04_kNNs-SVM-RBF

June 19, 2022

CPSC 330
Applied Machine Learning

1 Lecture 4: k-Nearest Neighbours and SVM RBFs

UBC 2022 Summer

Instructor: Mehrdad Oveisi

> If two things are similar, the thought of one will tend to trigger the thought of the other
> – Aristotle

1.1 Imports

[1]:
```
import sys

import IPython
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from IPython.display import HTML

sys.path.append("code/.")

import ipywidgets as widgets
import mglearn
from IPython.display import display
from ipywidgets import interact, interactive
from plotting_functions import *
from sklearn.dummy import DummyClassifier
from sklearn.model_selection import cross_validate, train_test_split
from utils import *

%matplotlib inline
```

```python
pd.set_option("display.max_colwidth", 200)
import warnings

warnings.filterwarnings("ignore")
```

1.1.1 Quick recap

- Why do we split the data?
- What are the advantages of cross-validation?
- What is overfitting?
- What's the fundamental trade-off in supervised machine learning?
- What is the golden rule of machine learning?
- Types of data for our purposes
  - Categorical:
    - Nominal (sometimes just called *categorical*!), Ordinal
  - Numerical:
    - Discrete, Continuous

Package Installation If you want to run this notebook you will have to install `ipywidgets`.

Using the following two commands should suffice for our course environment:

```
conda install -n base -c conda-forge jupyterlab_widgets
conda install -n cpsc330 -c conda-forge ipywidgets
```

If you are still having problems or for more information, follow the installation instructions here.

1.2 Learning outcomes

From this lecture, you will be able to

- explain the notion of similarity-based algorithms;
- broadly describe how k-NNs use distances;
- discuss the effect of using a small/large value of the hyperparameter k when using the k-NN algorithm;
- describe the problem of curse of dimensionality;
- explain the general idea of SVMs with RBF kernel;
- broadly describe the relation of `gamma` and `C` hyperparameters of SVMs with the fundamental tradeoff.

1.3 Motivation and distances [video]

1.3.1 Analogy-based models

- Suppose you are given the following training examples with corresponding labels and are asked to label a given test example.

source

- An intuitive way to classify the test example is by finding the most "similar" example(s) from the training set and using that label for the test example.

1.3.2 Analogy-based algorithms in practice

- Recommendation systems
- Herta's High-tech Facial Recognition
  - Feature vectors for human faces
  - k-NN to identify which face is on their watch list

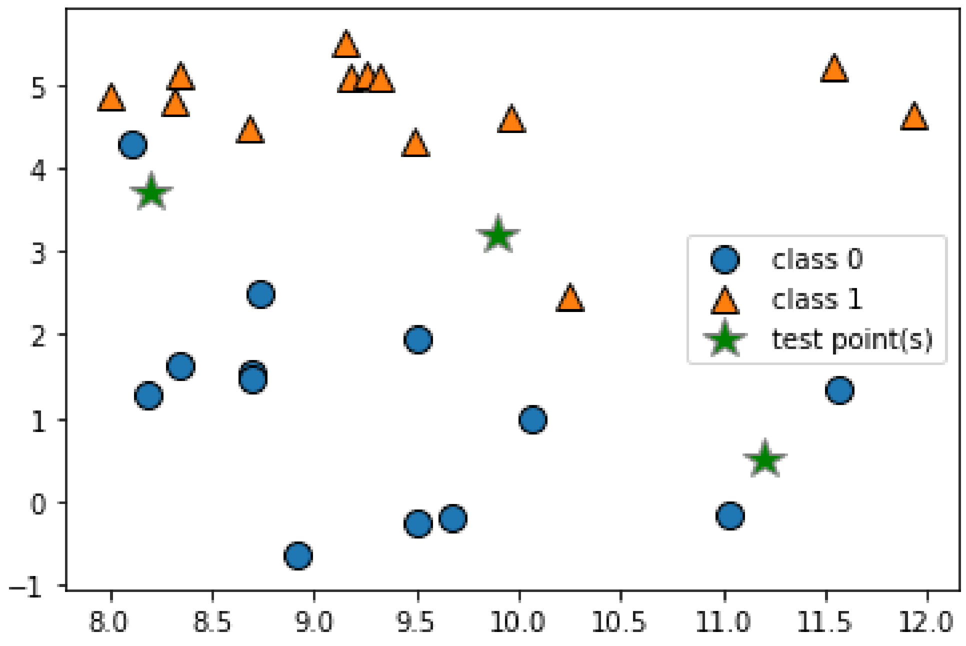
source

1.3.3 General idea of k-nearest neighbours algorithm

- Consider the following toy dataset with two classes.
  - blue circles $\rightarrow$ class 0
  - red triangles $\rightarrow$ class 1
  - green stars $\rightarrow$ test examples

[2]:
```
X, y = mglearn.datasets.make_forge()
X_test = np.array([[8.2, 3.7], [9.9, 3.2], [11.2, 0.5]])
```

[3]:
```
plot_train_test_points(X, y, X_test)
```



- Given a new data point, predict the class of the data point by finding the "closest" data point in the training set, i.e., by finding its "nearest neighbour" or majority vote of nearest neighbours.

[4]:
```
def f(n_neighbors):
    return plot_knn_clf(X, y, X_test, n_neighbors=n_neighbors)
```

```
[5]: interactive(
         f,
         n_neighbors=widgets.IntSlider(min=1, max=7, step=2, value=1),
     )
```

```
interactive(children=(IntSlider(value=1, description='n_neighbors', max=7,␣
↪min=1, step=2), Output()), _dom_cla…
```

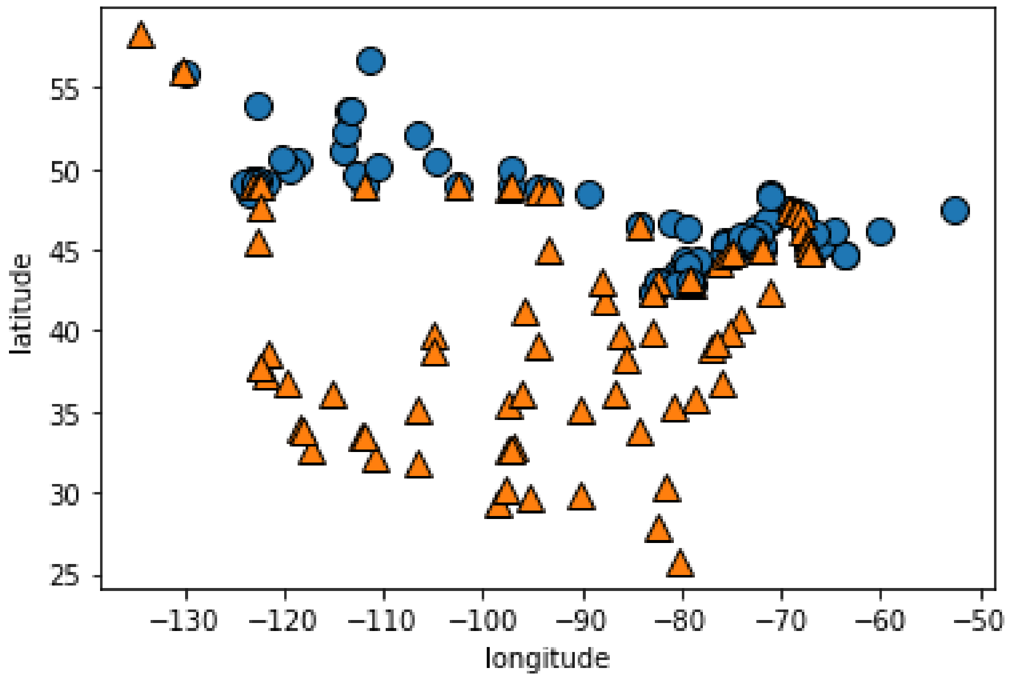
1.3.4 Geometric view of tabular data and dimensions

- To understand analogy-based algorithms it's useful to think of data as points in a high dimensional space.
- Our `X` represents the the problem in terms of relevant **features** (d) with one dimension for each **feature** (column).
- Examples are **points in a d-dimensional space**.

How many dimensions (features) are there in the cities data?

```
[6]: cities_df = pd.read_csv("data/canada_usa_cities.csv")
     X_cities = cities_df[["longitude", "latitude"]]
     y_cities = cities_df["country"]
```

```
[7]: mglearn.discrete_scatter(X_cities.iloc[:, 0], X_cities.iloc[:, 1], y_cities)
     plt.xlabel("longitude")
     plt.ylabel("latitude");
```



- Recall the Spotify Song Attributes dataset from homework 1.

- How many dimensions (features) we used in the homework?

```
[8]: spotify_df = pd.read_csv("data/spotify.csv", index_col=0)
     X_spotify = spotify_df.drop(columns=["target", "song_title", "artist"])
     print("The number of features in the Spotify dataset: %d" % X_spotify.shape[1])
     X_spotify.head()
```

```
The number of features in the Spotify dataset: 13
```

```
[8]:    acousticness  danceability  duration_ms  energy  instrumentalness  key  \
     0        0.0102         0.833       204600   0.434          0.021900    2
     1        0.1990         0.743       326933   0.359          0.006110    1
     2        0.0344         0.838       185707   0.412          0.000234    2
     3        0.6040         0.494       199413   0.338          0.510000    5
     4        0.1800         0.678       392893   0.561          0.512000    5

        liveness  loudness  mode  speechiness    tempo  time_signature  valence
     0    0.1650    -8.795     1       0.4310  150.062             4.0    0.286
     1    0.1370   -10.401     1       0.0794  160.083             4.0    0.588
     2    0.1590    -7.148     1       0.2890   75.044             4.0    0.173
     3    0.0922   -15.236     1       0.0261   86.468             4.0    0.230
     4    0.4390   -11.648     0       0.0694  174.004             4.0    0.904
```

1.3.5 Dimensions in ML problems

In ML, usually we deal with high dimensional problems where examples are hard to visualize.

- $d \approx 20$ is considered low dimensional
- $d \approx 1000$ is considered medium dimensional
- $d \approx 100,000$ is considered high dimensional

1.3.6 Feature vectors

Feature vector is composed of feature values associated with an example.

They correspond to rows of our dataframes.

Some example feature vectors are shown below.

```
[9]: print("An example feature vector from the cities dataset: %s"
           % (X_cities.iloc[0].to_numpy()))
     print("An example feature vector from the Spotify dataset: \n%s"
           % (X_spotify.iloc[0].to_numpy()))
```

```
An example feature vector from the cities dataset: [-130.0437   55.9773]
An example feature vector from the Spotify dataset:
[ 1.02000e-02  8.33000e-01  2.04600e+05  4.34000e-01  2.19000e-02
  2.00000e+00  1.65000e-01 -8.79500e+00  1.00000e+00  4.31000e-01
  1.50062e+02  4.00000e+00  2.86000e-01]
```

1.3.7 Similarity between examples

Let's take 2 points (two feature vectors) from the cities dataset.

[10]:
```
two_cities = X_cities.sample(2, random_state=120)
two_cities
```

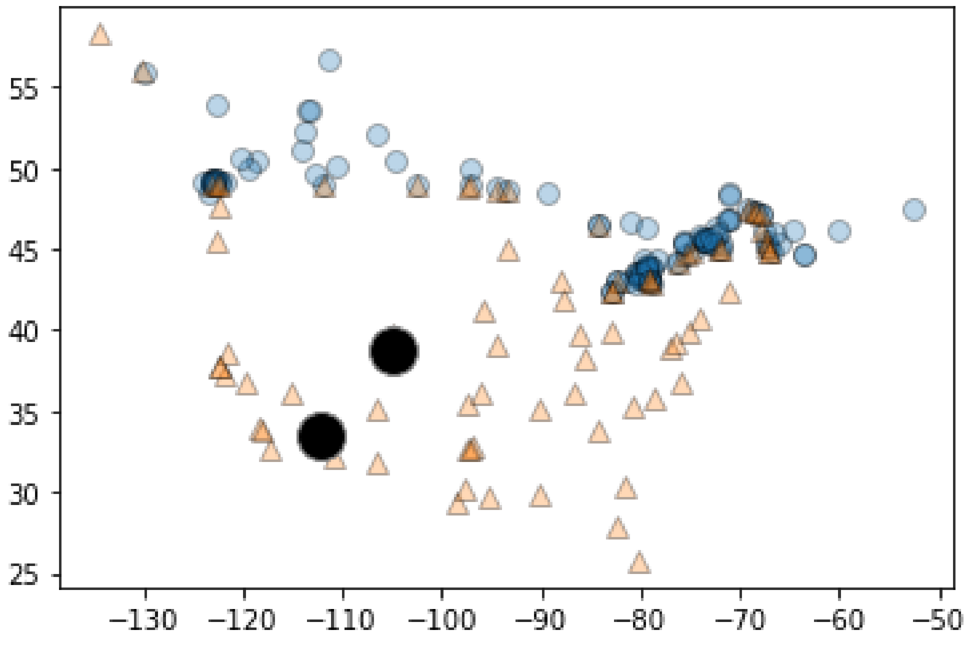
[10]:
```
    longitude  latitude
69  -104.8253   38.8340
35  -112.0741   33.4484
```

[11]:
```
# also recall that y was the country for each city
y_cities.unique()
```

[11]:
```
array(['USA', 'Canada'], dtype=object)
```

The two sampled points are shown as big black circles.

[12]:
```
mglearn.discrete_scatter(
    X_cities.iloc[:, 0], X_cities.iloc[:, 1], y_cities, s=8, alpha=0.3
)
mglearn.discrete_scatter(
    two_cities.iloc[:, 0], two_cities.iloc[:, 1], markers="o", c="k", s=18
);
```



1.3.8 Distance between feature vectors

- For the `two_cities` at the two big circles, what is the *distance* between them?

- A common way to calculate the distance between vectors is calculating the **Euclidean distance**.
- The euclidean distance
  - in one dimension between u and v:

$$distance(u,v) = \sqrt{(u-v)^2} = |u-v|$$

  - in two dimensions between $u =< u_1, u_2 >$ and $v =< v_1, v_2 >$:

$$distance(u,v) = \sqrt{(u_1-v_1)^2 + (u_2-v_2)^2}$$

  - generally, in higher dimensions between vectors $u =< u_1, u_2, \dots, u_n >$ and $v =< v_1, v_2, \dots, v_n >$ is defined as:

$$distance(u,v) = \sqrt{\sum_{i=1}^{n}(u_i - v_i)^2}$$

1.3.9 Euclidean distance

[13]:
```
two_cities
```

[13]:
```
     longitude  latitude
69   -104.8253   38.8340
35   -112.0741   33.4484
```

- Subtract the two cities
- Square the difference
- Sum them up
- Take the square root

[14]:
```
# Subtract the two cities
print("Subtract the cities: \n%s\n"
      % (two_cities.iloc[1] - two_cities.iloc[0]))

# Squared sum of the difference
print("Sum of squares: %0.4f"
      % (np.sum((two_cities.iloc[1] - two_cities.iloc[0]) ** 2)))

# Take the square root
print("Euclidean distance between cities: %0.4f"
      % (np.sqrt(np.sum((two_cities.iloc[1] - two_cities.iloc[0]) ** 2))))
```

```
Subtract the cities:
longitude   -7.2488
latitude    -5.3856
dtype: float64

Sum of squares: 81.5498
Euclidean distance between cities: 9.0305
```

[15]:
```
two_cities
```

[15]:
```
    longitude  latitude
69  -104.8253   38.8340
35  -112.0741   33.4484
```

[16]:
```
# Euclidean distance using sklearn
from sklearn.metrics.pairwise import euclidean_distances

euclidean_distances(two_cities)
```

[16]:
```
array([[0.        , 9.03049217],
       [9.03049217, 0.        ]])
```

Note: `scikit-learn` supports a number of other distance metrics.

1.3.10 Finding the nearest neighbour

- Let's look at distances from all cities to all other cities

[17]:
```
dists = euclidean_distances(X_cities)
np.fill_diagonal(dists, np.inf)  # why is this needed?
print("All distances: %s\n\n%s" % (dists.shape, dists))
```

```
All distances: (209, 209)

[[        inf  4.95511263  9.869531   … 52.42640992 58.03345923
  51.49856241]
 [ 4.95511263         inf 14.6775792  … 57.25372435 62.77196948
  56.25216034]
 [ 9.869531   14.6775792          inf … 44.23515175 50.24972011
  43.69922405]
 …
 [52.42640992 57.25372435 44.23515175 …         inf  6.83784786
   3.32275537]
 [58.03345923 62.77196948 50.24972011 …  6.83784786         inf
   6.55573969]
 [51.49856241 56.25216034 43.69922405 …  3.32275537  6.55573969
          inf]]
```

Let's look at the distances between City 0 and some other cities.

[18]:
```
print("Feature vector for city 0: \n%s\n" % (X_cities.iloc[0]))
print("Distances from city 0 to the first 5 cities: \n%s\n" % (dists[0,:5]))
# We can find the closest city with `np.argmin`:
print("The closest city from city 0 is: %d \nwith feature vector: \n%s"
      % (np.argmin(dists[0]), X_cities.iloc[np.argmin(dists[0])]))
```

```
Feature vector for city 0:
longitude   -130.0437
```

```
latitude      55.9773
Name: 0, dtype: float64

Distances from city 0 to the first 5 cities:
[        inf  4.95511263  9.869531   10.10645223 10.44966612]

The closest city from city 0 is: 81
with feature vector:
longitude   -129.9912
latitude      55.9383
Name: 81, dtype: float64
```

So, we managed to find out the closest city to City 0, which is City 81.

1.3.11 Question

- Why did we set the diagonal entries to infinity before finding the closest city? So they don't match to themselves.

1.3.12 Finding the distances to a query point

We can also find the distances to a new "test" or "query" city:

[19]:
```python
# Let's find a city that's closest to the a query city
query_point = [[-80, 25]]

dists = euclidean_distances(X_cities, query_point)
dists[0:10]
```

[19]:
```
array([[58.85545875],
       [63.80062924],
       [49.30530902],
       [49.01473536],
       [48.60495488],
       [39.96834506],
       [32.92852376],
       [29.53520104],
       [29.52881619],
       [27.84679073]])
```

[20]:
```python
print("The query point %s is closest to the city with index %d \n"
      "and the distance between them is: %0.4f"
      % (query_point, np.argmin(dists), dists[np.argmin(dists)]))
```

```
The query point [[-80, 25]] is closest to the city with index 72
and the distance between them is: 0.7982
```

[21]:
```python
# See the closest city (72) among some other cities with thir distances to␣
 ↪query point
```

```
X_cities.join(pd.DataFrame(dists, columns=['dist'])).head(np.argmin(dists) + 3).
 ↪tail()
```

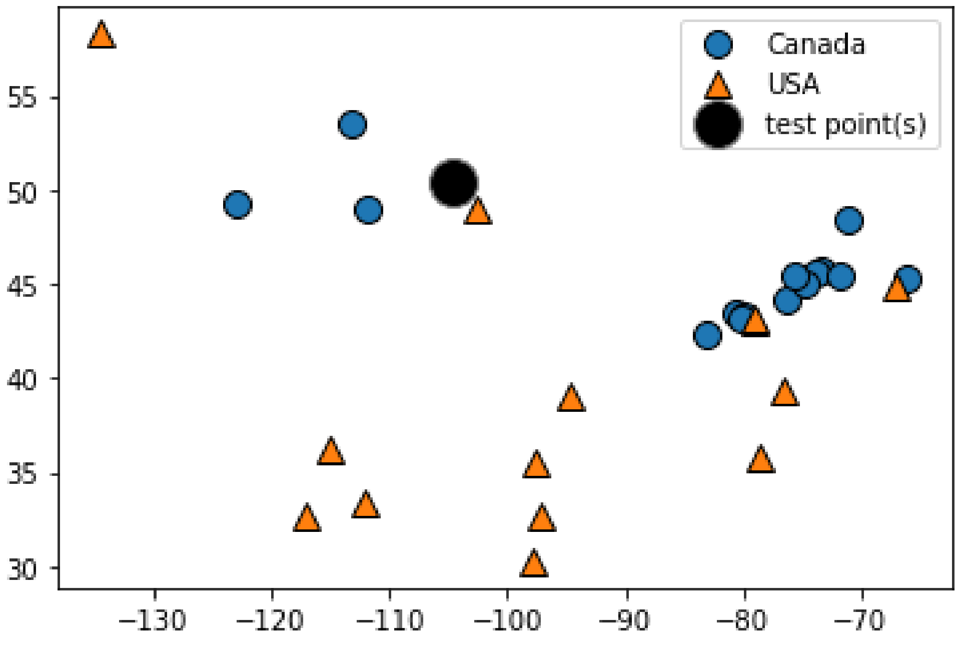
[21]:
```
     longitude  latitude       dist
70   -95.9384   41.2587  22.767914
71   -78.6391   35.7804  10.865959
72   -80.1937   25.7743   0.798160
73  -118.1916   33.7690  39.185376
74   -75.9774   36.8530  12.516985
```

1.4 *k*-Nearest Neighbours (*k*-NNs) [video]

[22]:
```
small_cities = cities_df.sample(30, random_state=90)
one_city = small_cities.sample(1, random_state=44)
# get all of small_cities excluding one_city:
small_train_df = pd.concat([small_cities, one_city]).drop_duplicates(keep=False)
```

[23]:
```
X_small_cities = small_train_df[["longitude", "latitude"]].to_numpy()
y_small_cities = small_train_df["country"].to_numpy()
test_point = one_city[["longitude", "latitude"]].to_numpy()
```

[24]:
```
plot_train_test_points(
    X_small_cities,
    y_small_cities,
    test_point,
    class_names=["Canada", "USA"],
    test_format="circle",
)
```

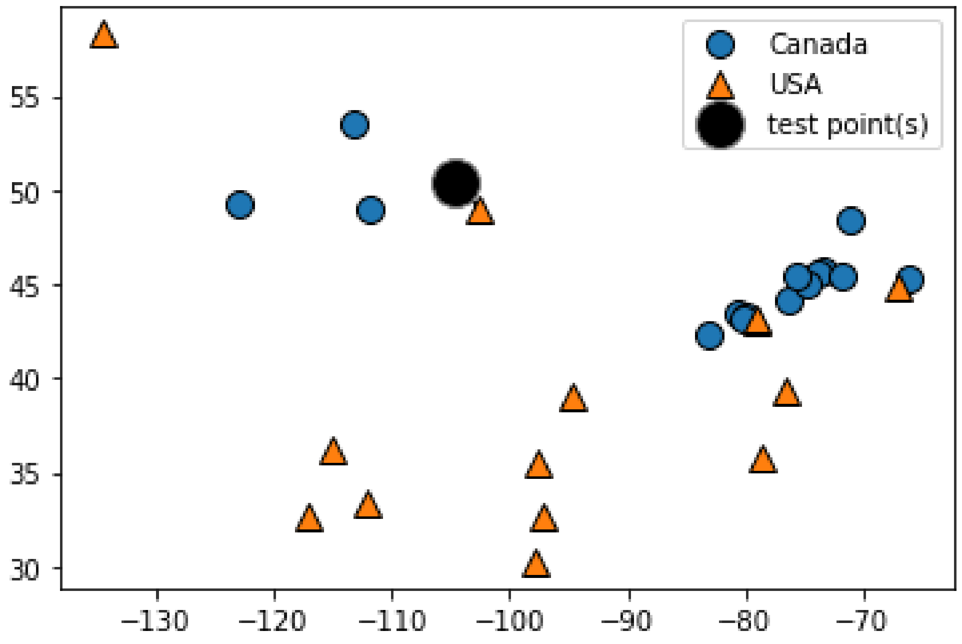


- Given a new data point, predict the class of the data point by finding the "closest" data point in the training set, i.e., by finding its "nearest neighbour" or majority vote of nearest neighbours.

Suppose we want to predict the class of the black point.
- An intuitive way to do this is predict the same label as the "closest" point ($k = 1$) (1-nearest neighbour) - We would predict a target of **USA** in this case.

[25]:
```
plot_knn_clf(
    X_small_cities,
    y_small_cities,
    test_point,
    n_neighbors=1,
    class_names=["Canada", "USA"],
    test_format="circle",
)
```

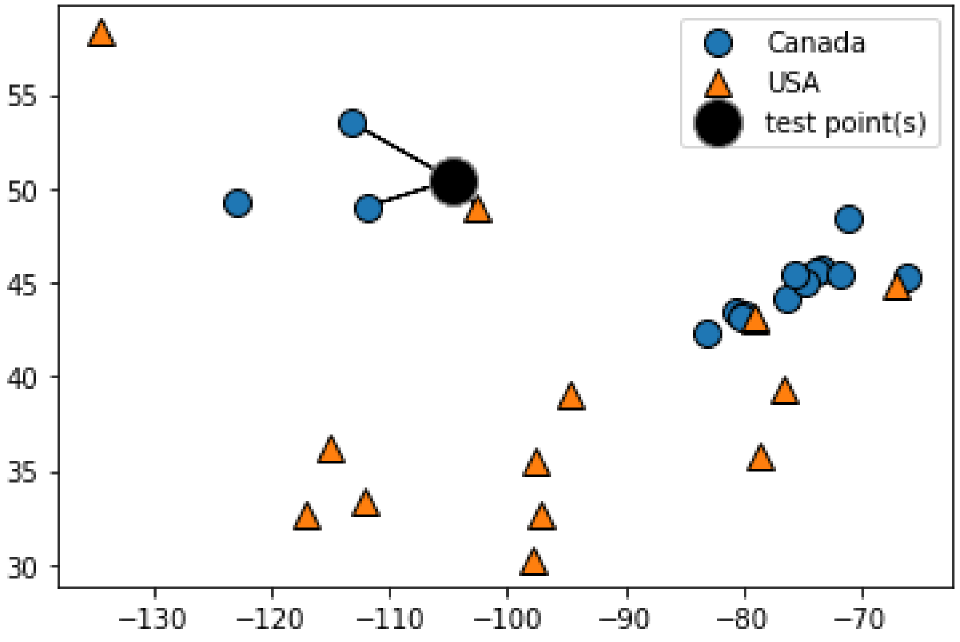
```
n_neighbors 1
```



How about using $k > 1$ to get a more robust estimate? - For example, we could also use the 3 closest points ($k = 3$) and let them **vote** on the correct class.
- The **Canada** class would win in this case.

[26]:
```
plot_knn_clf(
    X_small_cities,
    y_small_cities,
    test_point,
    n_neighbors=3,
    class_names=["Canada", "USA"],
    test_format="circle",
)
```

```
n_neighbors 3
```



[27]:
```python
from sklearn.neighbors import KNeighborsClassifier

k_values = [1, 3]

for k in k_values:
    neigh = KNeighborsClassifier(n_neighbors=k)
    neigh.fit(X_small_cities, y_small_cities)
    print("Prediction of the black dot with %d neighbours: %s"
          % (k, neigh.predict(test_point)))
```

```
Prediction of the black dot with 1 neighbours: ['USA']
Prediction of the black dot with 3 neighbours: ['Canada']
```

1.4.1 Questions

- Is it a good or a bad idea to consider an odd number for k? Why or why not? Good idea. There is no tie case in a binary classification.
- Try different values of k in the above code.

1.4.2 Choosing `n_neighbors`

- The **primary hyperparameter** of the model is `n_neighbors` (k) which decides how many neighbours should vote during prediction?
- What happens when we play around with `n_neighbors`?
- Are we more likely to overfit with a low `n_neighbors` or a high `n_neighbors`?
- Let's examine the effect of the hyperparameter on our cities data.

[28]:
```
X = cities_df.drop(columns=["country"])
y = cities_df["country"]

# split into train and test sets
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.1, random_state=123
)
```

[29]:
```
k = 1
knn1 = KNeighborsClassifier(n_neighbors=k)
scores = cross_validate(knn1, X_train, y_train, return_train_score=True)
pd.DataFrame(scores)
```

[29]:
```
   fit_time  score_time  test_score  train_score
0  0.001951    0.005579    0.710526          1.0
1  0.002087    0.003500    0.684211          1.0
2  0.002119    0.002406    0.842105          1.0
3  0.001943    0.002858    0.702703          1.0
4  0.001600    0.002084    0.837838          1.0
```

[30]:
```
k = 100
knn100 = KNeighborsClassifier(n_neighbors=k)
scores = cross_validate(knn100, X_train, y_train, return_train_score=True)
pd.DataFrame(scores)
```

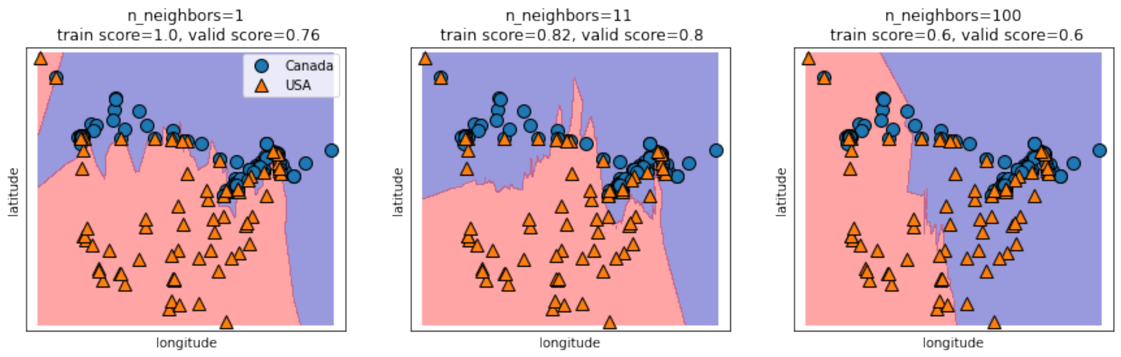
[30]:
```
   fit_time  score_time  test_score  train_score
0  0.003005    0.005593    0.605263     0.600000
1  0.001776    0.004325    0.605263     0.600000
2  0.002361    0.003898    0.605263     0.600000
3  0.002631    0.006192    0.594595     0.602649
4  0.002928    0.004580    0.594595     0.602649
```

[31]:
```
def f(n_neighbors=1):
    results = {}
    knn = KNeighborsClassifier(n_neighbors=n_neighbors)
    scores = cross_validate(knn, X_train, y_train, return_train_score=True)
    results["n_neighbours"] = [n_neighbors]
    results["mean_train_score"] = [round(scores["train_score"].mean(), 3)]
    results["mean_valid_score"] = [round(scores["test_score"].mean(), 3)]
    print(pd.DataFrame(results))
```

```
interactive(
    f,
    n_neighbors=widgets.IntSlider(min=1, max=101, step=10, value=1),
)
```

```
interactive(children=(IntSlider(value=1, description='n_neighbors', max=101,␣
↪min=1, step=10), Output()), _dom_…
```

[32]:
```
plot_knn_decision_boundaries(X_train, y_train, k_values=[1, 11, 100])
```



1.4.3 How to choose `n_neighbors`?

- `n_neighbors` is a hyperparameter
- We can use hyperparameter optimization to choose `n_neighbors`.

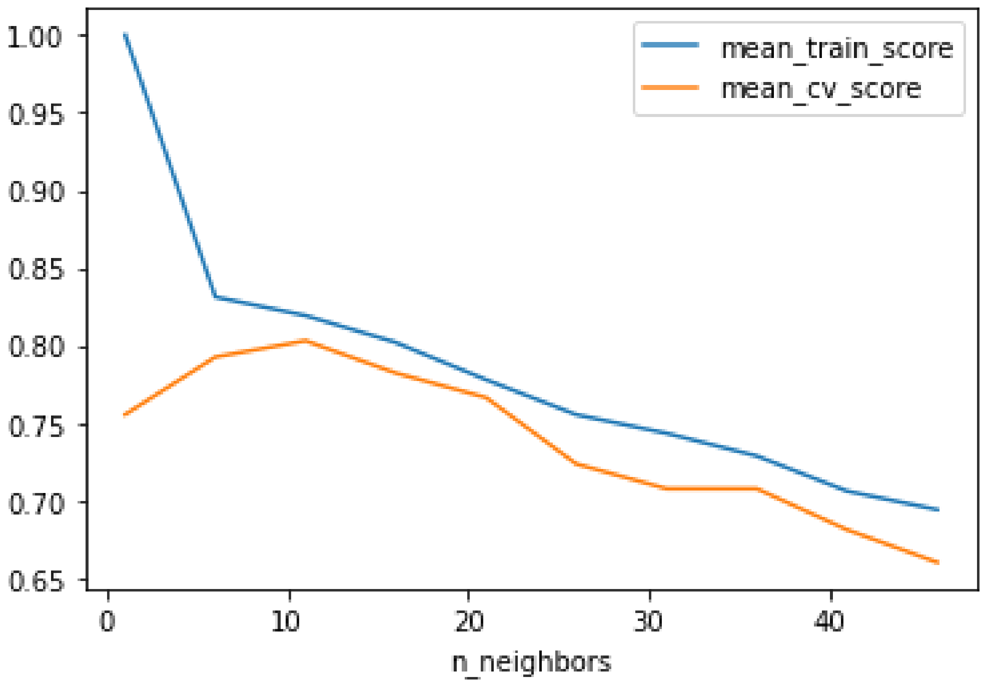
[33]:
```
results_dict = {
    "n_neighbors": [],
    "mean_train_score": [],
    "mean_cv_score": [],
    "std_cv_score": [],
    "std_train_score": [],
}
param_grid = {"n_neighbors": np.arange(1, 50, 5)}

for k in param_grid["n_neighbors"]:
    knn = KNeighborsClassifier(n_neighbors=k)
    scores = cross_validate(knn, X_train, y_train, return_train_score=True)
    results_dict["n_neighbors"].append(k)

    results_dict["mean_cv_score"].append(np.mean(scores["test_score"]))
    results_dict["mean_train_score"].append(np.mean(scores["train_score"]))
    results_dict["std_cv_score"].append(scores["test_score"].std())
    results_dict["std_train_score"].append(scores["train_score"].std())
```

```
results_df = pd.DataFrame(results_dict)
```

[34]:
```
results_df = results_df.set_index("n_neighbors")
results_df
```

[34]:
```
             mean_train_score  mean_cv_score  std_cv_score  std_train_score
n_neighbors
1                    1.000000       0.755477      0.069530         0.000000
6                    0.831135       0.792603      0.046020         0.013433
11                   0.819152       0.802987      0.041129         0.011336
16                   0.801863       0.782219      0.074141         0.008735
21                   0.777934       0.766430      0.062792         0.016944
26                   0.755364       0.723613      0.061937         0.025910
31                   0.743391       0.707681      0.057646         0.030408
36                   0.728777       0.707681      0.064452         0.021305
41                   0.706128       0.681223      0.061241         0.018310
46                   0.694155       0.660171      0.093390         0.018178
```

[35]:
```
results_df[["mean_train_score", "mean_cv_score"]].plot();
```



[36]:
```
best_n_neighbours = results_df.idxmax()["mean_cv_score"]
best_n_neighbours
```

[36]:
```
11
```

Let's try our best model on test data.

[37]:
```python
knn = KNeighborsClassifier(n_neighbors=best_n_neighbours)
knn.fit(X_train, y_train)
print("Test accuracy: %0.3f" % (knn.score(X_test, y_test)))
```

```
Test accuracy: 0.905
```

1.4.4 Questions on distances and k-NNs

Exercise 4.1 True or False: 1. Analogy-based models find examples from the test set that are most similar to the query example we are predicting. 2. A dataset with 10 dimensions is considered low dimensional. 3. Euclidean distance will always have a positive value.

Solutions:

1. True.
2. True. 20 is low. 1000 is medium. 100,000 is high.
3. True.

Exercise 4.2 What would be the Euclidean distance between the following two vectors `u` and `v`?

[38]:
```python
u = np.array([0, 0, 20, -2])
v = np.array([-1, 0, 18, -4])
```

3. sqrt of sum of squared differences

Exercise 4.3 True or False: 1. Unlike with decision trees, with k-NNs most of the work is done at the `predict` stage. 2. With k-NN, setting the hyperparameter k to larger values typically reduces training error. 3. Similar to decision trees, k-NNs finds a small set of good features. 4. In k-NN, the classification of the closest neighbour to the test example always contributes the most to the prediction.

1. True.
2. False.
3. False.
4. False. Sometimes k-NN is not weighted.

Exercise 4.4 k-NN practice questions

1. When we calculated Euclidean distances from all cities to all other cities, why did we set the diagonal entries to infinity before finding the closest city?
2. Consider this toy dataset:

$$X = \begin{bmatrix} 5 & 2 \\ 4 & 3 \\ 2 & 2 \\ 10 & 10 \\ 9 & -1 \\ 9 & 9 \end{bmatrix}, \quad y = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 2 \end{bmatrix}.$$

- If $k = 1$, what would you predict for $x = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$? 1
- If $k = 3$, what would you predict for $x = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$? 0

1.5 More on k-NNs [video]

1.5.1 Other useful arguments of `KNeighborsClassifier`

- `weights` $\rightarrow$ When predicting label, you can assign higher weight to the examples which are closer to the query example.

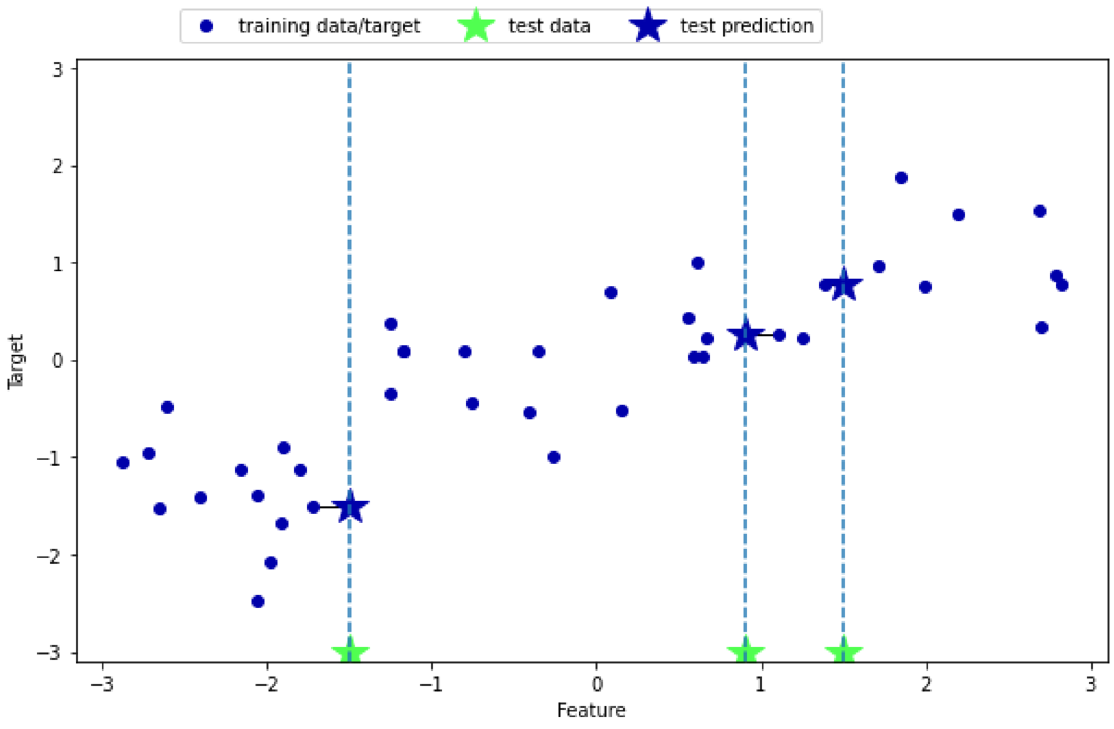
- Exercise for you: Play around with this argument. Do you get a better validation score?

1.5.2 (Optional) Regression with k-nearest neighbours (k-NNs)

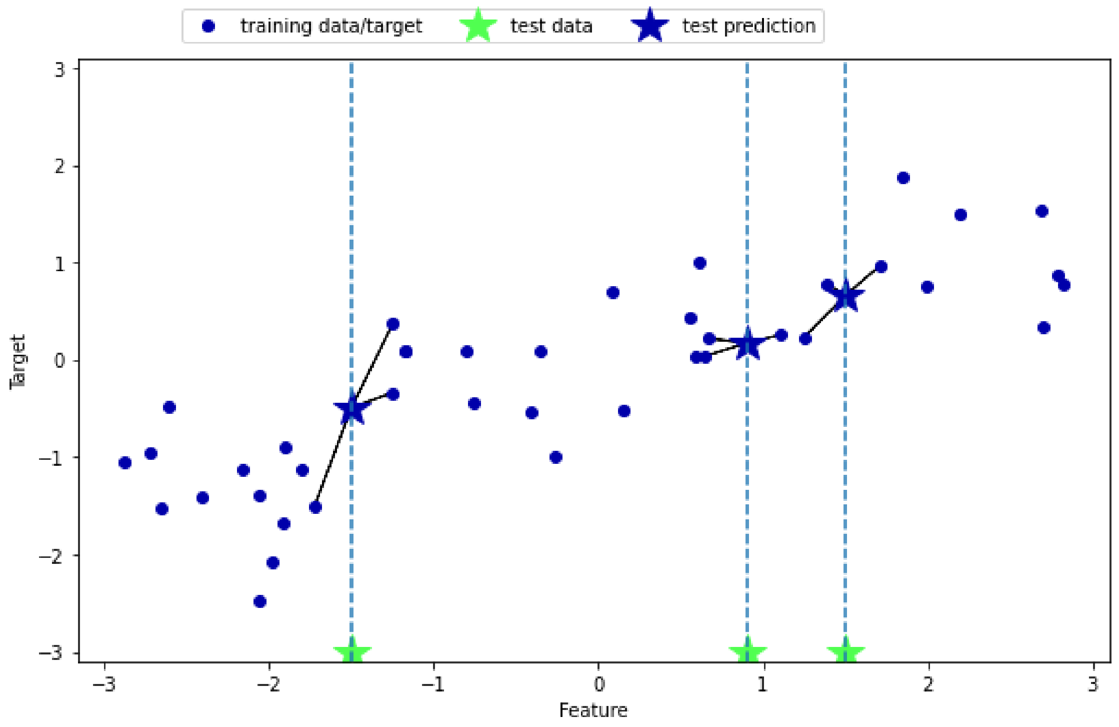
- Can we solve regression problems with k-nearest neighbours algorithm?
- In k-NN regression we take the average of the k-nearest neighbours.
- We can also have weighted regression.

See an example of regression in the lecture notes.

```
[39]: mglearn.plots.plot_knn_regression(n_neighbors=1)
```



```
[40]: mglearn.plots.plot_knn_regression(n_neighbors=3)
```



1.5.3 Pros of k-NNs for supervised learning

- Easy to understand, interpret.
- Simple hyperparameter k (`n_neighbors`) controlling the fundamental tradeoff.
- Can learn very complex functions given enough data.
- **Lazy learning**: Takes no time to `fit`

1.5.4 Cons of k-NNs for supervised learning

- Can be potentially VERY **slow during prediction** time, especially when the training set is very large.
- Often not that great test accuracy compared to the modern approaches.
- It does not work well on datasets with many features or where most feature values are 0 most of the time (sparse datasets).

Important For regular k-NN for supervised learning (not with sparse matrices), you should scale your features. We'll be looking into it soon.

1.5.5 Parametric vs non parametric

- You might see a lot of definitions of these terms.
- A simple way to think about this is:
  - *For n samples, do you need to store at least $O(n)$* worth of stuff to make predictions? If so, it's non-parametric.

- **Non-parametric example**: k-NN is a classic example of non-parametric models.

- **Parametric example**: decision stump
- (Optional) If you want to know more about this, find some reading material here, here, and here.
- (Optional) By the way, the terms "parametric" and "non-paramteric" are often used differently by statisticians, see here for more…

Note $\mathcal{O}(n)$ is referred to as big $\mathcal{O}$ notation. It tells you how fast an algorithm is or how much storage space it requires. For example, in simple terms, if you have n samples and you need to store them all you can say that the algorithm requires $\mathcal{O}(n)$ worth of stuff.

1.5.6 Curse of dimensionality

- Affects all learners but especially bad for nearest-neighbour.
- k-NN usually works well when the number of dimensions d is small but things fall apart quickly as d goes up.
- If there are many irrelevant attributes, k-NN is hopelessly confused because all of them contribute to finding similarity between examples.
- With enough irrelevant attributes the accidental similarity swamps out meaningful similarity and k-NN is no better than random guessing.

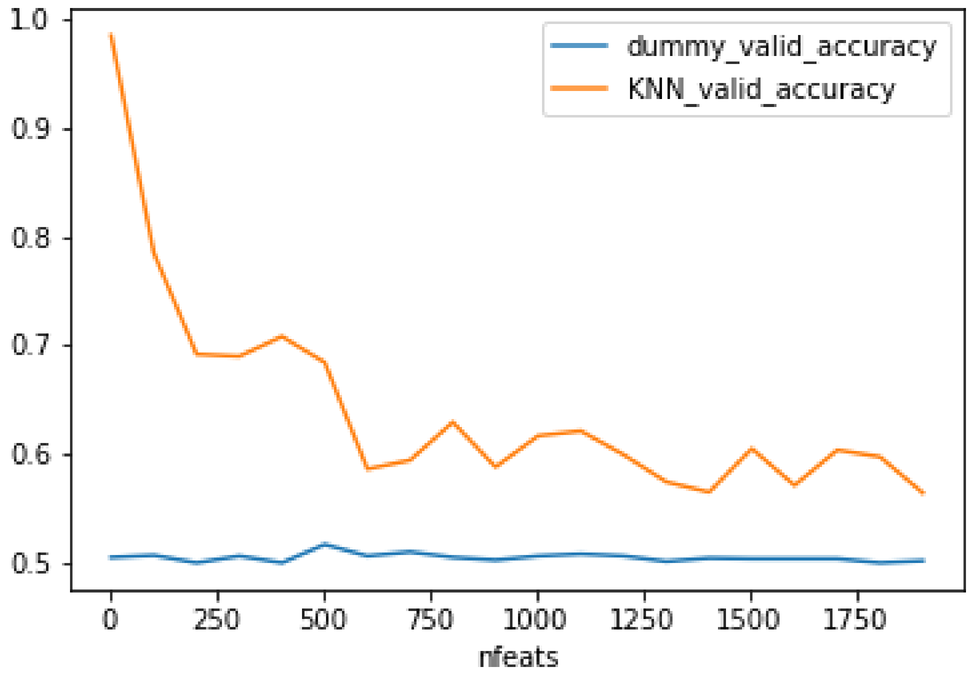
[41]:
```
from sklearn.datasets import make_classification

nfeats_accuracy = {"nfeats": [], "dummy_valid_accuracy": [],␣
 ↪"KNN_valid_accuracy": []}
for n_feats in range(4, 2000, 100):
    X, y = make_classification(n_samples=2000, n_features=n_feats, n_classes=2)
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=0.2, random_state=123
    )
    dummy = DummyClassifier(strategy="most_frequent")
    dummy_scores = cross_validate(dummy, X_train, y_train,␣
 ↪return_train_score=True)

    knn = KNeighborsClassifier()
    scores = cross_validate(knn, X_train, y_train, return_train_score=True)
    nfeats_accuracy["nfeats"].append(n_feats)
    nfeats_accuracy["KNN_valid_accuracy"].append(np.mean(scores["test_score"]))
    nfeats_accuracy["dummy_valid_accuracy"].append(np.
 ↪mean(dummy_scores["test_score"]))
```

[42]:
```
pd.DataFrame(nfeats_accuracy)
```

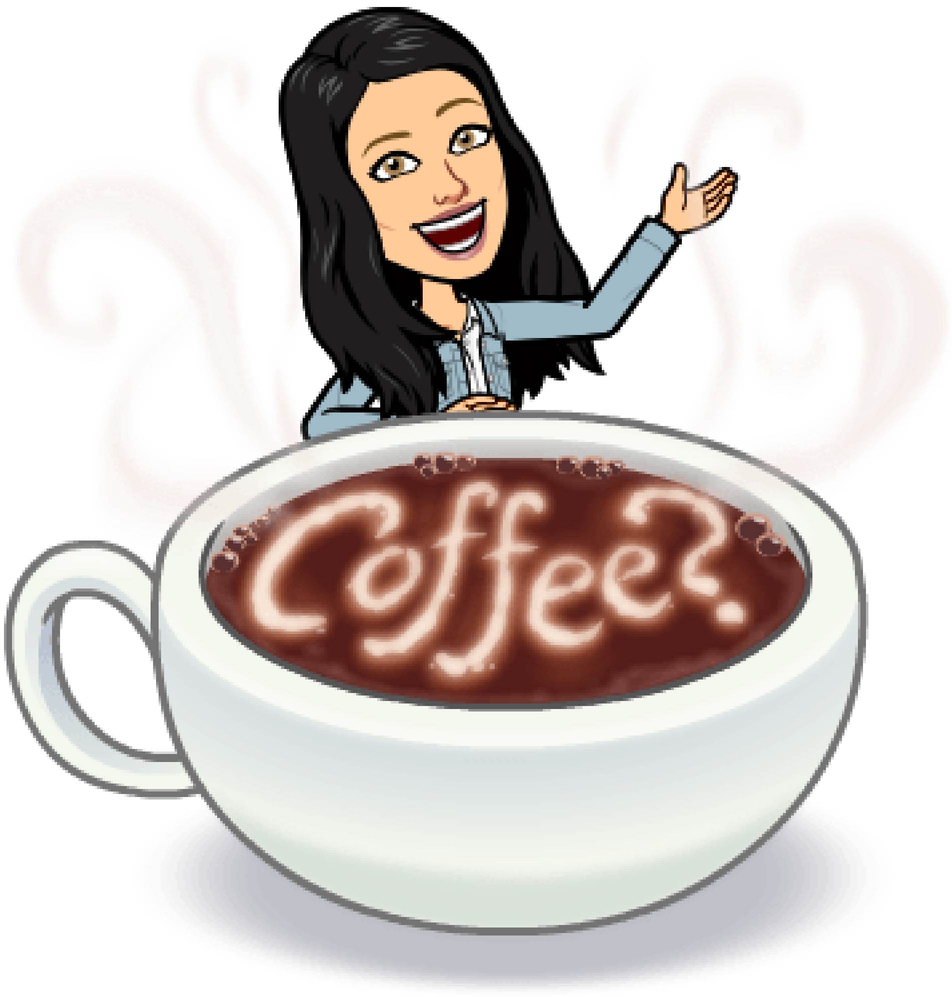
[42]:
```
    nfeats  dummy_valid_accuracy  KNN_valid_accuracy
0        4              0.505000            0.985000
1      104              0.506875            0.785000
2      204              0.500000            0.691875
```

```
3      304            0.506250            0.690000
4      404            0.500000            0.708125
5      504            0.516875            0.684375
6      604            0.506250            0.586250
7      704            0.510000            0.594375
8      804            0.505000            0.629375
9      904            0.502500            0.588125
10    1004            0.506250            0.616875
11    1104            0.508125            0.621250
12    1204            0.506250            0.599375
13    1304            0.501250            0.574375
14    1404            0.504375            0.565000
15    1504            0.503750            0.605000
16    1604            0.503750            0.571250
17    1704            0.503750            0.603750
18    1804            0.500000            0.597500
19    1904            0.501875            0.564375
```

[43]:
```python
pd.DataFrame(nfeats_accuracy).set_index("nfeats").plot();
```



1.6 Break (5 min)



1.7 Support Vector Machines (SVMs) with RBF kernel [video]

- Very high-level overview
- Our goals here are
  - Use `scikit-learn`'s SVM model.
  - Broadly explain the notion of support vectors.

  - Broadly explain the similarities and differences between *k*-NNs and SVM RBFs.
  - Explain how `C` and `gamma` hyperparameters control the fundamental tradeoff.

  (Optional) RBF stands for radial basis functions. We won't go into what it means here. Refer to this video if you want to know more.

1.7.1 Overview

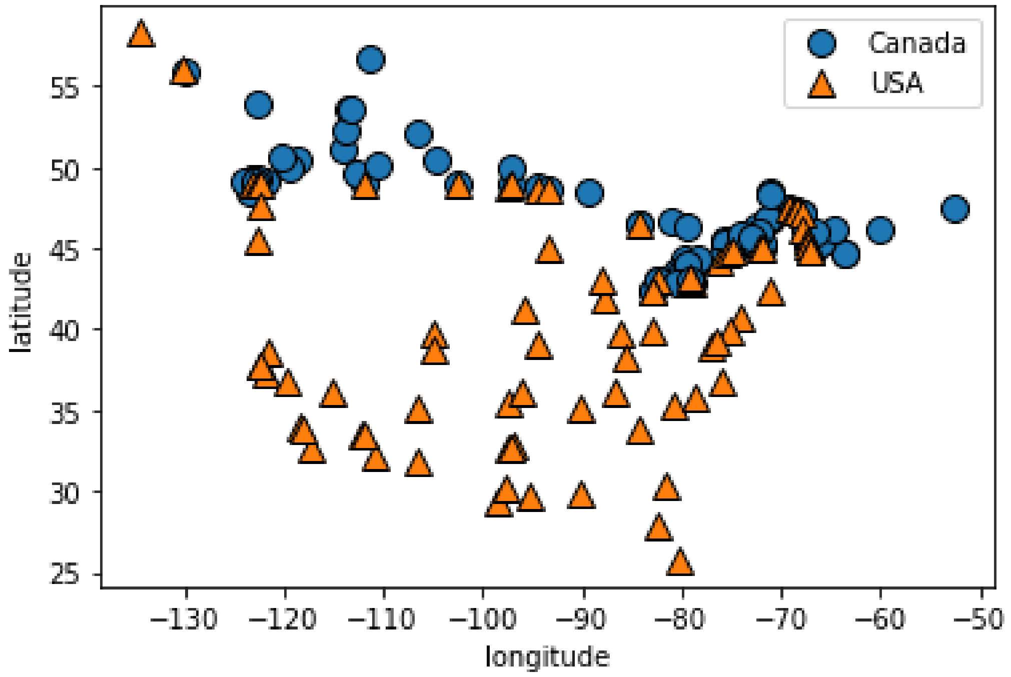
- Another popular similarity-based algorithm is Support Vector Machines with RBF Kernel (SVM RBFs)

- Superficially, SVM RBFs are more like weighted k-NNs.
  - The decision boundary is defined by **a set of positive and negative examples** and **their weights** together with **their similarity measure**.
  - A test example is labeled positive if on average it looks more like positive examples than the negative examples.
- The primary difference between k-NNs and SVM RBFs is that
  - Unlike k-NNs, SVM RBFs **only remember the key examples (*support vectors*)**. So it's more efficient than k-NN.
  - SVMs use a different similarity metric which is called a "kernel" in SVM land. A popular kernel is Radial Basis Functions (RBFs)
  - They usually perform better than k-NNs!

1.7.2 Let's explore SVM RBFs

Let's try SVMs on the cities dataset.

[44]:
```python
mglearn.discrete_scatter(X_cities.iloc[:, 0], X_cities.iloc[:, 1], y_cities)
plt.xlabel("longitude")
plt.ylabel("latitude")
plt.legend(loc=1);
```



[45]:
```python
X_train, X_test, y_train, y_test = train_test_split(
    X_cities, y_cities, test_size=0.2, random_state=123
)
```

```
[46]: knn = KNeighborsClassifier(n_neighbors=best_n_neighbours)
      scores = cross_validate(knn, X_train, y_train, return_train_score=True)
      print("Mean validation score %0.3f" % (np.mean(scores["test_score"])))
      pd.DataFrame(scores)
```

```
Mean validation score 0.803
```

```
[46]:    fit_time  score_time  test_score  train_score
      0  0.002639    0.003402    0.794118     0.819549
      1  0.003361    0.005078    0.764706     0.819549
      2  0.003809    0.004711    0.727273     0.850746
      3  0.003980    0.005426    0.787879     0.828358
      4  0.002620    0.004390    0.939394     0.783582
```

```
[47]: from sklearn.svm import SVC

      svm = SVC(gamma=0.01)  # Ignore gamma for now
      scores = cross_validate(svm, X_train, y_train, return_train_score=True)
      print("Mean validation score %0.3f" % (np.mean(scores["test_score"])))
      pd.DataFrame(scores)
```

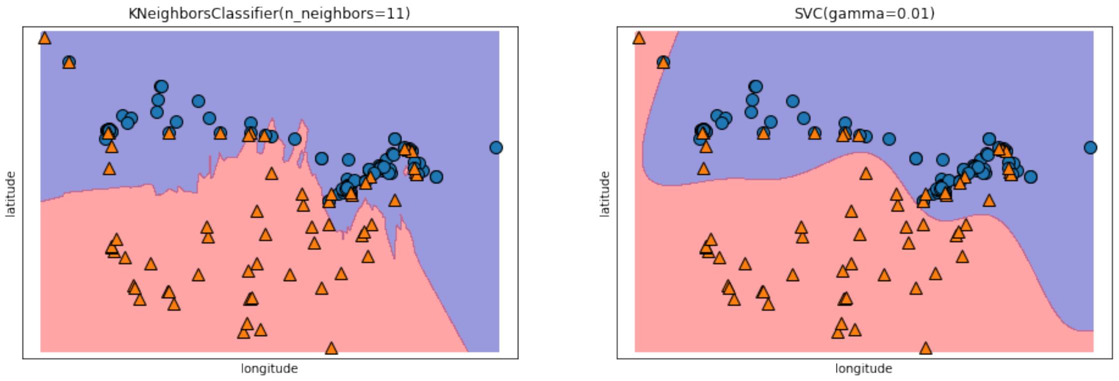
```
Mean validation score 0.820
```

```
[47]:    fit_time  score_time  test_score  train_score
      0  0.005395    0.002488    0.823529     0.842105
      1  0.002779    0.001778    0.823529     0.842105
      2  0.004292    0.002858    0.727273     0.858209
      3  0.004943    0.003094    0.787879     0.843284
      4  0.004807    0.002745    0.939394     0.805970
```

1.7.3 Decision boundary of SVMs

- We can think of SVM with RBF kernel as “smooth KNN”.

```
[48]: fig, axes = plt.subplots(1, 2, figsize=(16, 5))

      for clf, ax in zip([knn, svm], axes):
          clf.fit(X_train, y_train)
          mglearn.plots.plot_2d_separator(
              clf, X_train.to_numpy(), fill=True, eps=0.5, ax=ax, alpha=0.4
          )
          mglearn.discrete_scatter(X_train.iloc[:, 0], X_train.iloc[:, 1], y_train,␣
       ↪ax=ax)
          ax.set_title(clf)
          ax.set_xlabel("longitude")
          ax.set_ylabel("latitude")
```

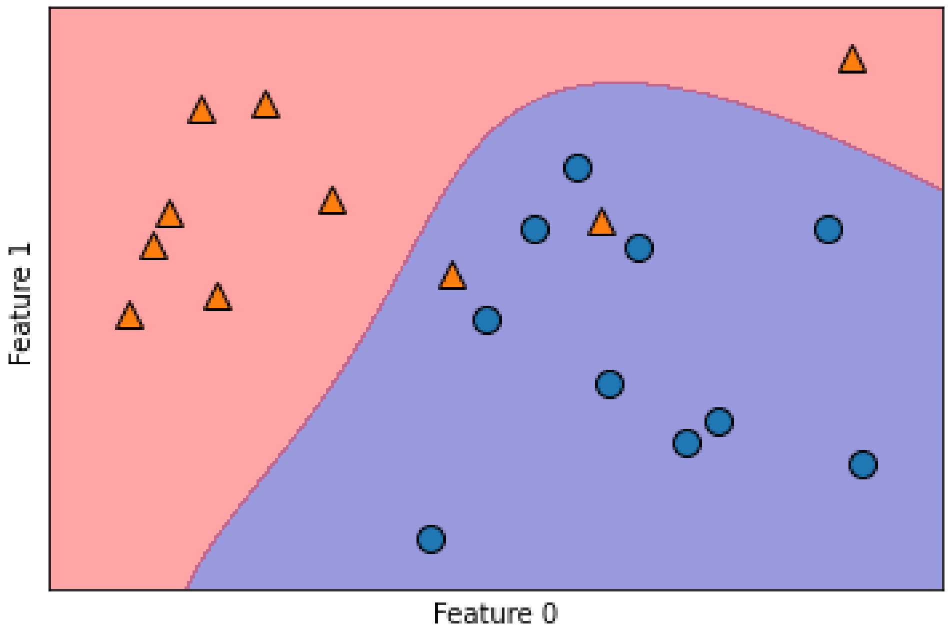


1.7.4 Support vectors

- Each training example either is or isn't a "support vector".
  - This gets decided during `fit`.
- **Main insight: the decision boundary only depends on the support vectors.**
- Let's look at the support vectors.

[49]:
```python
from sklearn.datasets import make_blobs

n = 20
n_classes = 2
X_toy, y_toy = make_blobs(
    n_samples=n, centers=n_classes, random_state=300
)  # Let's generate some fake data
```

[50]:
```python
mglearn.discrete_scatter(X_toy[:, 0], X_toy[:, 1], y_toy)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
svm = SVC(kernel="rbf", C=10, gamma=0.1).fit(X_toy, y_toy)
mglearn.plots.plot_2d_separator(svm, X_toy, fill=True, eps=0.5, alpha=0.4)
```



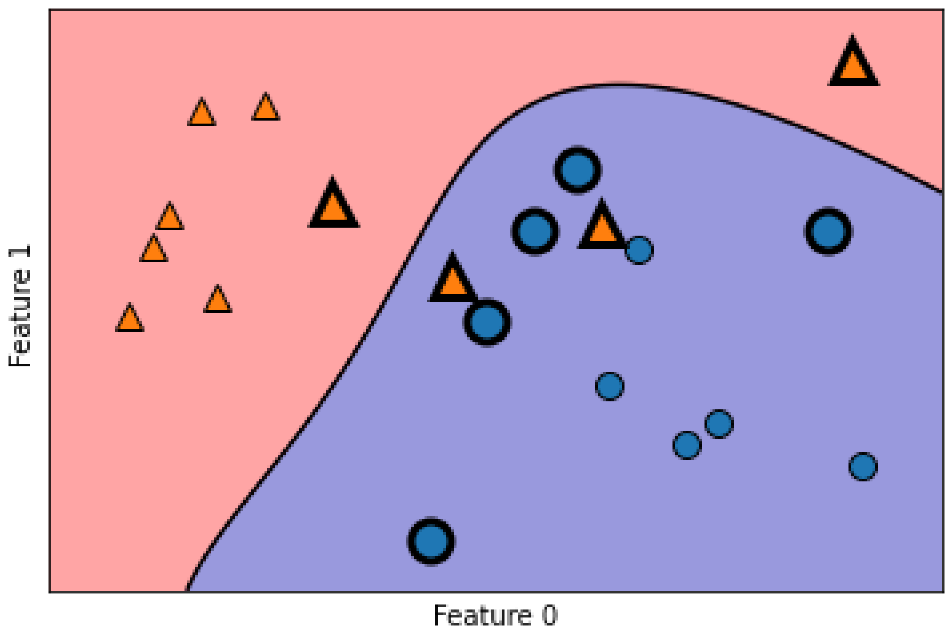
```
[51]: svm.support_
```

```
[51]: array([ 3,  8,  9, 14, 19,  1,  4,  6, 17], dtype=int32)
```

```
[52]: X_toy[svm.support_]
```

```
[52]: array([[-2.03730912, -7.49852518],
              [-1.10776665, -3.6937689 ],
              [ 0.483923  , -4.33773178],
              [-1.6829074 , -5.24397776],
              [-1.37419828, -4.3357658 ],
              [-1.89963176, -4.78162382],
              [-2.66622445, -4.01676697],
              [ 0.62593608, -2.57502093],
              [-0.96004841, -4.23630586]])
```

```
[53]: mglearn.plots.plot_2d_separator(svm, X_toy, fill=True, eps=0.5, alpha=0.4)
      plot_support_vectors(svm, X_toy, y_toy)
```



The support vectors are the bigger points in the plot above.

1.7.5 Hyperparameters of SVM

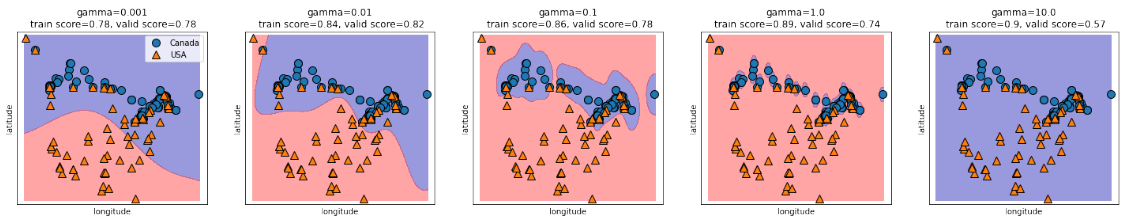
- Key hyperparameters of `rbf` SVM are
  - `gamma`
  - `C`
- We are not equipped to understand the meaning of these parameters at this point but you are expected to describe **their relation to the fundamental tradeoff**.

Optionally, see `scikit-learn`'s explanation of RBF SVM parameters.

1.7.6 Relation of `gamma` and the fundamental trade-off

- `gamma` controls the complexity (fundamental trade-off), just like other hyperparameters we've seen.
  - larger `gamma` $\rightarrow$ more complex
  - smaller `gamma` $\rightarrow$ less complex

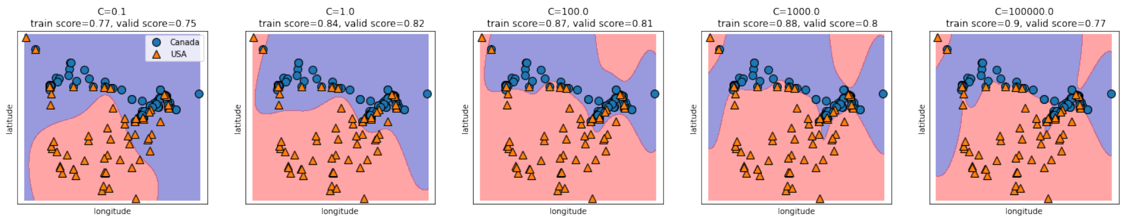
[54]:
```
gamma = [0.001, 0.01, 0.1, 1.0, 10.0]
plot_svc_gamma(
    gamma,
    X_train.to_numpy(),
    y_train.to_numpy(),
    x_label="longitude",
    y_label="latitude",
)
```



1.7.7 Relation of `C` and the fundamental trade-off

- `C` *also* affects the fundamental tradeoff
  - larger `C` → more complex
  - smaller `C` → less complex

```
[55]: C = [0.1, 1.0, 100.0, 1000.0, 100000.0]
      plot_svc_C(
          C, X_train.to_numpy(), y_train.to_numpy(), x_label="longitude",␣
        ↪y_label="latitude"
      )
```



1.7.8 Search over multiple hyperparameters

- So far you have seen how to carry out search over a hyperparameter
- In the above case the best training error is achieved by the most complex model (large `gamma`, large `C`).
- Best validation error requires a hyperparameter search to balance the fundamental tradeoff.
  - In general we can't search them one at a time.
  - More on this next week. But if you cannot wait till then, you may look up the following:
    * sklearn.model_selection.GridSearchCV
    * sklearn.model_selection.RandomizedSearchCV

1.7.9 SVM Regressor

- Similar to KNNs, you can use SVMs for regression problems as well.
- See `sklearn.svm.SVR` for more details.

1.7.10 Questions on SVM RBFs

Exercise 4.5 Exercise 4.5: SVM RBF True/False questions 1. k-NN may perform poorly in high-dimensional space (say, $d > 1000$). True 2. Similar to KNN, SVM with RBF kernel is a non-parametric model. 3. In SVM RBF, removing a non-support vector would not change the decision boundary. 4. In sklearn's SVC classifier, large values of gamma tend to result in higher training score but probably lower validation score. 5. If we increase both gamma and C, we can't be certain if the model becomes more complex or less complex.

More practice questions

- Check out some more practice questions here.

1.8 Summary

- We have KNNs and SVMs as new supervised learning techniques in our toolbox.
- These are analogy-based learners and the idea is to assign nearby points the same label.
- Unlike decision trees, all features are equally important.
- Both can be used for classification or regression (much like the other methods we've seen).

1.8.1 Coming up:

Lingering questions: - Are we ready to do machine learning on real-world datasets? - What would happen if we use k-NNs or SVM RBFs on the spotify dataset from hw1?
- What happens if we have missing values in our data? - What do we do if we have features with categories or string values?

SEE YOU SOON