of the space.**
- If the entire space is the **Unit Hyper-cube** points will probably be at the edge. Why? #### What is the probability of a point to lie in the center in 2 dimension?

```
0,0                          1,0
| eps | ------------- | eps |
|                           |
|                           |
|                           |
|                           |
|                           |
|                           |
| eps | ------------- | eps |
0,1        length = 1          1,1
```

- Assuming points are sampled uniformly, prob. to be at the center is $p = (1 - 2\epsilon)^2$
- So if ϵ is small $p \approx 0.81$. In 2-D, p decreases.

k-NN and data embedded in a High Dimensional Space

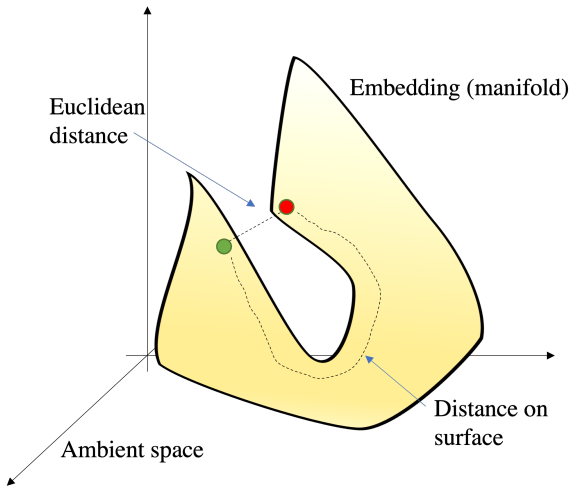
- In high-dimension, points are concentrated at the **boundary of the space.**
- If the entire space is the **Unit Hyper-cube** points will probably be at the edge. Why? #### What is the probability of a point to lie in the center in 128 dimension?

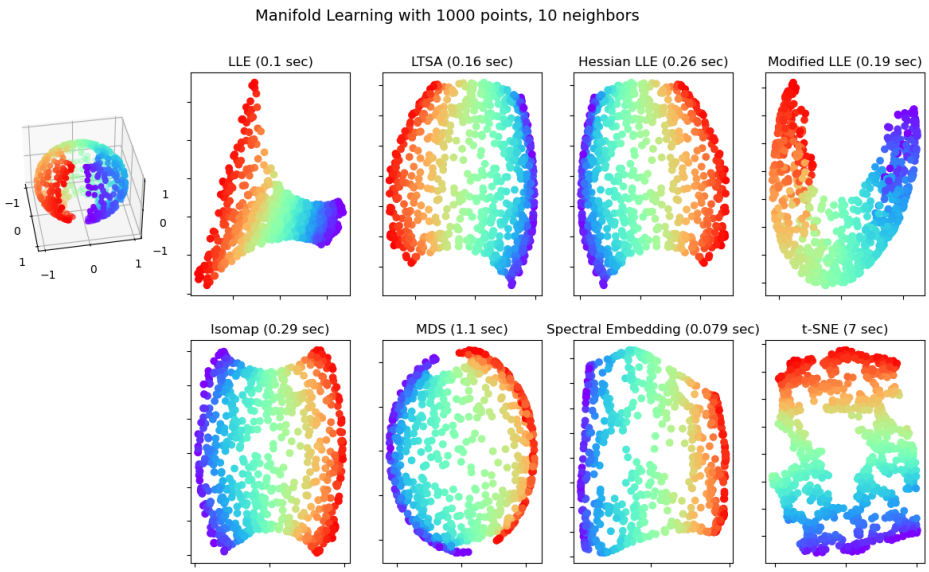
$p = (1 - 2\epsilon)^{128}$ = 0.9^128 = 0.000001390084524

If most of the points are at the boundaries, how can k-NN work?

Are we doomed?

Recall: High-dimensional data may live on a subspace or manifold





Wrap-up of k-NN

Complexity

Suppose we have N examples each of dimension D:

- $\mathcal{O}(D)$ to compute distance to one examples
- $\mathcal{O}(ND)$ to compute distances to all examples
- Plus time to find k closest examples

Trade-off:

- Very expensive for a large number of samples
- But we need a large number of samples for kNN to work well!

Reducing Complexity

Various exact and approximate methods for reducing complexity:

- Reduce dimensionality of the data (Feature Removal with cross-validation)
- Find projection to a lower dimensional space so that the distances between samples are approximated
- Use advanced data structures for fast search such as **K-D trees** and perform **ANN** (Approximate NN).

Advantages of k-NN

- **[Key assumptions] The output varies smoothly wrt the input**
- Can be applied to the data from any distribution **(even multi-modal):**
- **Complex** decision boundary that adapt to data density
- Very simple and intuitive
- Good classification if the number of samples is large enough

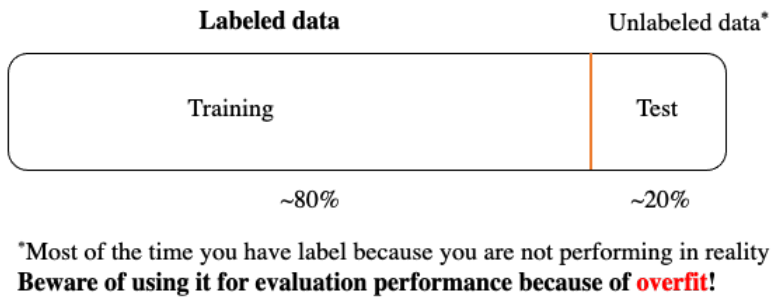
Disadvantages of k-NN

- Sensitive to label noise (Sol: smoothing)
- Sensitive to scales of attributes (sol: feature norm)
- Distances are less meaningful in high dimensions
- Scales linearly with number of examples

Validation set or development set

Ok we train on `train` split and we `test` on test split.

How do we select the distance and k neighbours?



Ok we train on `train` split and we `test` on test split.

How do we select the distance and k neighbours?