Lecture 04 – K-Nearest Neighbors

Kien C Nguyen

August 1, 2020

Contents

Instance-based Learning

- Instance-based learning is often termed lazy learning, as there is typically no "transformation" of training instances into more general "statements"
- Instead, the presented training data is simply stored and, when a new query instance is encountered, a set of similar, related instances is retrieved from memory and used to classify the new query instance
- Hence, instance-based learners never form an explicit general hypothesis regarding the target function. They simply compute the classification of each new query instance as needed

k-Nearest Neighbors

- The simplest, most used instance-based learning algorithm is the k-NN algorithm
- k-NN assumes that all instances are points in some n-dimensional space and defines neighbors in terms of distance (usually Euclidean in R-space)
- k is the number of neighbors considered

1-NN Illustration

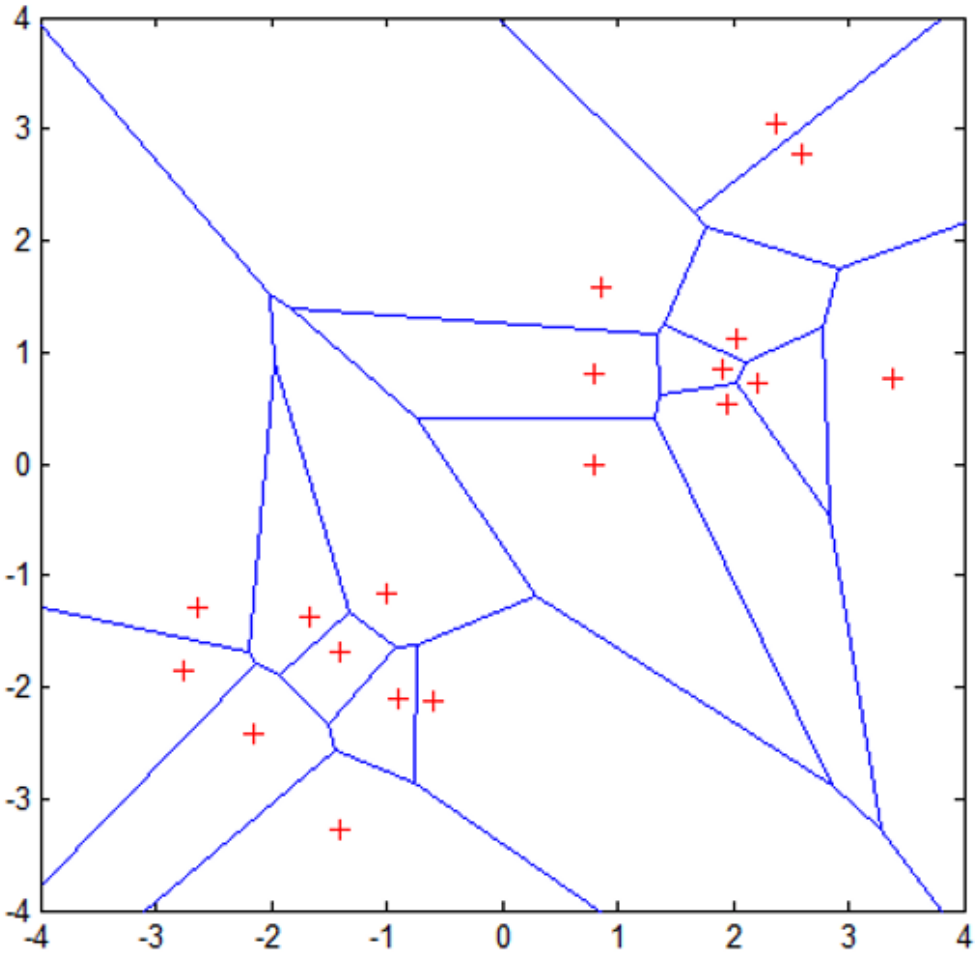

Figure: Voronoi diagram for 1-nearest neighbor algorithm. The sample in each Voronoi cell is the nearest neighbor for any point in the same cell [1].

Contents

k-NN classification

- The k-NN classification rule is to assign to a test sample the majority category label of its k nearest training samples
- In practice, k is usually chosen to be odd, so as to avoid ties
- The $k = 1$ rule is generally called the nearest-neighbor classification rule

- V is the finite set of target values
- For each training instance $t = (x, f(x))$, add t to the set T.
- Given a query instance q to be classified Let $x_1, \ldots, x_k$ be the k training instances in T nearest to q. Return

$$\hat{f}(q) = \arg\max_{v \in V} \sum_{i=1}^{k} \delta\left(v, f\left(x_i\right)\right) \tag{1}$$

  where $\delta\left(a, b\right)$ is the Kronecker delta. $\delta\left(a, b\right) = 1$ if $a = b$, and 0 otherwise.
- Intuitively, the k-NN algorithm assigns to each new query instance the majority class among its k nearest neighbors

k-NN Ilustration

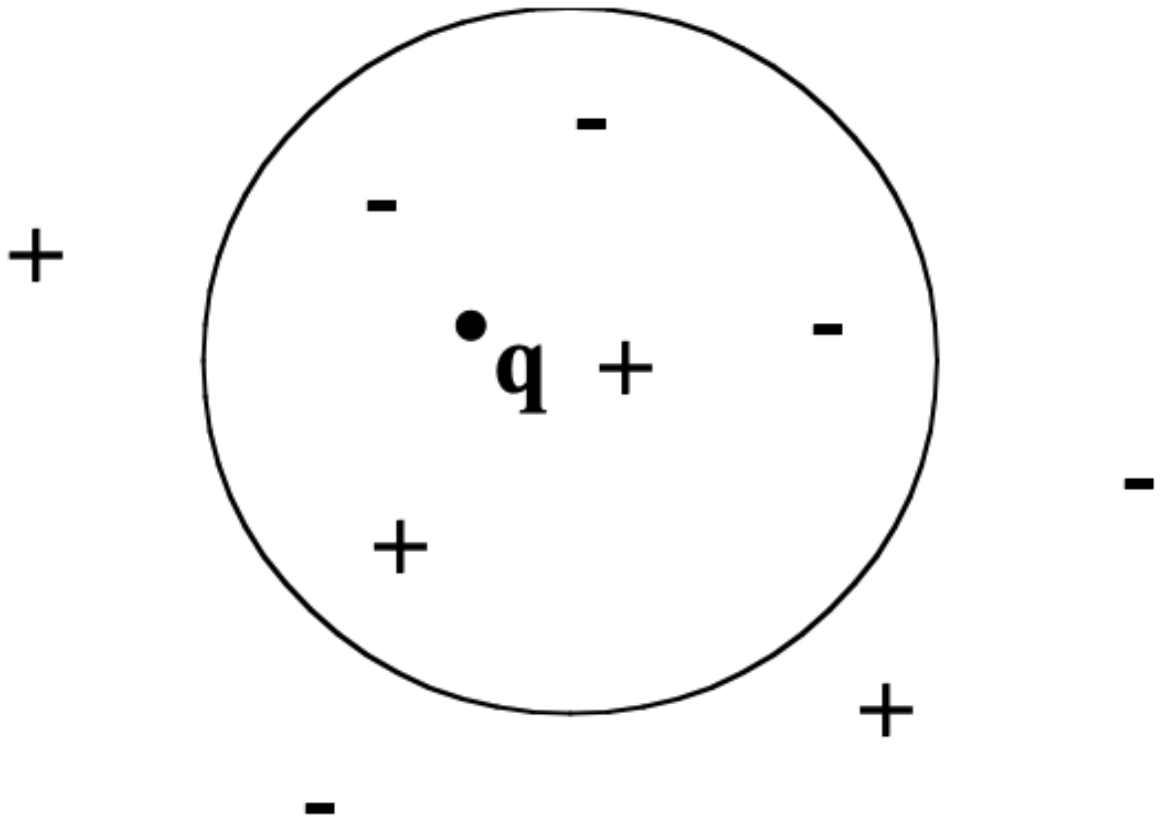

Figure: q is + under 1-NN, but – under 5-NN [1].

Distance-weighted k-NN

- V is the finite set of target values
- For each training instance $t = (x, f(x))$, add t to the set T.
- Given a query instance q to be classified Let $x_1, \ldots, x_k$ be the k training instances in T nearest to q. Return

$$\hat{f}(q) = \arg\max_{v \in V} \sum_{i=1}^{k} \frac{1}{d\left(x_i, x_q\right)^2} \delta\left(v, f\left(x_i\right)\right) \tag{2}$$

Scale Effects

- Different features may have different measurement scales
  - E.g., In predicting house prices, areas may range from tens to thousands of square meters, numbers of floors may range from 1 to 10.
- Consequences
  - Patient weight will have a much greater influence on the distance between samples
  - May bias the performance of the classifier

Standardizing Numerical Variables – Standard scaling

Z-score normalization are features that are re-scaled to have a mean of zero and a standard deviation of one.

$$z = \frac{x - \mu}{\sigma}$$

Where:

- z: z-score.
- x: previous feature value.
- μ: mean of feature value.
- σ: standard deviation of feature value.

Standard scaling example

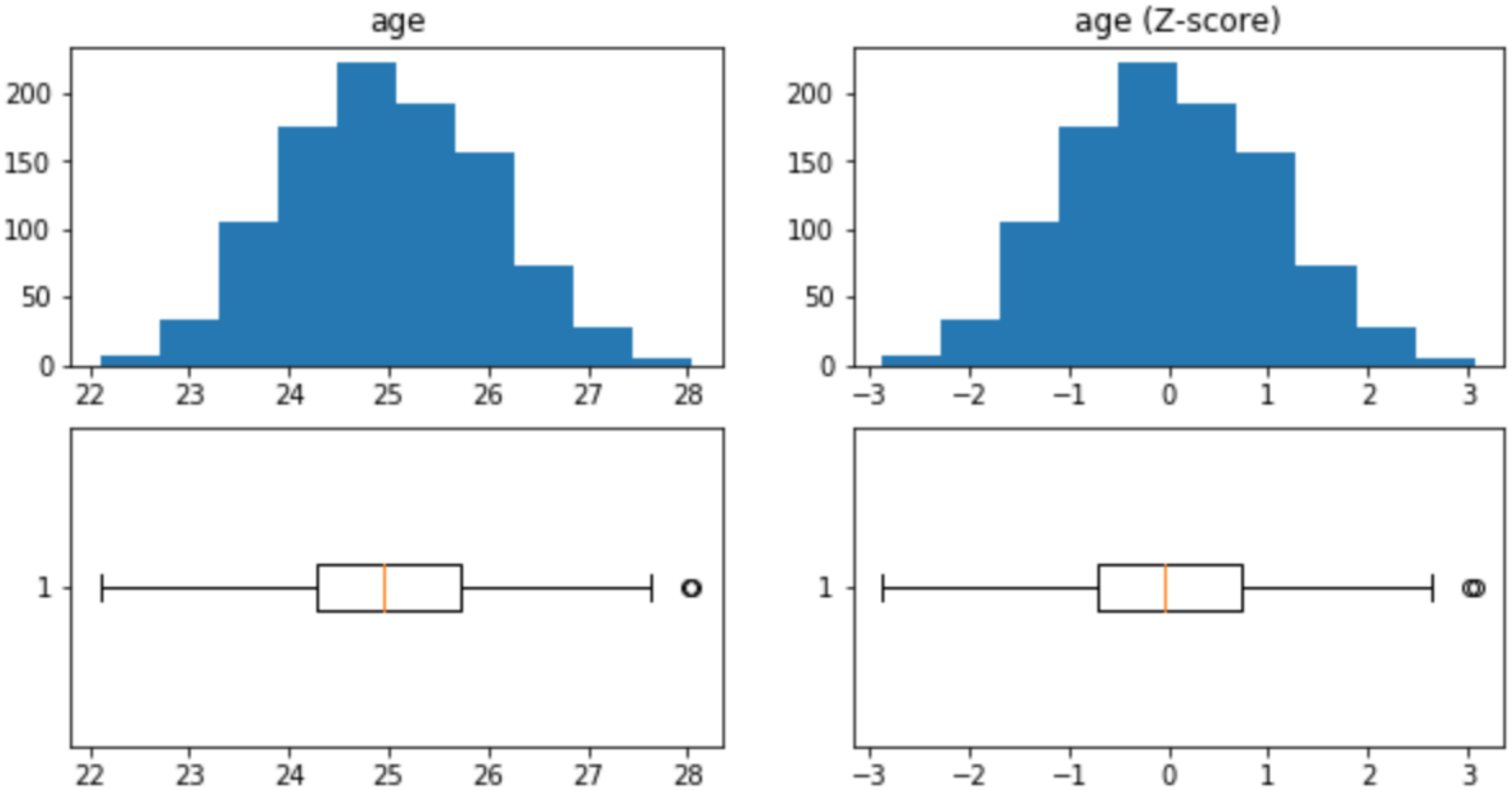

Figure: Z-score transformation on age feature

Standardizing Numerical Variables – min-max scaling

The idea is to get every input feature into approximately a $[0, 1]$ range. The name comes from the use of min and max functions, namely the smallest and greatest values in your dataset. It requires dividing the input values by the range (i.e. the maximum value minus the minimum value) of the input variable:

$$x_i' = \frac{x_i - min(x_i)}{max(x_i) - min(x_i)}$$

Where:

- x_i: is the original i-th input value.
- x_i': normalize feature.

Min-max scaling

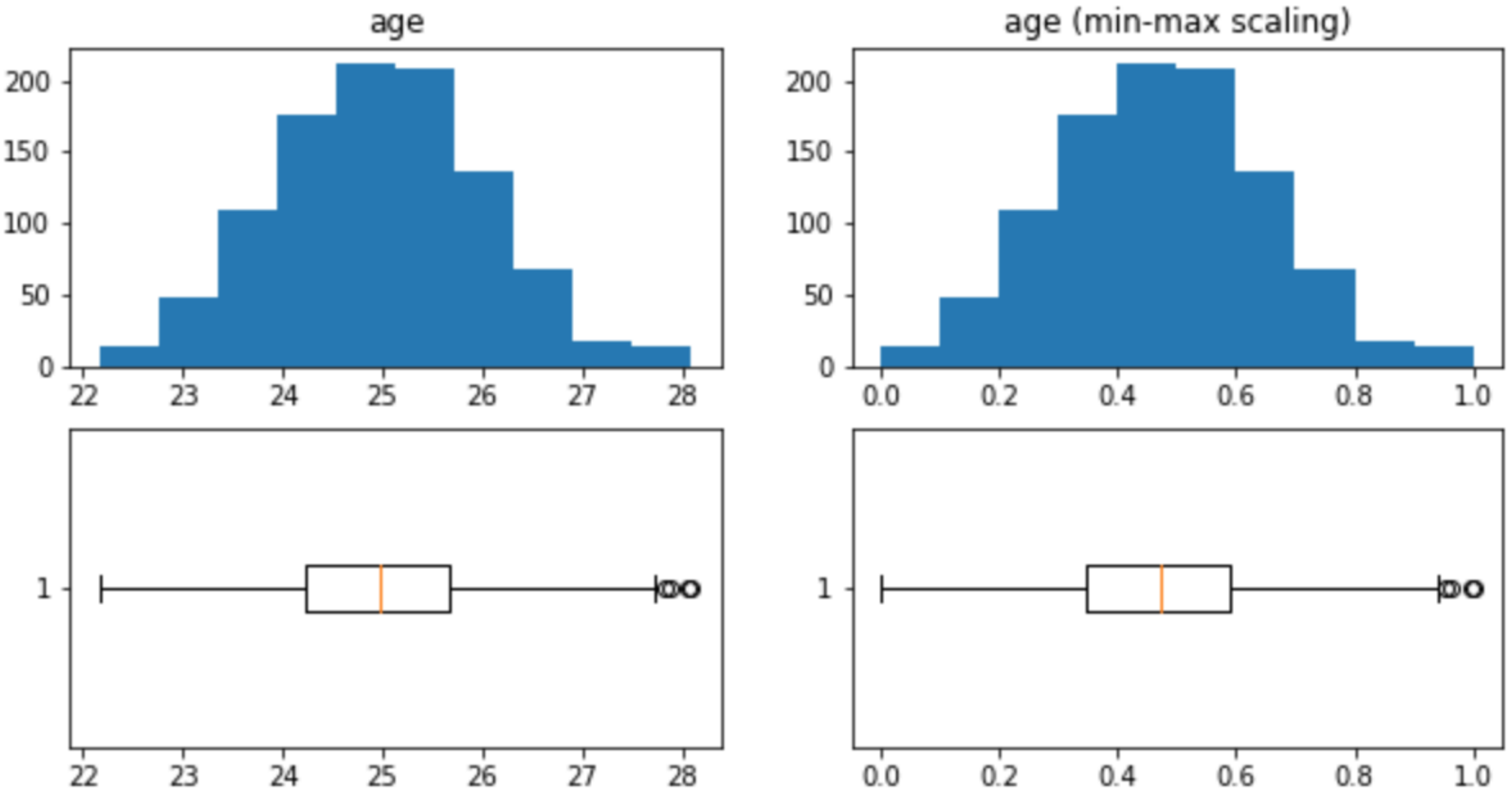

Figure: Min-Max scaling on age feature

Some Remarks

- k-NN works well on many practical problems and is fairly noise tolerant (depending on the value of k)
- k-NN is subject to the curse of dimensionality (i.e., presence of many irrelevant attributes)
- k-NN needs adequate distance measure
- k-NN relies on efficient indexing

Contents

k-NN regression

- For each training instance $t = (x, f(x))$, add t to the set T.
- Given a query instance q to be predicted Let $x_1, \ldots, x_k$ be the k training instances in T nearest to q. Return

$$\hat{f}(q) = \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i} \tag{3}$$

  where w_i is the weight for instance i.
- Note: unweighted corresponds to $w_i = 1$ for all i.

How do we determine K? [2]

- Try different values, and see which works best on the test set?
  - Could do that, but then we are selecting the best K for our particular test set. This means that the performance on our test set is now an overestimate of how well we'd do on new data
- Solution: set aside a validation set (which is separate from both the training and the test set), and select the K for the best performance on the validation set, but report the results on the test set
  - Generally, the performance on the validation set will be better than on the test set
  - What about the performance on the training set?

What does the best K say about the data?

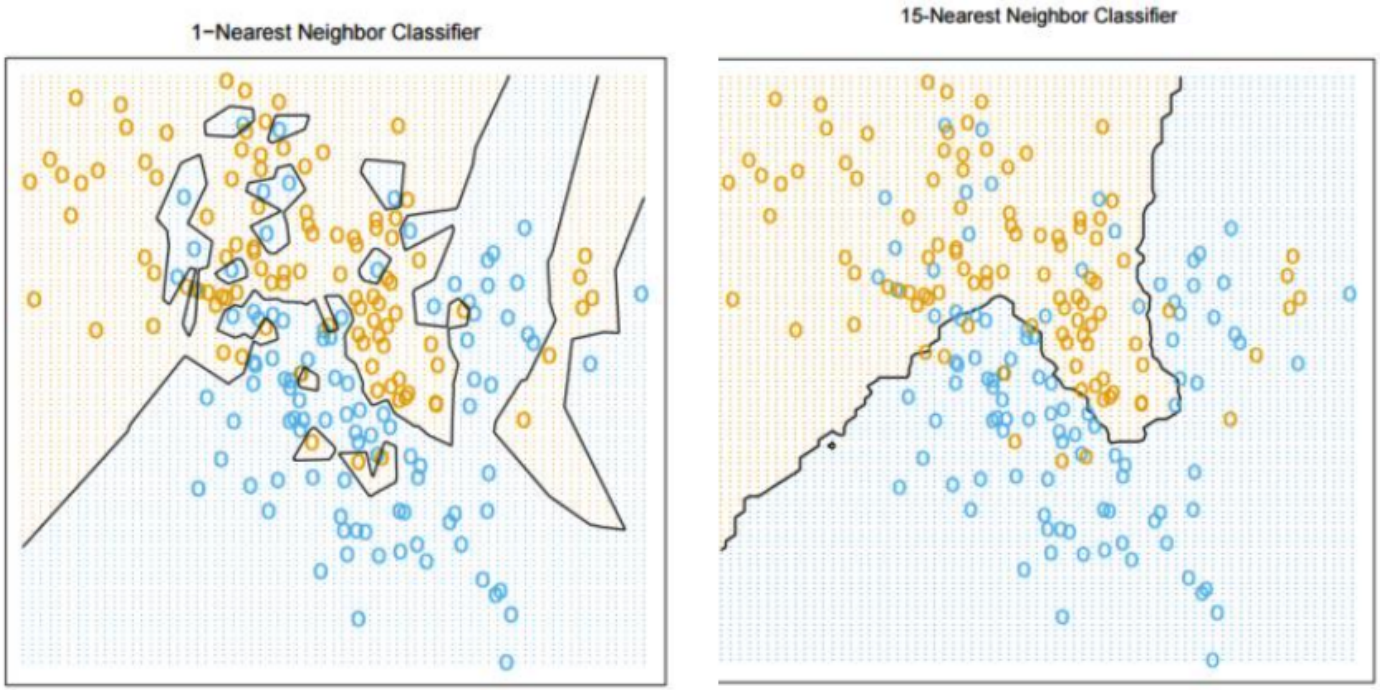

Figure: k = 1 (left) and k=15 (right)

- Large k: relatively simple boundary, no small "islands" in the data. Small changes in x do no generally change the label
- Small k: a complex boundary between the labels. Small changes in x often change the labels

Why not let K be very large?

- Performance on both the training set and the test set is poor
- This is an example of underfitting

Contents

Müller & Guido's book [3]

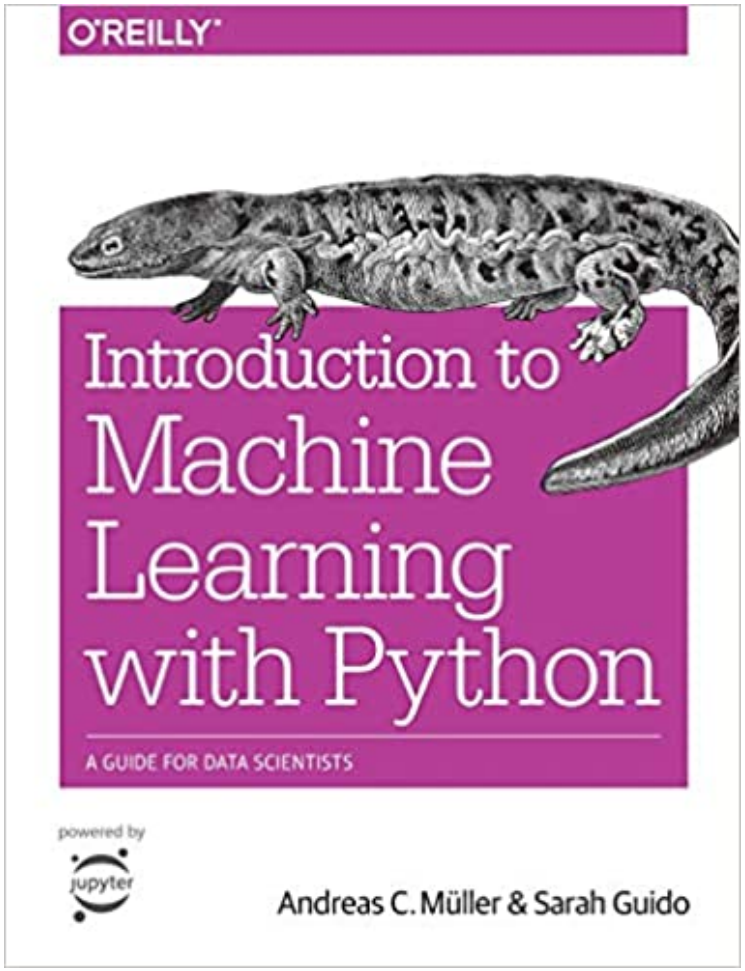

Figure: Introduction to Machine Learning with Python

Cloning code from Müller & Guido's github

Clone code from `github.com/amueller/introduction_to_ml_with_python.git` using https

```
git clone https://github.com/amueller/introduction_to_ml_with_python.git

Cloning into 'introduction_to_ml_with_python'...
remote: Enumerating objects: 436, done.
remote: Total 436 (delta 0), reused 0 (delta 0), pack-reused 436
Receiving objects: 100% (436/436), 178.38 MiB | 2.21 MiB/s, done.
Resolving deltas: 100% (189/189), done.
```

Müller & Guido's git repo

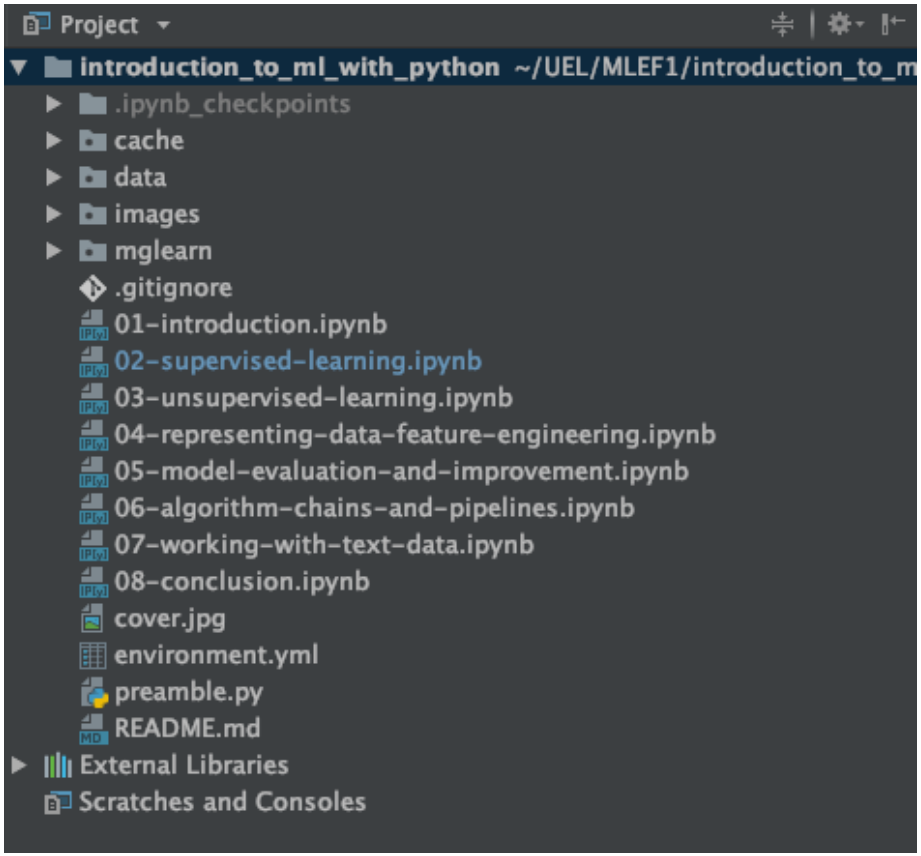

Figure: Introduction to Machine Learning with Python – git repo

```
# generate dataset
X, y = mglearn.datasets.make_forge()
# plot dataset
mglearn.discrete_scatter(X[:, 0], X[:, 1], y)
plt.legend(["Class 0", "Class 1"], loc=4)
plt.xlabel("First feature")
plt.ylabel("Second feature")
print("X.shape:", X.shape)
```

```
X.shape: (26, 2)
```

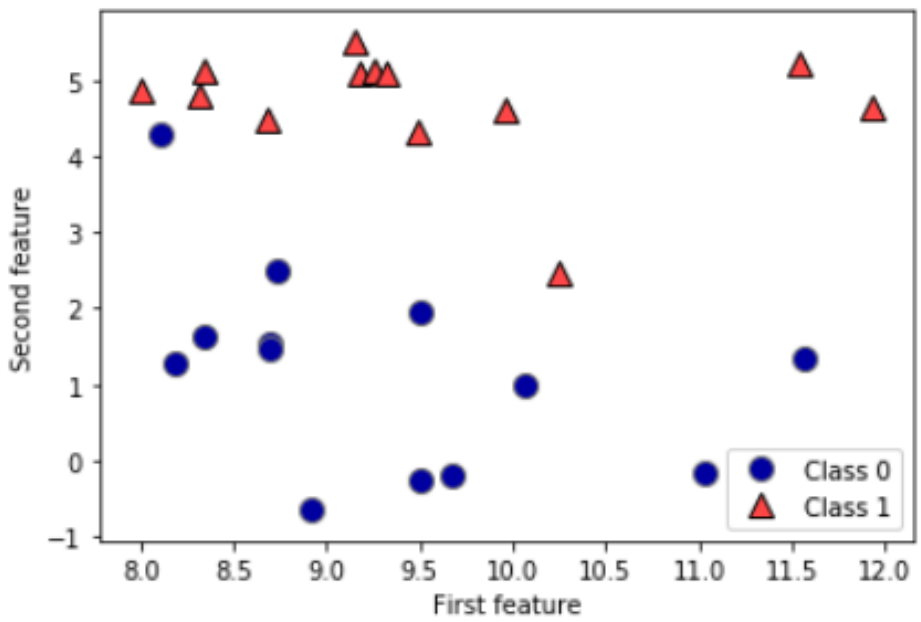

Figure: Visualization of the dataset

```
mglearn.plots.plot_knn_classification(n_neighbors=1)
```

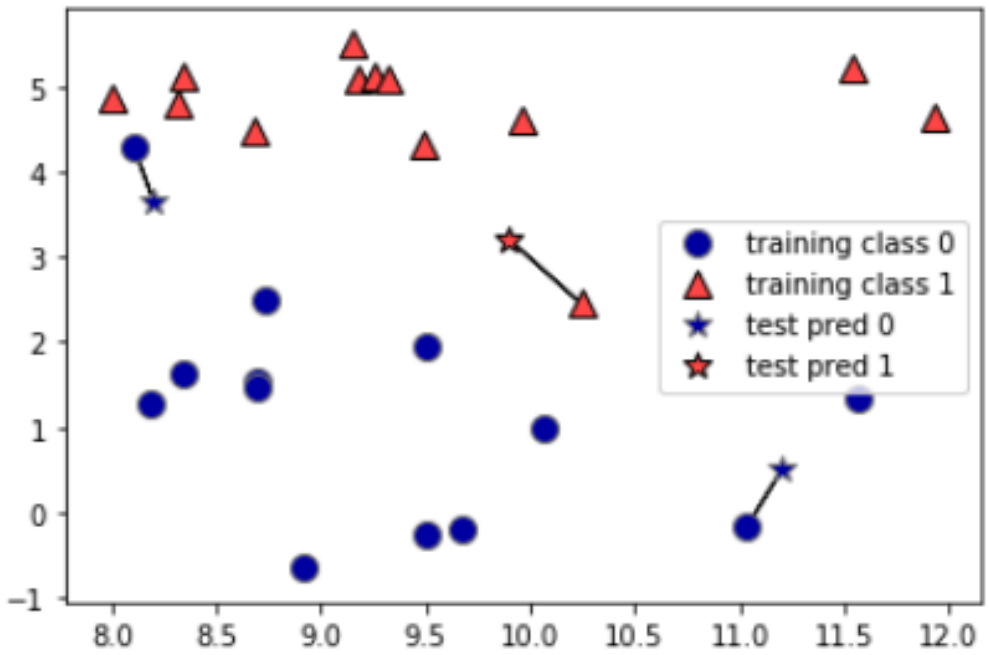

Figure: Predictions made by the one-nearest-neighbor model on the forge dataset

3-NN

```
mglearn.plots.plot_knn_classification(n_neighbors=3)
```

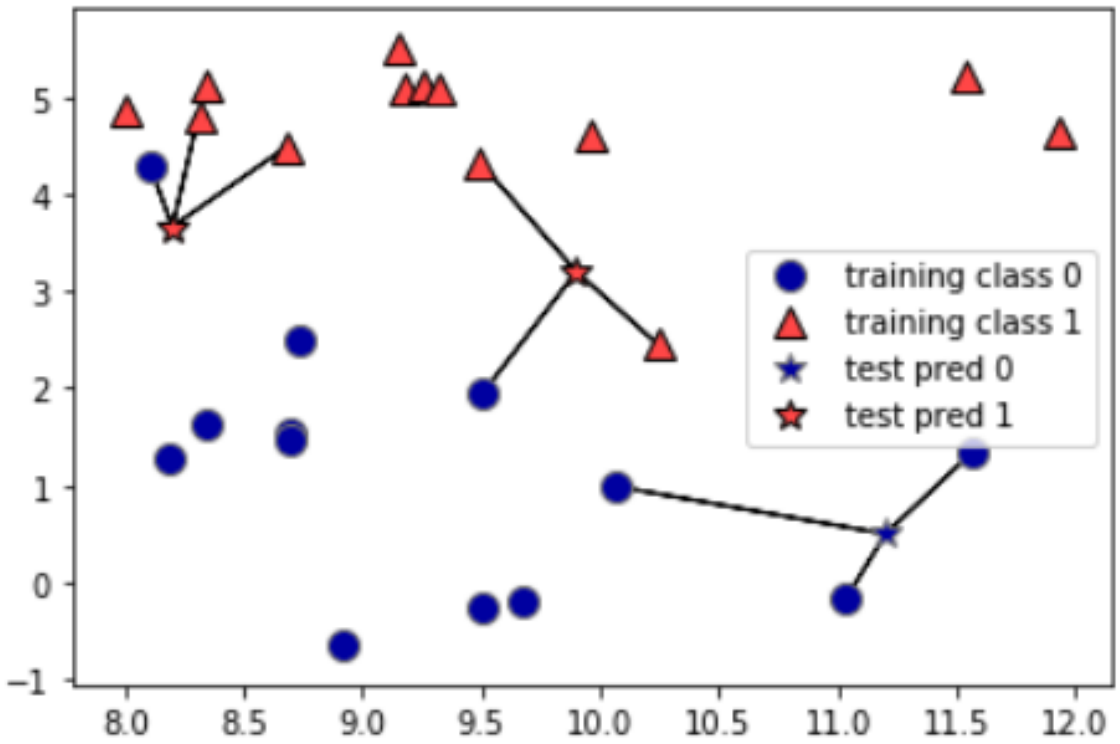

Figure: Predictions made by the three-nearest-neighbor model on the forge dataset

Why not let K be very small?

- Great for the performance on the training set!
  - Perfect performance guaranteed for $k = 1$
- If the test data does not look exactly like the training data, the performance on the test data will be worse for k that is too small
  - The training data could be noisy (e.g., in the orange region, data points are sometimes blue with probability 5%, randomly)
  - This is an example of overfitting – building a classifier that works well on the training set, but does not generalize well to the test set

References

[1] BYU, CS 478 Tools for Machine Learning and Data Mining, http://dml.cs.byu.edu/~cgc/docs/mldm_tools
[2] University of Toronto, CSC411: Machine Learning and Data Mining (Winter 2017), https://www.cs.toronto.edu/~guerzhoy/411/
[3] A C Müller, S. Guido, Introduction to Machine Learning with Python, O'Reilly Media, Inc., 2017