
# COMPSCI 389
# Introduction to Machine Learning

**Nearest Neighbor Variants**

Prof. Philip S. Thomas (pthomas@cs.umass.edu)

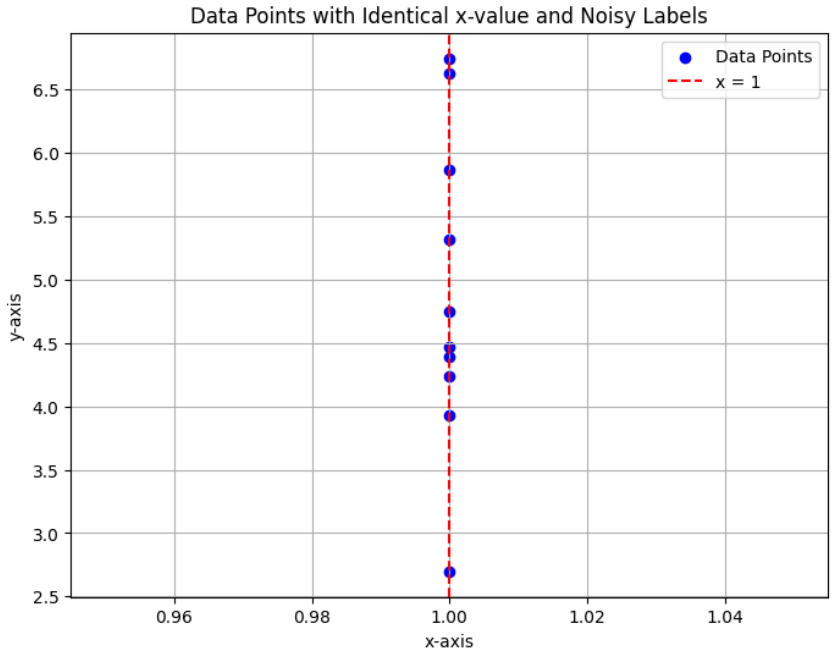
# Nearest Neighbor

- Find the point in the training data that is “closest” to the query.
- Return the corresponding label.
- If two points are equally close, select the label of any one point.
- Pros:
  - Simple
  - Sometimes all that you need!
- Cons:
  - Not always accurate, even with large amounts of data

```python
import pandas as pd
from sklearn.neighbors import KDTree
from sklearn.base import BaseEstimator
import numpy as np

class NearestNeighbor(BaseEstimator):
    def fit(self, X, y):
        # Convert X and y to NumPy arrays if they are DataFrames.
        if isinstance(X, pd.DataFrame):
            X = X.values
        if isinstance(y, pd.Series):
            y = y.values

        # Store the training data and labels.
        self.X_data = X
        self.y_data = y

        # Create a KDTree for efficient nearest neighbor search
        self.tree = KDTree(X)

        return self

    def predict(self, X):
        # Convert X to a NumPy array if it's a DataFrame
        if isinstance(X, pd.DataFrame):
            X = X.values

        # We will iteratively load predictions, so it starts empty
        predictions = []

        # Loop over rows in the query
        for x in X:
            # Query the tree for the nearest neighbor
            dist, ind = self.tree.query([x], k=1)
            nearest_label = self.y_data[ind[0][0]]
            predictions.append(nearest_label)

        # Return the array of predictions we have created
        return np.array(predictions)
```

# Nearest Neighbor Improvements

- Notions of “distance” that are customized to the data set.
- Better handling of the case where many points are equally “close” to the query:

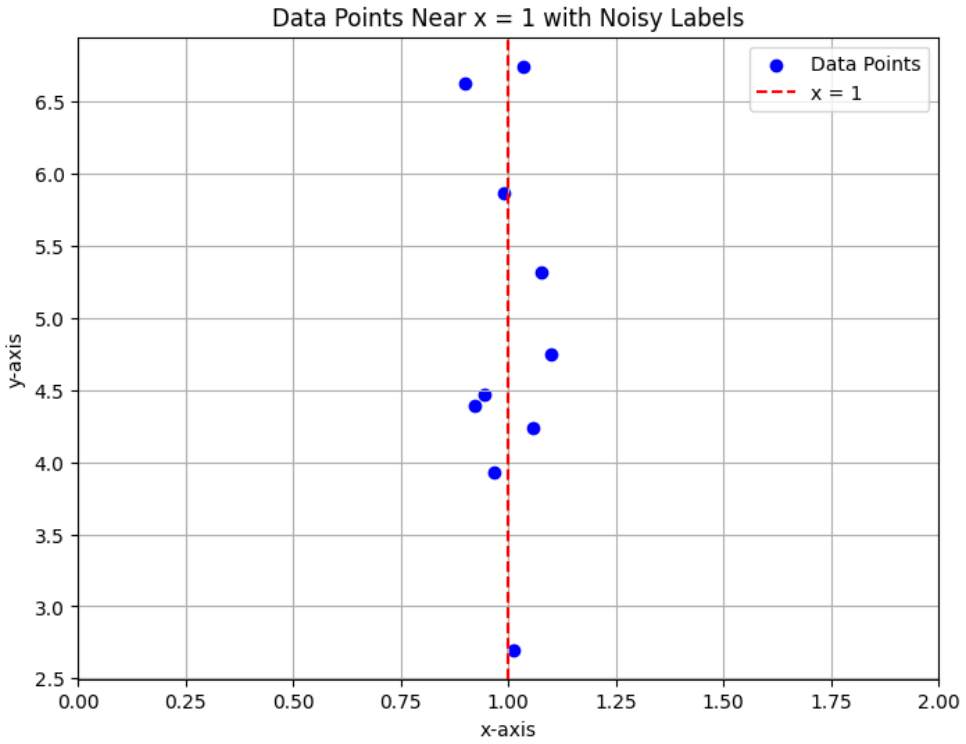



- Query = 1.0 (horizontal axis)
- All points are zero distance from the query.
- Points have different labels (vertical axis).
- With infinite data, the basic NN algorithm will not make reasonable predictions.
- What should we do in this case?
  - Take the average of the labels (or median)

When is the median more appropriate?

# Nearest Neighbor Improvements

- What if the points are not equally close, but many points are close?




- Only showing the closest points to the query.
  - Imagine many points for other input values (horizontal axis), which are not shown.
- **Idea**: Average the labels of the $k$ nearest points
  - $k$ is an integer hyperparameter.

# Hyperparameter

- A **hyperparameter** of an ML algorithm is a variable, like $k$, that changes the behavior of the algorithm.
  - It is often set by the data scientist applying the algorithm.
  - It is not “learned” by the algorithm.

# $k$-Nearest Neighbors (k-NN)

- **Idea**: Average the labels of the $k$ nearest points
- Pseudocode:
  - Find the $k$ nearest neighbors to the query point.
    - Called the “nearest neighbors”
    - If you will run many queries, consider using a data structure like a KD-Tree to find the nearest neighbors
  - Set the prediction to be the average label of these $k$ nearest neighbors.
- Code:

```
class KNearestNeighbors(BaseEstimator):
    # Add a constructor that stores the value of k (a hyperparameter)
    def __init__(self, k=3):
        self.k = k
```

Hyperparameter,
default value $k = 3$

# Fit: Unchanged from NN

```
def fit(self, X, y):
    # Convert X and y to NumPy arrays if they are DataFrames
    if isinstance(X, pd.DataFrame):
        X = X.values
    if isinstance(y, pd.Series):
        y = y.values

    # Store the training data and labels
    self.X_data = X
    self.y_data = y

    # Create a KDTree for efficient nearest neighbor search
    self.tree = KDTree(X)

    return self
```

```python
def predict(self, X):
    # Convert X to a NumPy array if it's a DataFrame
    if isinstance(X, pd.DataFrame):
        X = X.values

    # Query the tree for the k nearest neighbors for all points in X
    dist, ind = self.tree.query(X, k=self.k)

    # Return the average label for the nearest neighbors of each query
    return np.mean(self.y_data[ind], axis=1)
```

Here `ind` is:

- $(m,k)$ NumPy array
  - $m$: Number of query points
  - $k$: The number of nearest neighbors
- `ind[i,j]`: The index of the $j^{\text{th}}$ nearest neighbor to the $i^{\text{th}}$ query point.

Here `self.y_data[ind]` is:

- $(m,k)$ NumPy array
  - $m$: Number of query points
  - $k$: The number of nearest neighbors
  - $[i,j]$: The label for the $j^{\text{th}}$ nearest neighbor to the $i^{\text{th}}$ query point.

```
array([[3.16667 , 3.2     , 3.73333 ],
       [3.55667 , 2.14333 , 1.33333 ],
       [3.23667 , 2.5     , 2.93333 ],
       ...,
       [0.973333, 2.60333 , 3.37667 ],
       [3.15    , 2.95    , 2.28    ],
       [3.42    , 3.31333 , 2.53667 ]])
```

`np.mean(self.y_data[ind], axis=1)`

- Axis=1 specifies to take the mean for each *row*.
- Computes the average of the $k$ labels for each query.
- Returns an array with one value per query (per row of `self.y_data[ind]`)

# NN vs $k$-NN (metrics)

```python
def mean_squared_error(predictions, labels):
    return np.mean((predictions - labels) ** 2)

def root_mean_squared_error(predictions, labels):
    return np.sqrt(mean_squared_error(predictions, labels))

def mean_absolute_error(predictions, labels):
    return np.mean(np.abs(predictions - labels))

def r_squared(predictions, labels):
    ss_res = np.sum((labels - predictions) ** 2)        # ss_res is the "Sum of Squares of Residuals"
    ss_tot = np.sum((labels - np.mean(labels)) ** 2)    # ss_tot is the "Total Sum of Squares"
    return 1 - (ss_res / ss_tot)
```

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2,$$

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}.$$

$$\mathrm{MAE} = \frac{1}{n}\sum_{i=1}^{n}\left|y_i - \hat{y}_i\right|.$$

$$R^2 = 1 - \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2},$$

# NN vs $k$-NN (data and train/test split)

```
from sklearn.model_selection import train_test_split

# Load the data set
df = pd.read_csv("data/GPA.csv", delimiter=',')

# We already loaded X and y, but do it again as a reminder
X = df.iloc[:, :-1]
y = df.iloc[:, -1]

# Split the data into training and testing sets (60% train, 40% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4, shuffle=True)
```

Why shuffle=True?

# NN vs $k$-NN

```
# List of values of k to test
k_values = [1, 2, 3, 5, 10, 100, 1000, 5000]

# List to store the results. This will be a list of dictionaries
results_list = []
```

We will load `results_list` with

- A list of dictionaries
- Each dictionary is a column name and a value
- Should look like this
- Can be converted to a DataFrame with:

```
# Create DataFrame from the list of results.
results = pd.DataFrame(results_list)

# Print the results
display(results)
```

```
[{'k': 1,
  'MSE': 1.1520837122547205,
  'RMSE': 1.073351625635663,
  'MAE': 0.8237428042373861,
  'R^2': -0.6877687897963096},
 {'k': 2,
  'MSE': 0.8534297082546841,
  'RMSE': 0.9238125936869902,
  'MAE': 0.713553491369357,
  'R^2': -0.2502494485042783},
 {'k': 3,
  'MSE': 0.7644680530281004,
  'RMSE': 0.8743386375015691,
  'MAE': 0.678162021552554,
  'R^2': -0.1199232732453802},
 {'k': 5,
  'MSE': 0.6883302537154519,
  'RMSE': 0.829656708353191,
  'MAE': 0.644951049894931 3,
  'R^2': -0.0083835982889575 17},
 {'k': 10,
```

# Desired Result:

| | k | MSE | RMSE | MAE | R^2 |
|---|---|---|---|---|---|
| 0 | 1 | 1.152084 | 1.073352 | 0.823743 | -0.687769 |
| 1 | 2 | 0.853430 | 0.923813 | 0.713553 | -0.250249 |
| 2 | 3 | 0.764468 | 0.874339 | 0.678162 | -0.119923 |
| 3 | 5 | 0.688330 | 0.829657 | 0.644951 | -0.008384 |
| 4 | 10 | 0.631001 | 0.794356 | 0.620237 | 0.075602 |
| 5 | 100 | 0.579404 | 0.761186 | 0.596919 | 0.151190 |
| 6 | 1000 | 0.581676 | 0.762677 | 0.600227 | 0.147861 |
| 7 | 5000 | 0.600544 | 0.774947 | 0.616670 | 0.120221 |

# Loading `results_list`:

```
# Evaluate NN and k-NN models
for k in k_values:
    model = KNearestNeighbors(k=k)
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)

    mse = mean_squared_error(predictions, y_test)
    rmse = root_mean_squared_error(predictions, y_test)
    mae = mean_absolute_error(predictions, y_test)
    r2 = r_squared(predictions, y_test)

    # Create a dictionary with the relevant variables from this value of k, and add it to results_list
    results_list.append({'k': k, 'MSE': mse, 'RMSE': rmse, 'MAE': mae, 'R^2': r2})
```

```
# List of values of k to test
k_values = [1, 2, 3, 5, 10, 100, 1000, 5000]

# List to store the results. This will be a list of dictionaries
results_list = []

# Evaluate NN and k-NN models
for k in k_values:
    model = KNearestNeighbors(k=k)
    model.fit(X_train, y_train)
    predictions = model.predict(X_test)

    mse = mean_squared_error(predictions, y_test)
    rmse = root_mean_squared_error(predictions, y_test)
    mae = mean_absolute_error(predictions, y_test)
    r2 = r_squared(predictions, y_test)

    # Create a dictionary with the relevant variables from this value of k, and add it to results_list.
    results_list.append({'k': k, 'MSE': mse, 'RMSE': rmse, 'MAE': mae, 'R^2': r2})

# Create DataFrame from the list of results. Each dictionary in the list becomes a row in the DataFrame
results = pd.DataFrame(results_list)

# Print the results
display(results)
```

# Highlighting best values (see .ipynb for code)

| | k | MSE | RMSE | MAE | R^2 |
|---|---|---|---|---|---|
| 0 | 1 | 1.152084 | 1.073352 | 0.823743 | -0.687769 |
| 1 | 2 | 0.853430 | 0.923813 | 0.713553 | -0.250249 |
| 2 | 3 | 0.764468 | 0.874339 | 0.678162 | -0.119923 |
| 3 | 5 | 0.688330 | 0.829657 | 0.644951 | -0.008384 |
| 4 | 10 | 0.631001 | 0.794356 | 0.620237 | 0.075602 |
| 5 | 100 | **0.579404** | **0.761186** | **0.596919** | **0.151190** |
| 6 | 1000 | 0.581676 | 0.762677 | 0.600227 | 0.147861 |
| 7 | 5000 | 0.600544 | 0.774947 | 0.616670 | 0.120221 |

Question: Is it surprising that $k = 100$ performs well here?

# Nearest Neighbor Improvements (Part 2)

- When does $k$-NN do something unreasonable?
  - When some of the nearest neighbor are very close and some are relatively far away.
  - $k$-NN does not consider how far away points are as long as they are within the $k$ nearest neighbors.
- **Idea**: Assign different weights to each of the $k$ neighbors based on their distance from the query point.
  - Called "Weighted $k$-Nearest Neighbors" (weighted $k$-NN)
  - Ensures that closer neighbors have a bigger influence on the prediction than neighbors that are far away.

# Weighted $k$-Nearest Neighbor

- Let $\left(x_i^{NN}, y_i^{NN}\right)$ be the $i^{\text{th}}$ nearest neighbor
- Let $w_i$ be the weight associated with this point
  - We consider only non-negative weights: $w_i \geq 0$.
  - We describe how $w_i$ can be computed on future slides.
- Weighted $k$-NN predicts the label:

$$\hat{y} = \frac{\sum_{i=1}^{k} w_i y_i^{NN}}{\sum_{j=1}^{k} w_j}$$

  - This is equivalent to:

$$\hat{y} = \sum_{i=1}^{k} \frac{w_i}{\sum_{j=1}^{k} w_j} y_i^{NN}$$

# Why divide by the sum of the weights?

$$\hat{y} = \frac{\sum_{i=1}^{k} w_i y_i^{NN}}{\sum_{j=1}^{k} w_j}$$

$$\hat{y} = \sum_{i=1}^{k} \frac{w_i}{\sum_{j=1}^{k} w_j} y_i^{NN}$$

- So that the weights sum to one.
  - This is a **weighted average**.
- Example (without dividing by weights)

$$w_1 = 2, w_2 = 3, y_1^{NN} = 9\ y_2^{NN} = 11$$

$$\hat{y} = \sum_{i=1}^{2} w_i\, y_i^{NN} = 2 \times 9 + 3 \times 11 = 51$$

- Weighted average of 9 and 11 is 51?!
- Example (making weights sum to one):

$$\hat{y} = \sum_{i=1}^{2} \frac{w_i}{\sum_{j=1}^{k} w_j} y_i^{NN} = \frac{2}{2+3} 9 + \frac{3}{2+3} 11 = 10.2$$

- Example normalizing at the end (fewer multiplication/division operations):

$$\hat{y} = \frac{\sum_{i=1}^{2} w_i y_i^{NN}}{\sum_{j=1}^{2} w_j} = \frac{2 \times 9 + 3 \times 11}{(2+3)} = 10.2$$

# Weighting Options

$$\hat{y} = \sum_{i=1}^{k} \frac{w_i}{\sum_{j=1}^{k} w_j} y_i^{NN}$$

- Would it be reasonable to use:

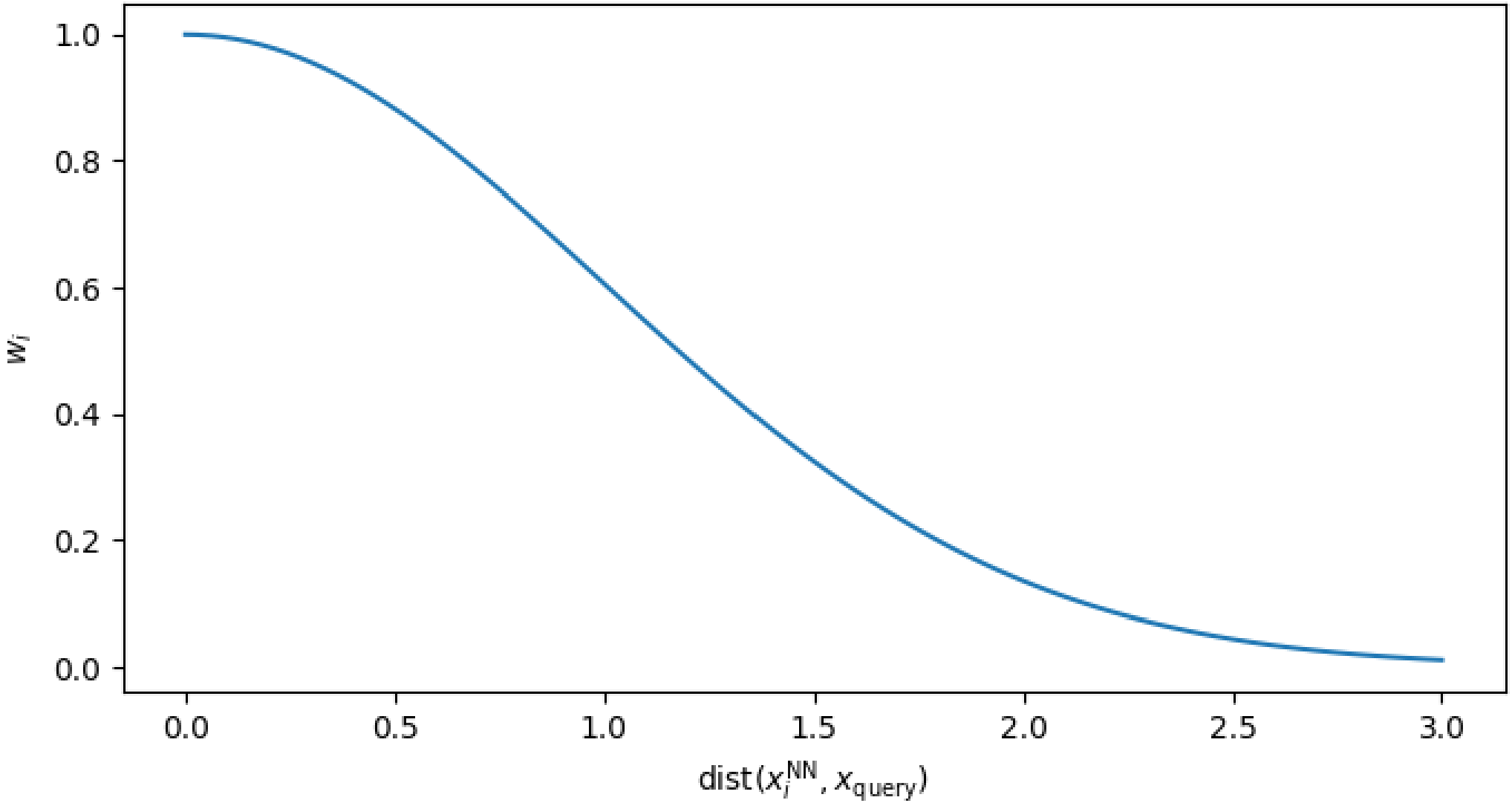
$$w_i = \mathrm{dist}\left(x_i^{NN}, x_{\mathrm{query}}\right)?$$

- No, this would place larger weights on points that are farther from the query.
- We could use the inverse of the distance:

$$w_i = \frac{1}{\mathrm{dist}\left(x_i^{NN}, x_{\mathrm{query}}\right)}$$

- We might want the weight to decrease faster for points that are farther away.

# Possible Weighting Scheme



# Gaussian Kernel

- The re-scaled *probability density function* (PDF) of a normal distribution.
  - PDF of a normal distribution

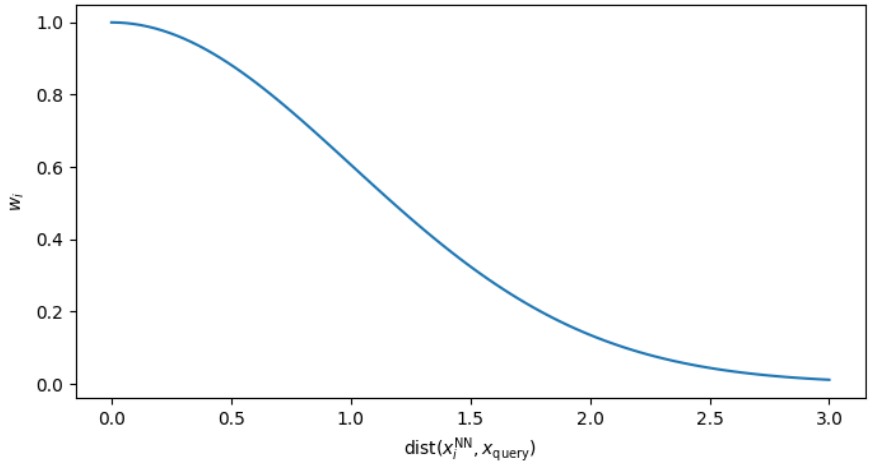
$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

  - Mean $\mu = 0$
  - Standard deviation $\sigma$ (a hyperparameter)
- Normalizing the weights makes the constant $\frac{1}{\sigma\sqrt{2\pi}}$ cancel out in each weight. Hence:

$$w_i = e^{-\frac{x^2}{2\sigma^2}}$$

- We use $x = \text{dist}\left(x_i^{NN}, x_{\text{query}}\right)$ giving:

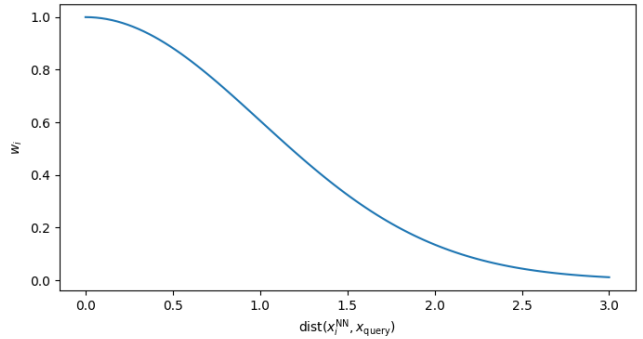
$$w_i = e^{-\frac{\text{dist}\left(x_i^{NN}, x_{\text{query}}\right)^2}{2\sigma^2}}$$



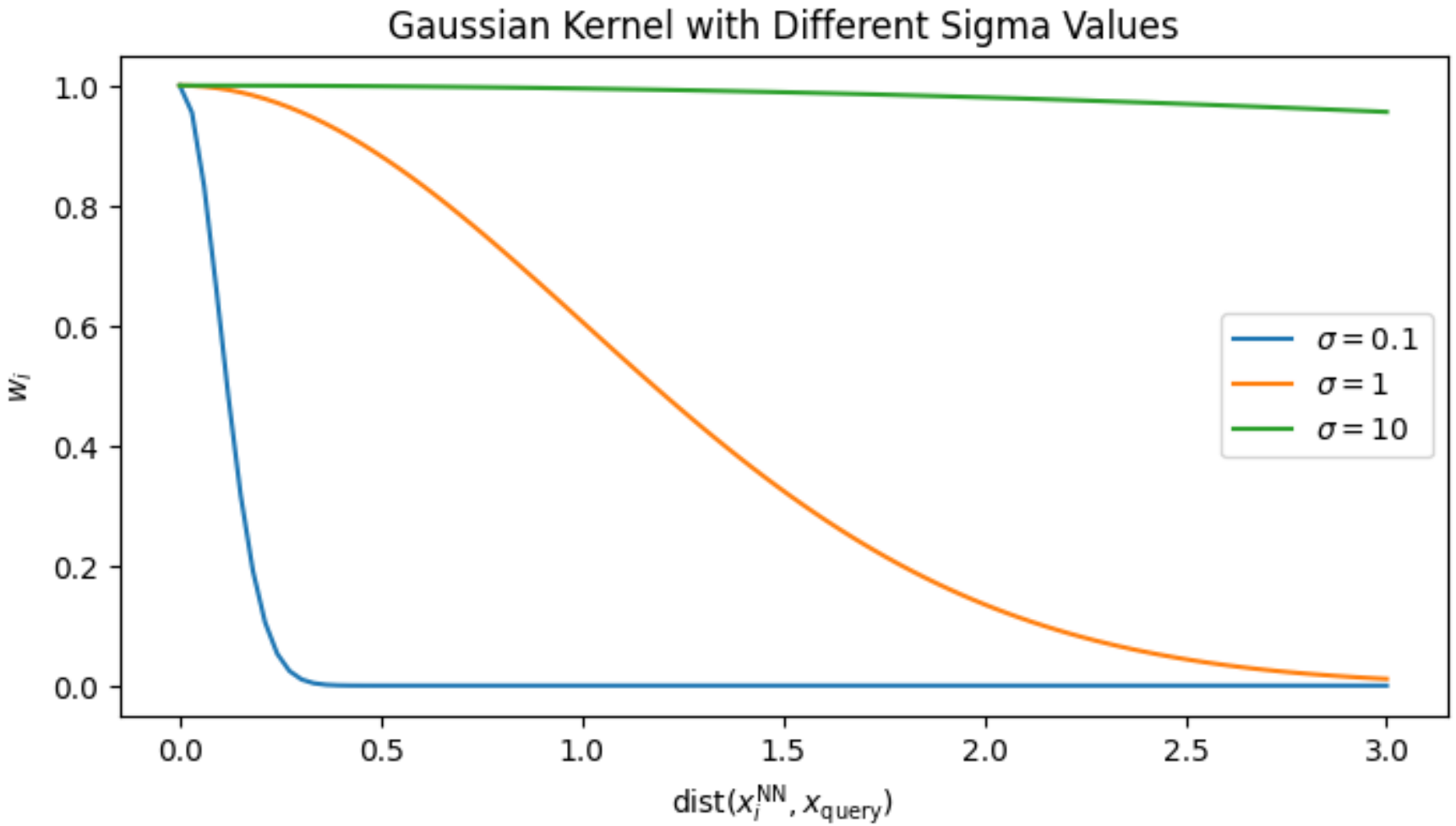
# Gaussian Kernel

$$\hat{y} = \sum_{i=1}^{k} \frac{w_i}{\sum_{j=1}^{k} w_j} y_i^{NN} \qquad w_i = e^{-\frac{\text{dist}\left(x_i^{NN}, x_{\text{query}}\right)^2}{2\sigma^2}}$$

- What is the impact of a bigger value of $\sigma$?
  - It makes the weight curve wider
  - Places larger weights (more emphasis) on points that are farther away
- What is the impact of a smaller value of $\sigma$?
  - It makes the weight curve tighter
  - Places more emphasis on points that are closer

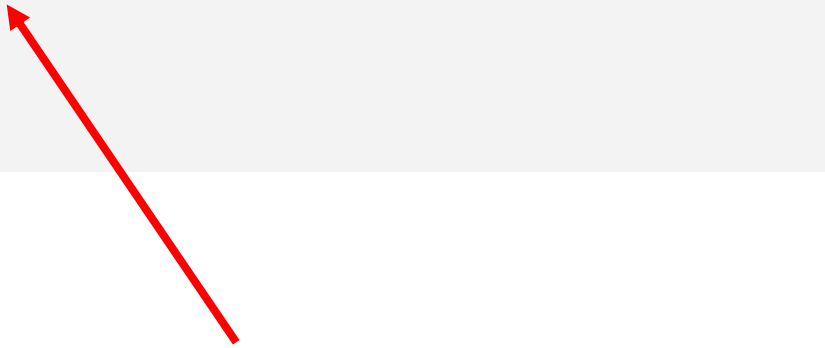


$$\hat{y} = \sum_{i=1}^{k} \frac{w_i}{\sum_{j=1}^{k} w_j} y_i^{NN} \qquad w_i = e^{-\frac{\text{dist}\left(x_i^{NN}, x_{\text{query}}\right)^2}{2\sigma^2}}$$



# Weighted $k$-Nearest Neighbor

```
class WeightedKNearestNeighbors(BaseEstimator):
    # Add a constructor that stores the value of k and sigma (hyperparameters)
    def __init__(self, k=3, sigma=1.0):
        self.k = k
        self.sigma = sigma
```



We now have two hyperparameters, $k$ and $\sigma$

# Fit is unchanged

```
def fit(self, X, y):
    # Convert X and y to NumPy arrays if they are DataFrames
    if isinstance(X, pd.DataFrame):
        X = X.values
    if isinstance(y, pd.Series):
        y = y.values

    # Store the training data and labels
    self.X_data = X
    self.y_data = y

    # Create a KDTree for efficient nearest neighbor search
    self.tree = KDTree(X)

    return self
```

# Gaussian Kernel

$$w_i = e^{-\frac{\text{dist}\left(x_i^{NN}, x_{\text{query}}\right)^2}{2\sigma^2}}$$

```python
def gaussian_kernel(self, distance):
    # Gaussian kernel function
    return np.exp(- (distance ** 2) / (2 * self.sigma ** 2))
```

# Predict

You could compute all `dist, ind` using one call to `tree.query`, not one call per query.

```
array([[100.40687128, 105.49888009, 109.22491566]])
```

```
array([[6178, 1856, 9622]], dtype=int64)
```

```python
def predict(self, X):
    # Convert X to a NumPy array if it's a DataFrame
    if isinstance(X, pd.DataFrame):
        X = X.values

    # We will iteratively load predictions, so it starts empty
    predictions = []

    # Loop over rows in the query
    for x in X:
        # Query the tree for the k nearest neighbors
        dist, ind = self.tree.query([x], k=self.k)

        # Calculate weights using the Gaussian kernel
        weights = self.gaussian_kernel(dist[0])

        # Check if weights sum to zero. This happens when all points are very far
        if np.sum(weights) == 0:
            # If weights sum to zero, assign equal weight to all neighbors
            weights = np.ones_like(weights)

        # Weighted average of the labels of the k nearest neighbors
        weighted_avg_label = np.average(self.y_data[ind[0]], weights=weights)
        predictions.append(weighted_avg_label)

    # Return the array of predictions we have created
    return np.array(predictions)
```

For $0^{th}$ query in `tree.query`

# NN vs $k$-NN vs Weighted $k$-NN

| | Model | MSE | RMSE | MAE | R^2 |
|---|---|---|---|---|---|
| 0 | k-NN k=1 sigma=None | 1.152084 | 1.073352 | 0.823743 | -0.687769 |
| 1 | k-NN k=100 sigma=None | 0.579404 | 0.761186 | 0.596919 | 0.151190 |
| 2 | k-NN k=100 sigma=100 | 0.579572 | 0.761297 | 0.596952 | 0.150943 |
| 3 | k-NN k=200 sigma=100 | 0.577554 | 0.759970 | **0.596220** | 0.153901 |
| 4 | k-NN k=300 sigma=100 | **0.577443** | **0.759897** | 0.596408 | **0.154062** |
| 5 | k-NN k=400 sigma=100 | 0.577620 | 0.760013 | 0.596670 | 0.153804 |
| 6 | k-NN k=500 sigma=100 | 0.578077 | 0.760314 | 0.597044 | 0.153135 |

# Nearest Neighbor Variants

- How can nearest neighbor algorithms be extended to the classification setting?
- **Common Solution**: Use a majority vote among the $k$ nearest neighbors.
  - **Unweighted**: The most common label among the nearest neighbors is selected.
  - **Weighted**: Each neighbor's vote is weighted using, for example, the Gaussian kernel.

# Tuning Hyperparameters

- How should we set $k$ and $\sigma$?
- **Idea**: Enumerate a "grid" of possible values.

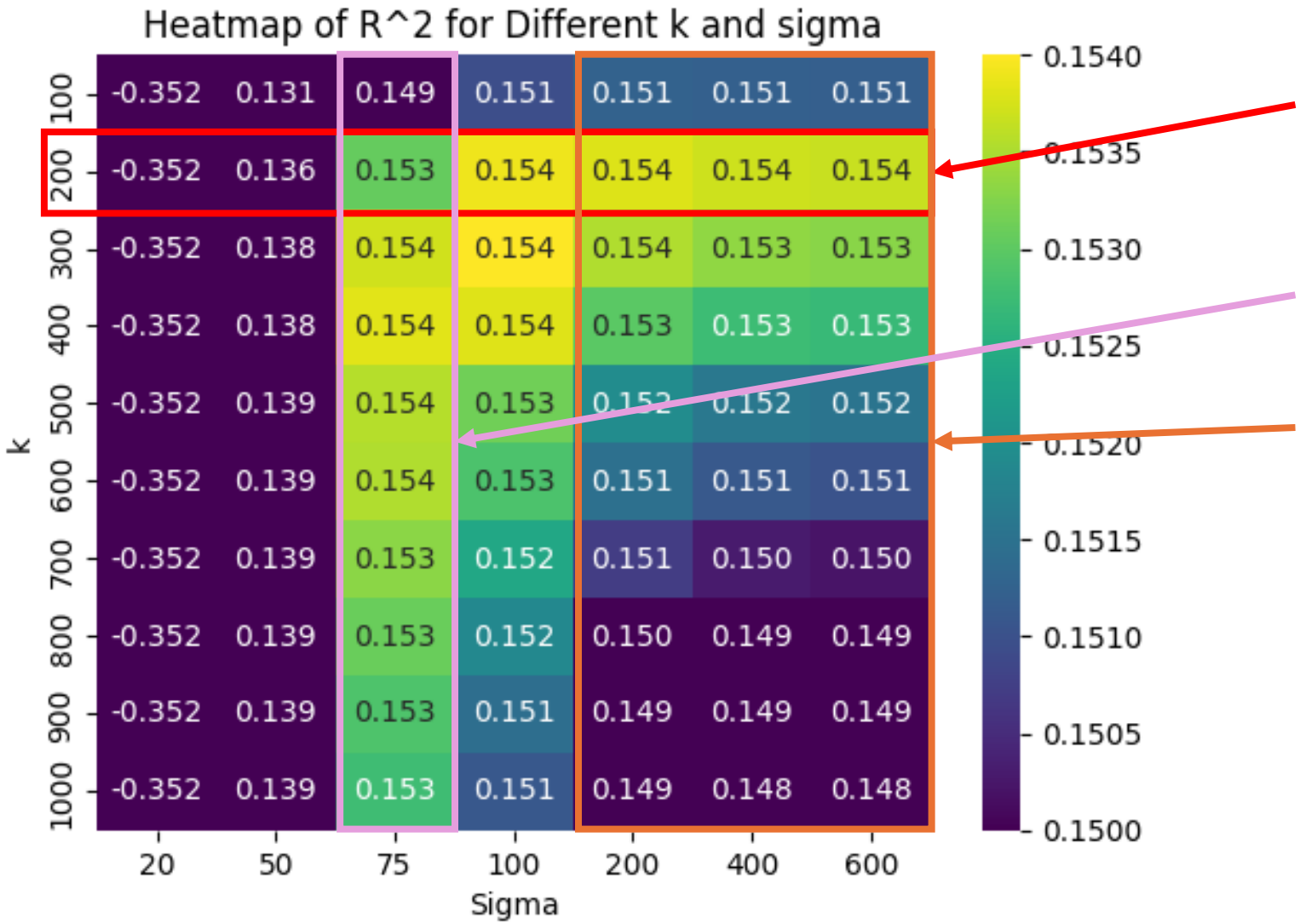
```
# Define the ranges for k and sigma
k_values = [k for k in range(100, 1100, 100)]
sigma_values = [20, 50, 75, 100, 200, 400, 600]
```

- Try all possible combinations of values of $k$ `in k_values` and $\sigma$ in `sigma_values`.
  - If plotted as points where the horizontal axis is $k$ and the vertical is $\sigma$ (or *vice versa*), the points would form a grid.
  - Hence, called "Grid Search"
- Select the values that result in the best evaluation

# Tuning Hyperparameters

- Grid search is common due to its simplicity.
- Research suggests that randomized searches may be more principled.
  - Randomly sample each hyperparameter from some distribution
  - Typically run for some fixed number of hyperparameter settings

# Grid Search Results (Weighted $k$-NN, GPA)




- Why do you think a value of $k$ around 200 is particularly effective for this problem?
- Why do you think $\sigma = 75$ works for a wide range of $k$?
- Why do you think changing $\sigma$ makes little difference when it is large?
- As you work with each ML algorithm, you'll get a sense for how to set the hyperparameters
  - It's an "art" and a "science"