Machine Learning

K-Nearest Neighbors

Kien C Nguyen

August 23, 2024

Instance-Based Learning - Key Concepts

Definition:

- Instance-based learning (IBL) is a type of machine learning where the model makes predictions based on the instances (examples) in the training dataset.
- Unlike other learning algorithms that create an explicit generalization of the data, instance-based learning methods compare new problem instances with instances seen in training.

Key Characteristics:

- **No Explicit Model:** The algorithm does not derive a model; it uses the training instances directly to make predictions.
- **Lazy Learning:** Instance-based learning is often referred to as "lazy learning" because it delays the generalization process until a query is made.
- **Similarity Measure:** Predictions are made based on a similarity measure (e.g., Euclidean distance) between the stored instances and the new instance.

Instance-Based Learning - Examples

Examples of Instance-Based Algorithms:

- **k-Nearest Neighbors (k-NN):** One of the most common instance-based learning algorithms. It classifies new instances based on the majority class among the k closest instances.
- **Locally Weighted Regression (LWR):** Performs regression at the query point using nearby training data points.
- **Case-Based Reasoning (CBR):** Solves new problems by adapting solutions that were used to solve old, similar problems.

Advantages:

- Simple to implement and understand.
- Naturally handles multi-class problems.
- Can adapt to changes in the problem domain without retraining.

Disadvantages:

- Computationally expensive, especially with large datasets, as the model needs to compute the similarity for each query.
- Storage-intensive since all instances need to be stored.
- Sensitive to irrelevant or noisy features.

k-Nearest Neighbors

- The k-NN algorithm is the simplest and most used instance-based learning method.
- k-NN operates under the assumption that all data instances are represented as points within an n-dimensional space, with neighbors determined by their distance, typically using Euclidean distance in R-space.
- The parameter k represents the number of neighboring points taken into account.

1-NN Illustration

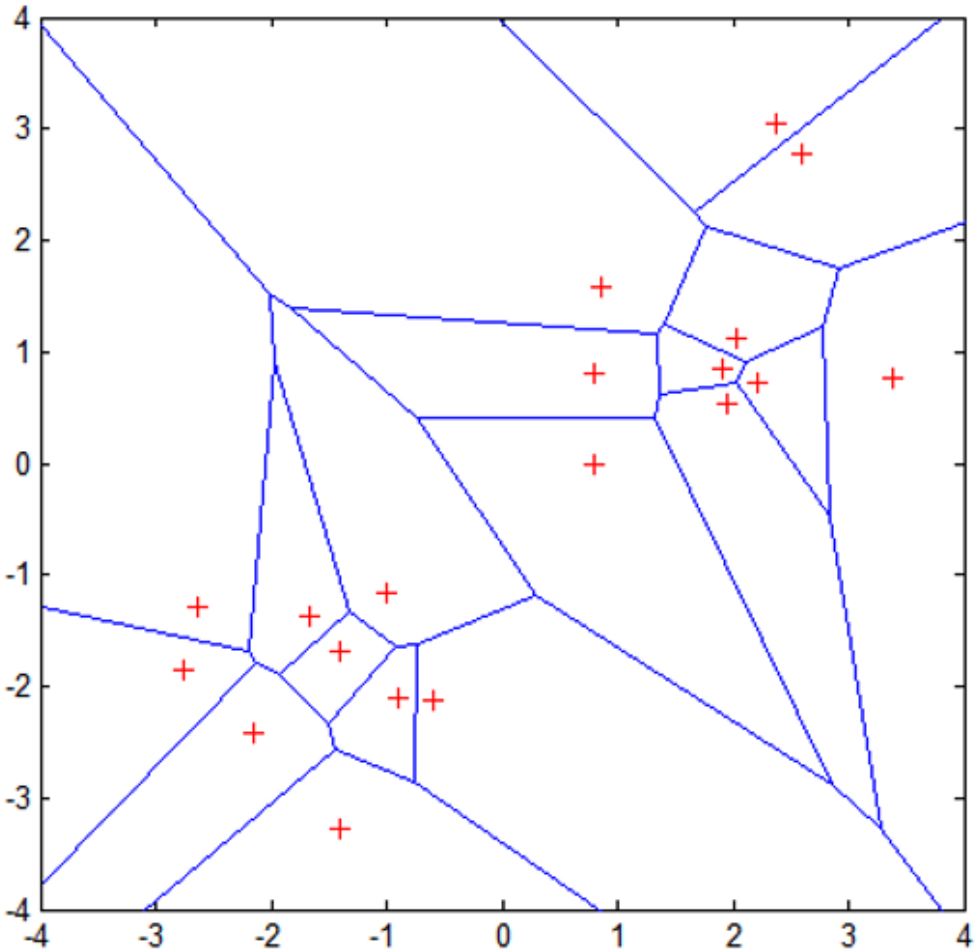

Figure: Voronoi diagram for 1-nearest neighbor algorithm. The sample in each Voronoi cell is the nearest neighbor for any point in the same cell [1].

k-NN classification

- In the k-NN classification model, we select the k training examples in the training dataset that are closest to the data point we need to classify.
- The class that is most common among these k training examples is selected as the prediction.
- In practice, for a binary classification problem, k is usually chosen to be odd, so as to avoid ties. For example, if k is even, there may be cases where $k/2$ training examples are labeled as Class 0, and the other $k/2$ training examples labeled as Class 1.
- The $k = 1$ rule is generally called the nearest-neighbor classification rule

Mathematical Formulation

- Let $\mathcal{C}$ denote the set of classes.
- For each training example $t = (\mathbf{x}_i, y_i)$, add t to the set T.
- Given a data point $\mathbf{q}$ to be classified.
- Let $\mathbf{x}_1, \ldots, \mathbf{x}_k$ be the k training examples in T nearest to q.
- Return

$$\hat{y}(q) = \arg\max_{c \in \mathcal{V}} \sum_{i=1}^{k} \delta(c, y_i) \tag{1}$$

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

k-NN Ilustration

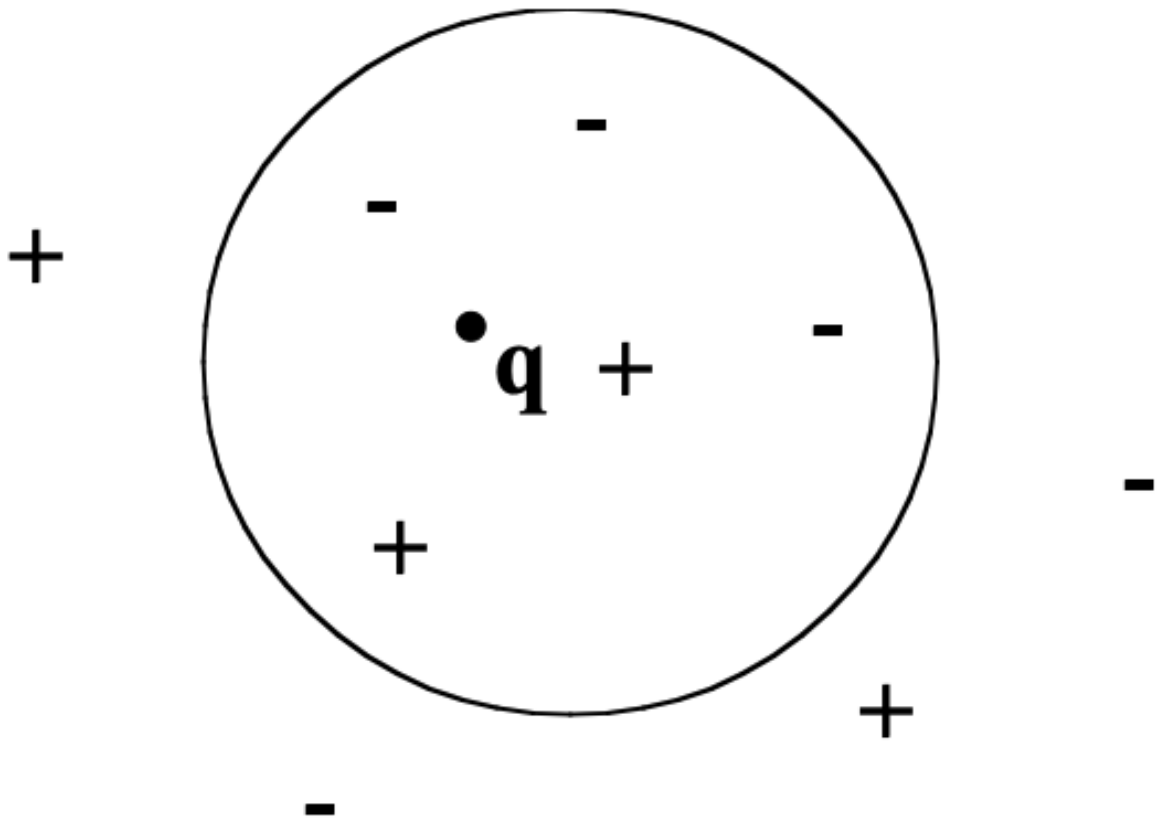

Figure: q is + under 1-NN, but – under 5-NN [1].

Distance-weighted k-NN

- Given a data point q to be classified.
- Let $\mathbf{x}_1, \ldots, \mathbf{x}_k$ be the k training examples in T nearest to q.
- Return

$$\hat{y}(q) = \arg\max_{c \in \mathcal{V}} \sum_{i=1}^{k} \frac{1}{d\left(\mathbf{x}_i, \mathbf{x}_q\right)^2} \delta\left(c, y_i\right) \tag{2}$$

- The vote of each nearest neighbor is scaled by $\frac{1}{d(\mathbf{x}_i, \mathbf{x}_q)^2}$: nearer neighbors will have higher weights.

Value ranges of features

- Features have different value ranges
  - 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
  - Areas will play a much more important role than the number of floors when we calculate distances between data points
  - May bias the performance of the classifier
- We need to scale features so that all features are on the same scale.
- In the next slides, we will present two scaling techniques
  - Standard scaling
  - Min-max scaling

Standard Scaling in `scikit-learn`

What is Standard Scaling?

- Standard Scaling (or Standardization) refers to the process of transforming the features of a dataset to have a mean of 0 and a standard deviation of 1.
- This scaling technique is also called Z-score normalization.
- This is commonly used in preprocessing data for machine learning models, especially when the models are sensitive to the scale of the input features, such as Support Vector Machines (SVM) and k-Nearest Neighbors (k-NN).

Why Standard Scaling?

- Ensures that all features contribute equally to the model.
- Improves convergence of gradient descent in optimization algorithms.
- Prevents features with larger scales from dominating the model's learning process.

Standardizing Numerical Variables – Standard scaling

The scaled feature value can be written as

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

Where:

- z: scaled feature value
- x: input feature value.
- μ: mean of the feature values.
- σ: standard deviation of the feature values-.

Standard scaling example

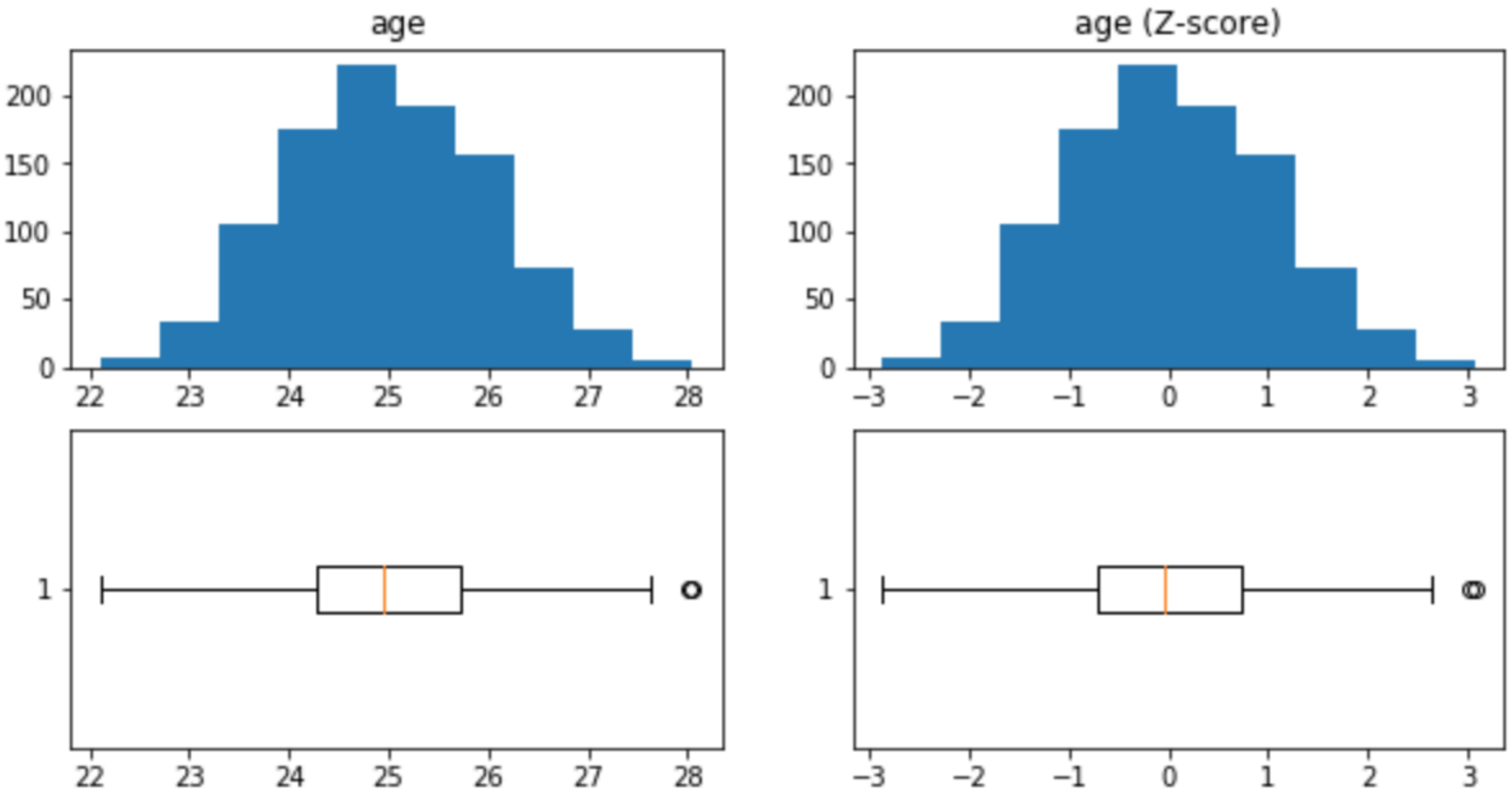

Figure: Z-score transformation on age feature

Standard Scaling in scikit-learn

How to Apply Standard Scaling in scikit-learn:

```
from sklearn.preprocessing import StandardScaler

# Create an instance of the StandardScaler
scaler = StandardScaler()

# Fit the scaler to the data and transform it
scaled_data = scaler.fit_transform(data)

# Alternatively, you can fit and transform separately
# Compute the mean and std for scaling
scaler.fit(data)
# Apply the scaling
scaled_data = scaler.transform(data)
```

Standard Scaling in `scikit-learn`

How to Apply Standard Scaling in `scikit-learn`:
Key Points:

- `fit()`: Computes the mean and standard deviation for scaling based on the training data.
- `transform()`: Scales the data using the mean and standard deviation computed during `fit()`.
- `fit_transform()`: Combines both steps, fitting the scaler and then transforming the data.

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

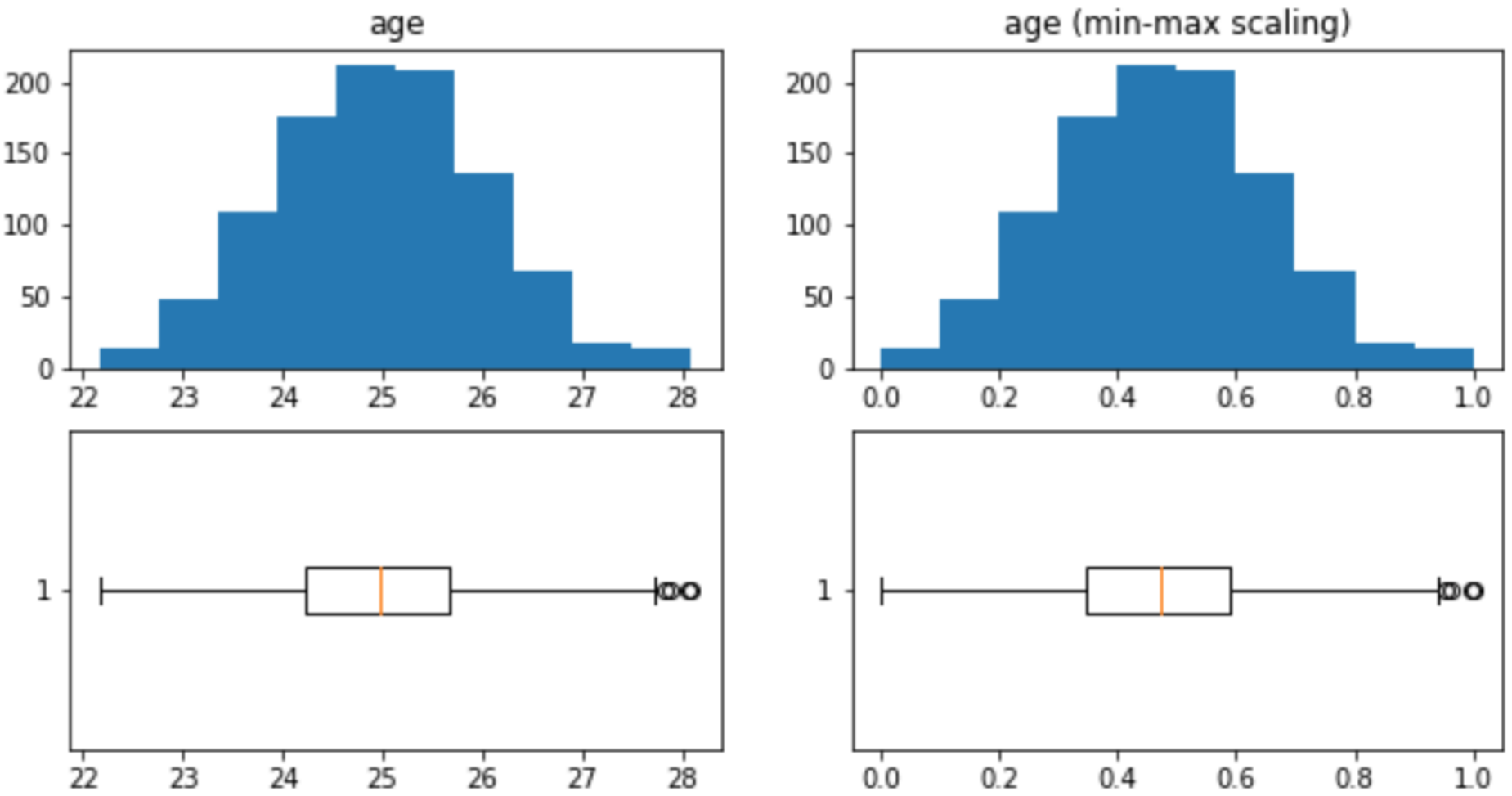

Figure: Min-Max scaling on age feature

Min-Max Scaling in scikit-learn

```
from sklearn.preprocessing import MinMaxScaler

# Sample data
data = [[-1, 2], [-0.5, 6], [0, 10], [1, 18]]

# Create an instance of MinMaxScaler with the desired ra
scaler = MinMaxScaler(feature_range=(0, 1))

# Fit the scaler to the data and transform it
scaled_data = scaler.fit_transform(data)

print("Original Data:\n", data)
print("Scaled Data:\n", scaled_data)
```

Min-Max Scaling in `scikit-learn`

Key Points:

- `feature_range=(0, 1)`: Specifies the range for the scaled data. The default is [0, 1].
- `fit_transform()`: Fits the scaler to the data and transforms it in a single step.
- `transform()`: Applies the scaling to new data using the previously computed parameters.

Standard Scaling vs Min-max Scaling

- Use Standard Scaling when your data is normally distributed or when using algorithms that assume normality, such as SVMs or linear models.
  - Algorithms that assume normality, like SVMs or linear models, often work better when the data is centered. Many optimization algorithms used in these models, like gradient descent, converge faster when the data is centered around zero.
  - Numerical Computations: When the data is centered and scaled, the algorithms are less prone to issues like numerical overflow or underflow, which can occur with very large or very small feature values.
  - Gradient Descent: In optimization algorithms like gradient descent, the speed of convergence is greatly influenced by the scale of the data. If features are on different scales, the gradients will differ significantly, leading to slow or unstable convergence.
- Use Min-Max Scaling when your data does not follow a normal distribution, especially when using algorithms like neural networks or k-NN that perform better with data in a bounded range.

Some remarks

- The k-NN algorithm is widely used in various practical applications, such as pattern recognition, image classification, and recommendation systems, because of its simplicity and effectiveness.
- k-NN algorithm is subject to the *curse of dimensionality*. In high-dimensional spaces, the concept of distance becomes less meaningful because the distance between any two points becomes similar, making it difficult for k-NN to identify true neighbors.
- k-NN relies on a distance metric to measure the similarity between instances. The most commonly used distance metric is Euclidean distance, but other metrics like Manhattan, Minkowski, or cosine similarity can also be used.
- The k-NN algorithm can be computationally expensive, especially when applied to large datasets. For each query point, the algorithm needs to calculate the distance to every point in the training set, which can be slow if the dataset is large. To improve efficiency, k-NN often relies on advanced data structures such as k-d trees or Ball trees.

k-NN regression

- For each training instance $t = (\mathbf{x}_i, y_i)$, add t to the set T.
- Given a data point q to be predicted. Let $\mathbf{x}_1, \ldots, \mathbf{x}_k$ be the k training instances in T nearest to q. Return

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

  where w_i is the weight for training example i.
- Note: The equal-weight case corresponds to $w_i = 1$ for all i.

How can we choose the value of k? [2]

- Experiment with different values of k and evaluate which one performs best on the test set.
  - This approach is possible, but it would result in selecting the optimal k specifically for our test set. Consequently, the test set performance would likely overestimate how well the model would perform on new, unseen data.
- Alternative approach: Reserve a separate validation set (different from both the training and test sets) to determine the optimal k based on its performance on the validation set, while still reporting the final results on the test set.
  - Typically, the model's performance on the validation set will be better than on the test set.
  - How does this affect performance on the training set?

How does the model change with k?

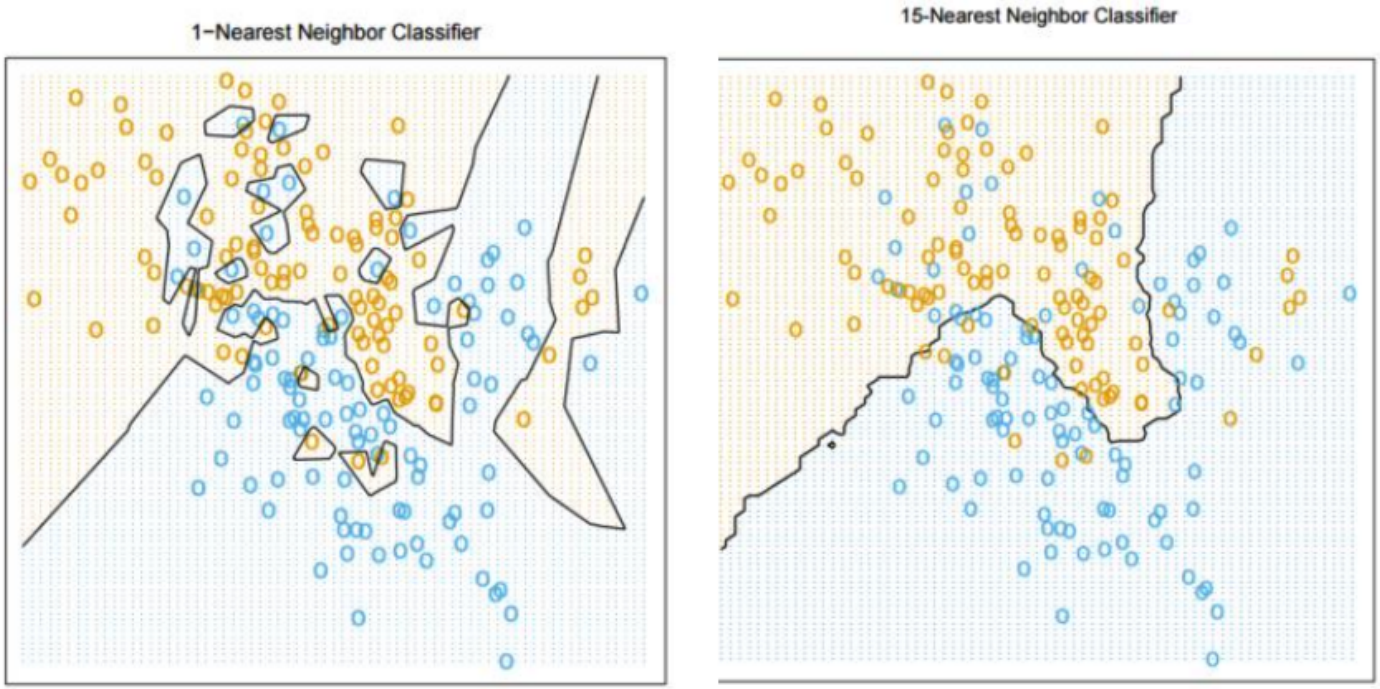

Figure: $k = 1$ (left) and $k = 15$ (right) [2]

- Large k (right): a simple boundary, no small "islands" decision regions. Small changes in $\mathbf{x}$ do not lead to the changes in the prediction.
- Small k (left): a complex boundary in the decision regions. Small changes in x often lead to the changes in the prediction.

When we choose k to be very large

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

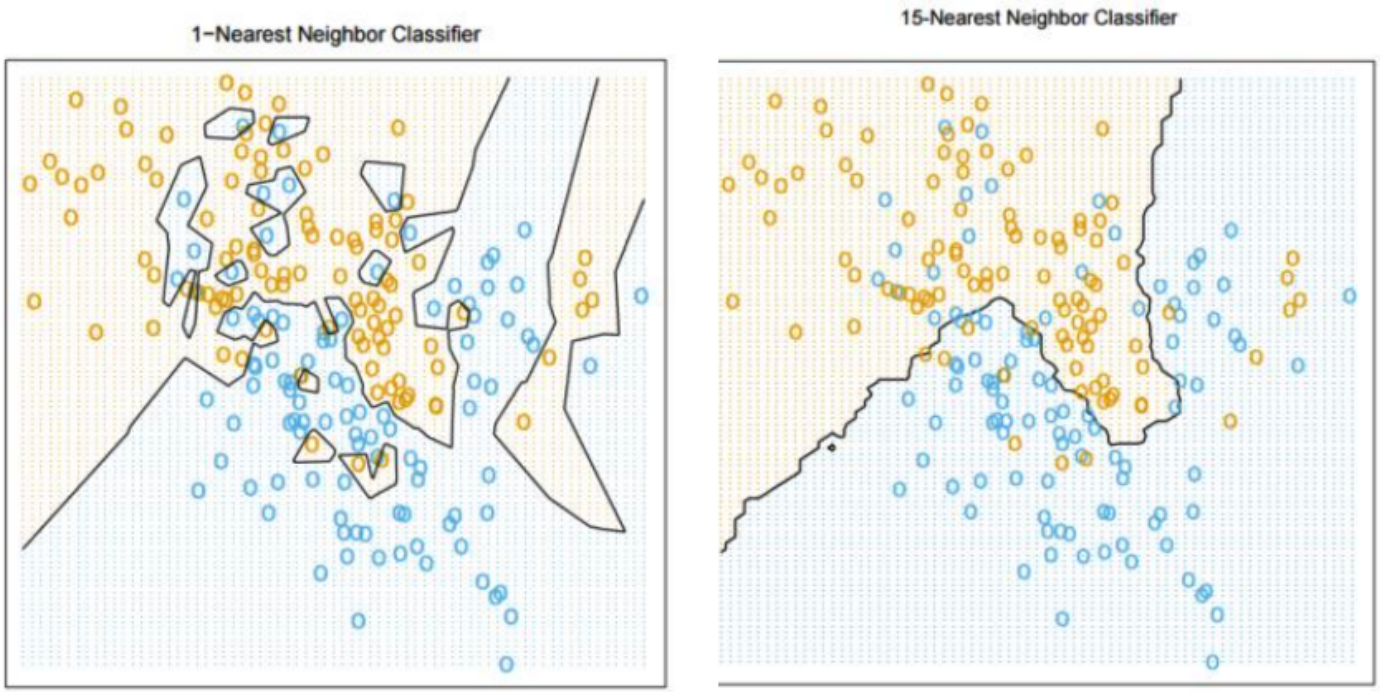

Figure: $k = 1$ (left) and $k = 15$ (right) [2]

When we choose k to be very small

- Excellent results on the training set!
  - For $k = 1$, flawless performance is guaranteed
- However, if the test data differs from the training data, the performance on the test set will likely decline when k is too small
  - The training data might contain noise (for instance, in the orange region, some data points could randomly be blue 5% of the time)
  - This scenario illustrates overfitting – creating a classifier that excels on the training data but fails to generalize effectively to the test data

Müller & Guido's book [3]

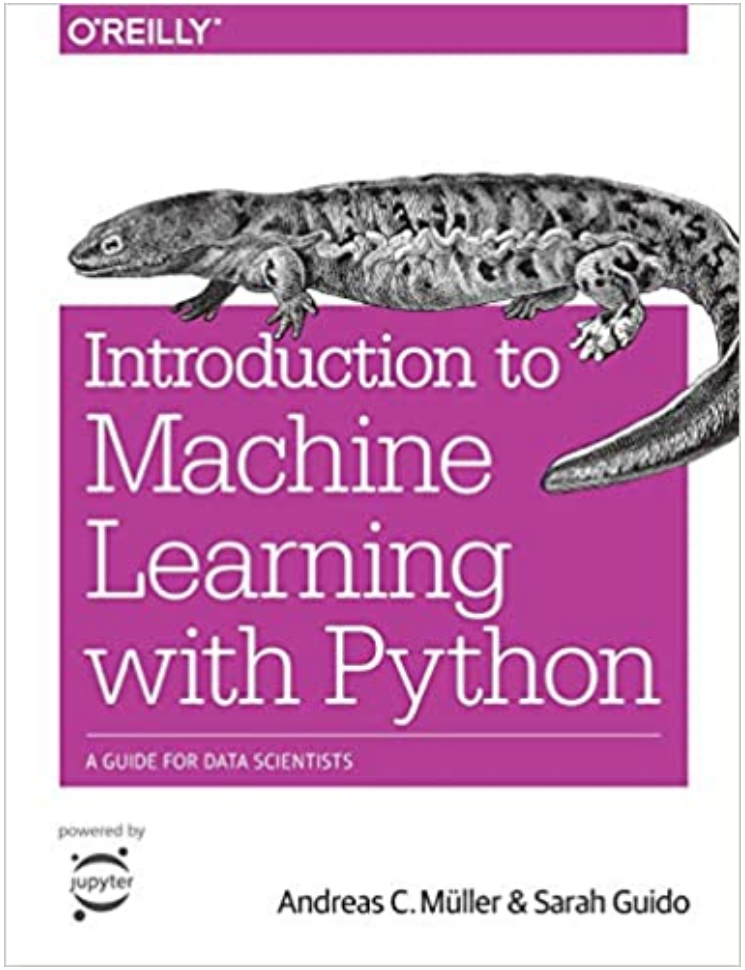

Figure: Introduction to Machine Learning with Python

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

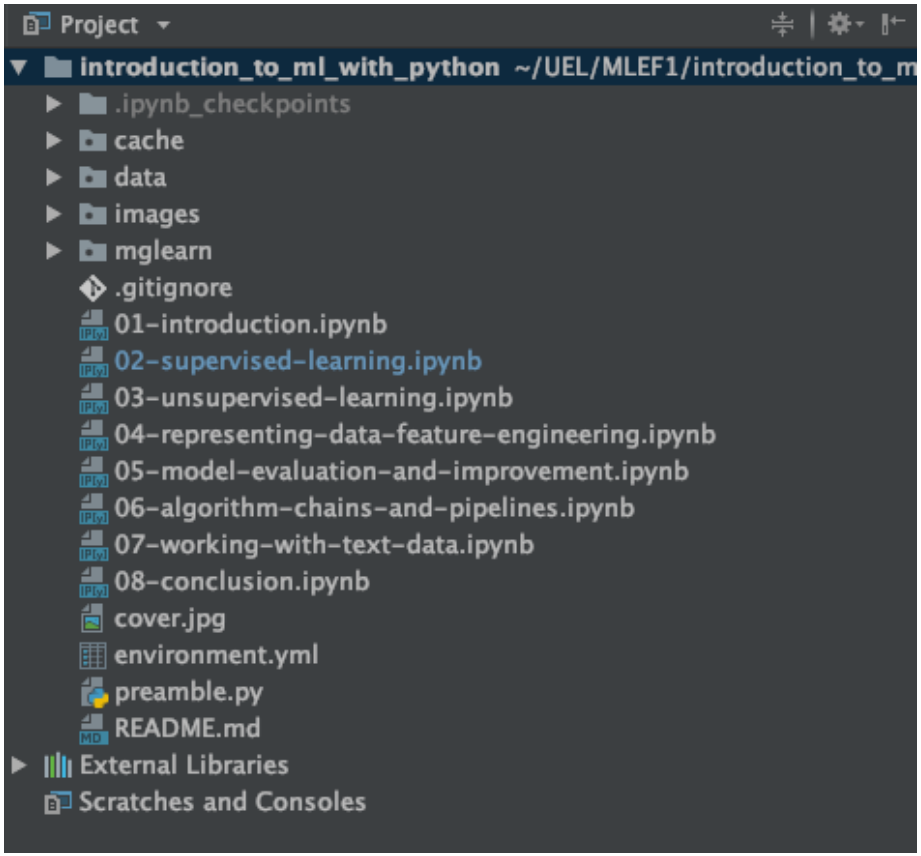

Figure: Introduction to Machine Learning with Python – git repo

```
# 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)
```

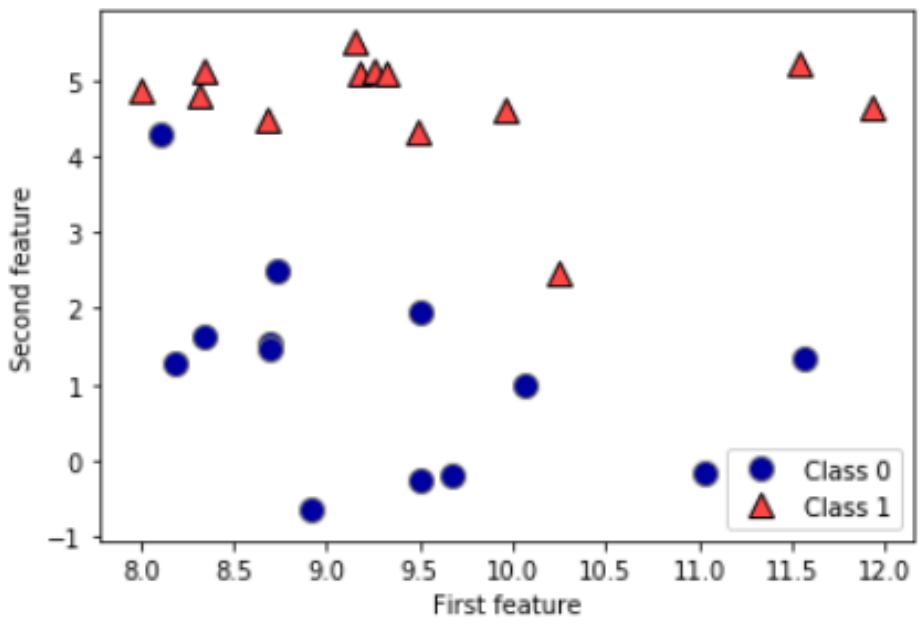

Figure: Visualization of the dataset

1-NN

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

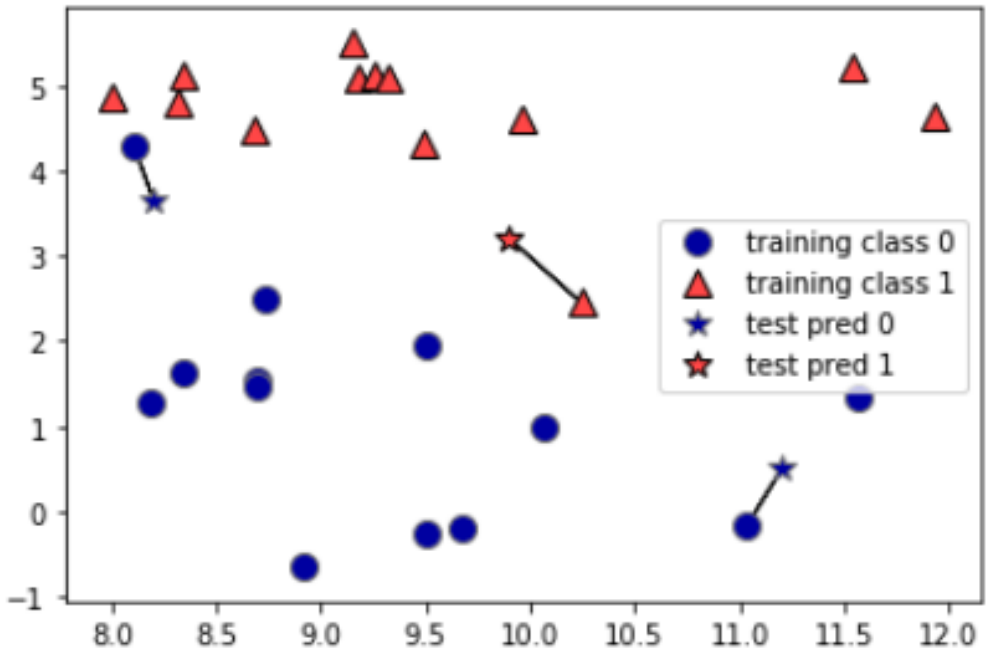

Figure: Predictions made by the one-nearest-neighbor model on the forge dataset

3-NN

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

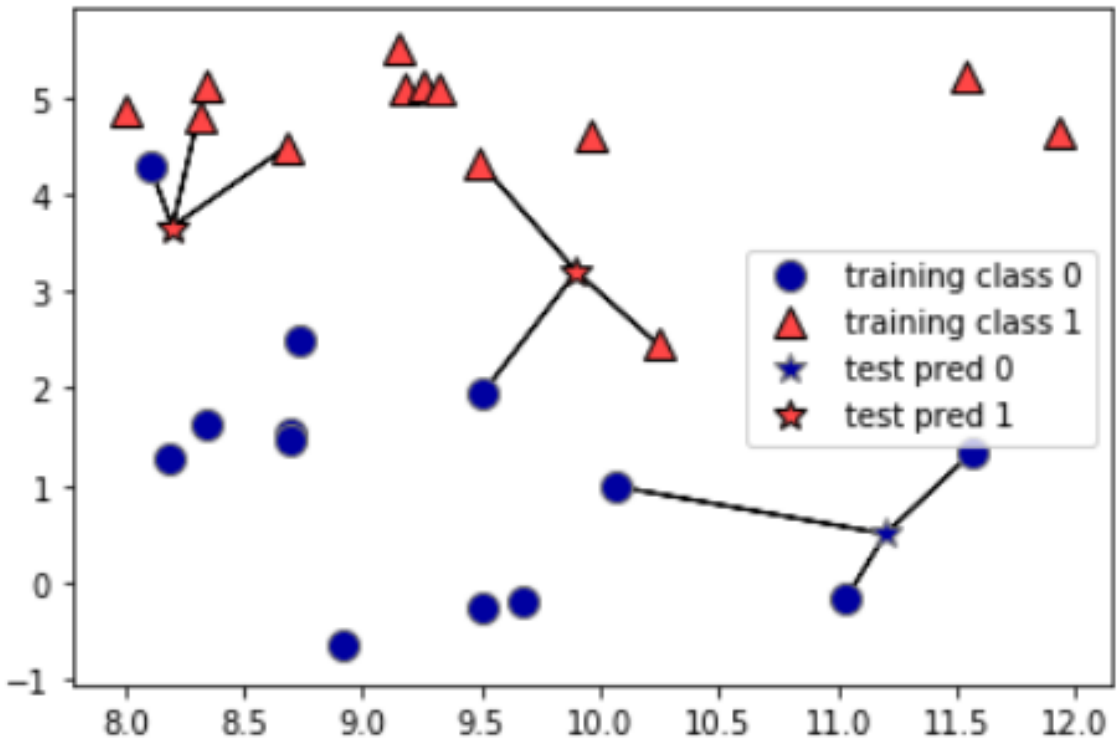

Figure: Predictions made by the three-nearest-neighbor model on the forge dataset

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