Machine Learning

K-Nearest Neighbours Classification

Tarapong Sreenuch

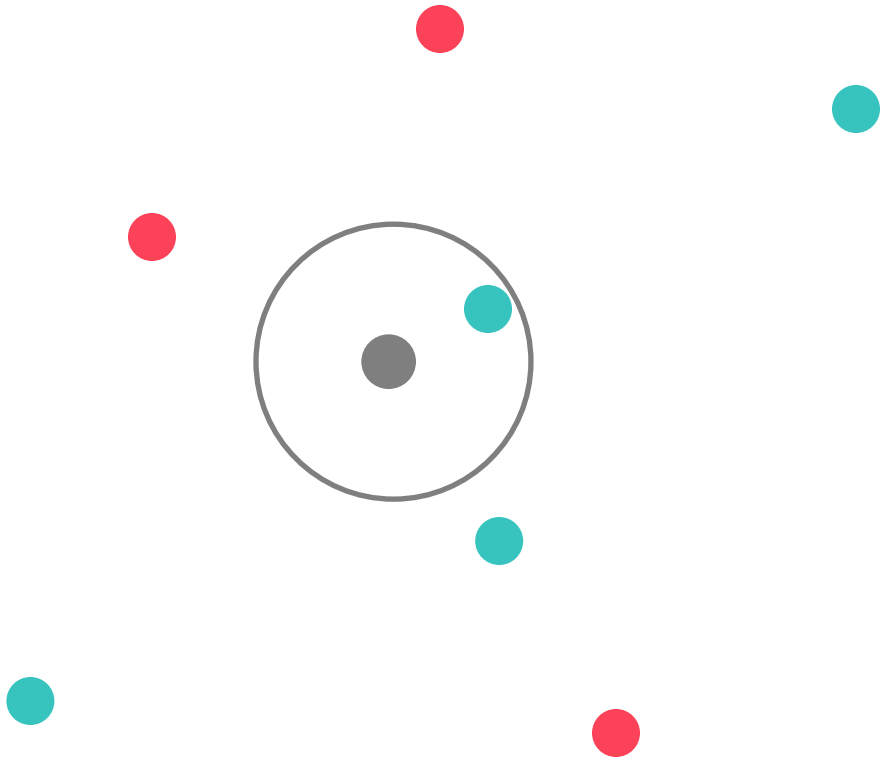
8 February 2024

克明峻德，格物致知


This one looks… different.
Where should it go?
I'll put it with what it most resembles.
If it looks and tastes like a tomato,
it's probably a tomato.
Q: How does KNN decide?
A: We ask the nearest few, and let the majority label it.
No Maths?
トマト
ミニトマト

K-Nearest Neighbours

The KNN algorithm predicts the labels of the test dataset by looking at the labels of its closest neighbors in the feature space of the training dataset. The "K" is the most important hyperparameter that can be tuned to optimize the performance of the model.



K = 1

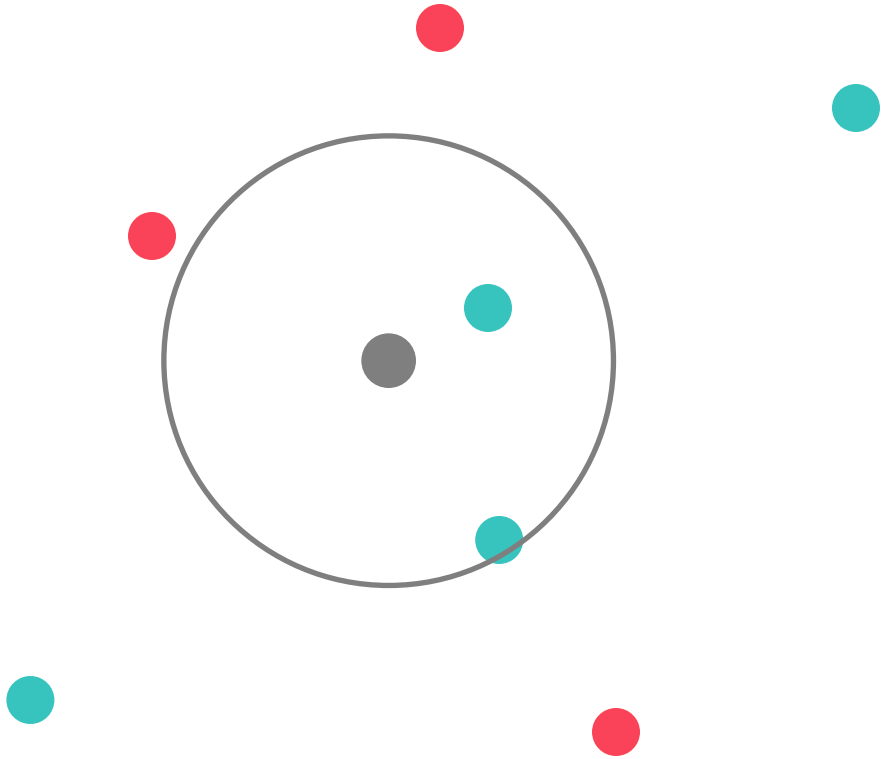
Green = 1

Red = 0

Prediction: Green

K-Nearest Neighbours

The KNN algorithm predicts the labels of the test dataset by looking at the labels of its closest neighbors in the feature space of the training dataset. The "K" is the most important hyperparameter that can be tuned to optimize the performance of the model.



K = 2

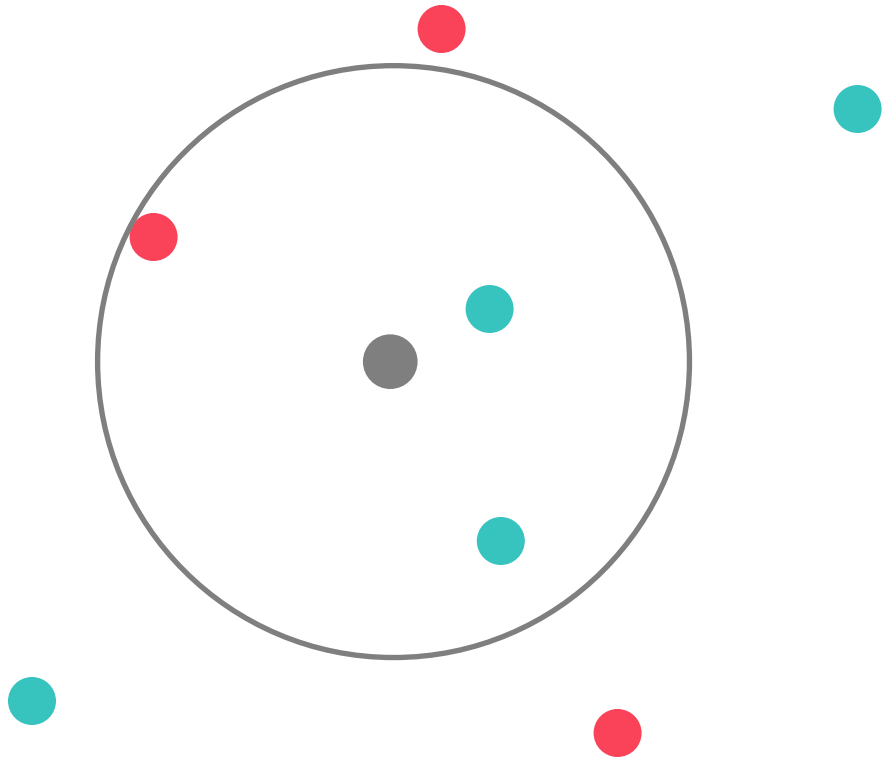
Green = 2

Red = 0

Prediction: Green

K-Nearest Neighbours

The KNN algorithm predicts the labels of the test dataset by looking at the labels of its closest neighbors in the feature space of the training dataset. The "K" is the most important hyperparameter that can be tuned to optimize the performance of the model.



K = 3

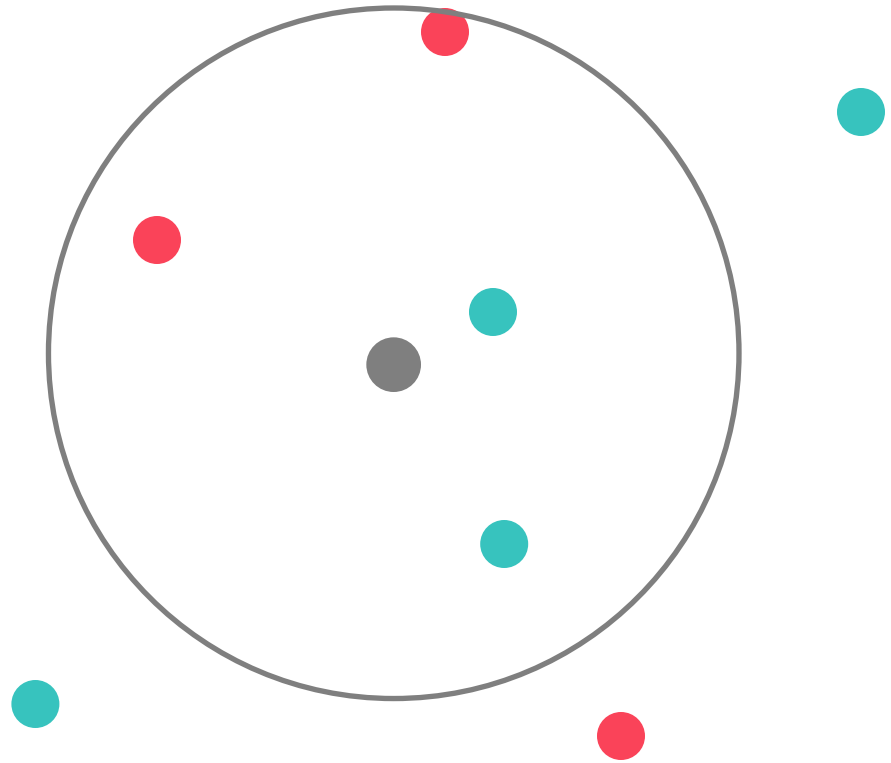
Green = 2

Red = 1

Prediction: Green

K-Nearest Neighbours

The KNN algorithm predicts the labels of the test dataset by looking at the labels of its closest neighbors in the feature space of the training dataset. The "K" is the most important hyperparameter that can be tuned to optimize the performance of the model.



K = 4

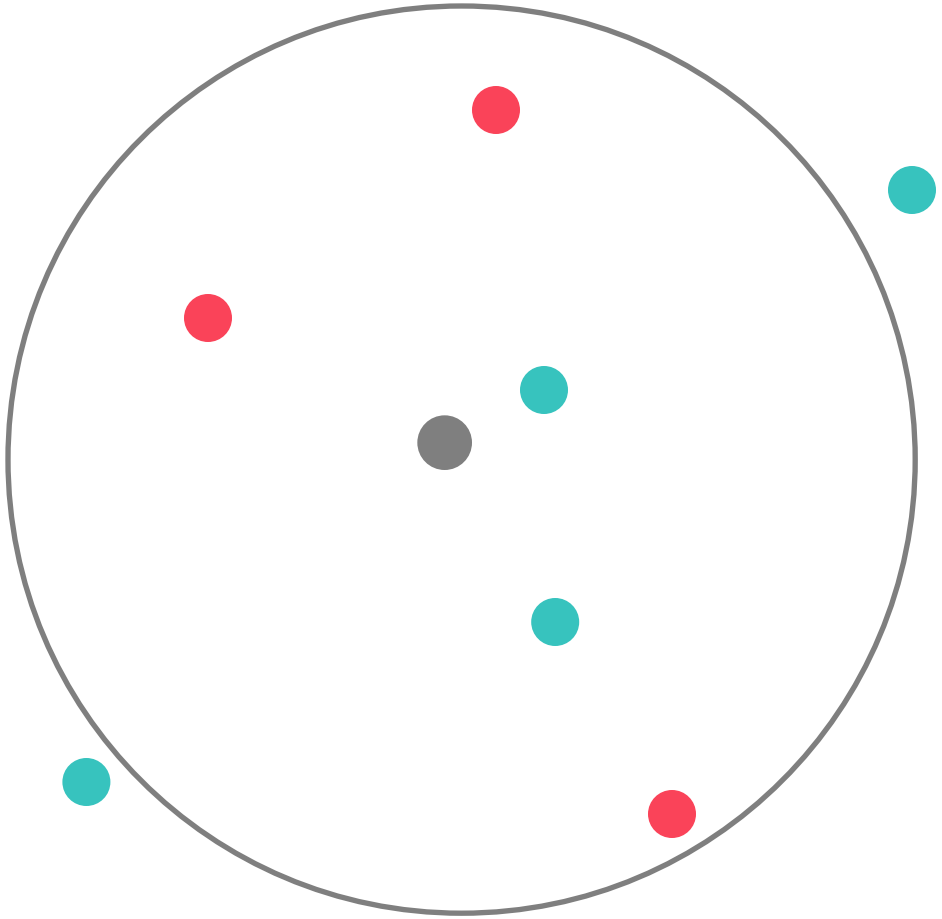
Green = 2

Red = 2

Prediction: It's a tie.

K-Nearest Neighbours

The KNN algorithm predicts the labels of the test dataset by looking at the labels of its closest neighbors in the feature space of the training dataset. The "K" is the most important hyperparameter that can be tuned to optimize the performance of the model.



K = 5

Green = 2

Red = 3

Prediction: **Red**

Pseudocode

```
# Inputs
#   data      ← training set of N examples (x, y)
#   k         ← number of neighbours
#   metric    ← distance function (e.g., Euclidean, Manhattan)
#   X_query   ← set of examples to classify

# ----- "fit" (lazy) -----
X_train ← data.x
y_train ← data.y

# ----- predict -----
ŷ ← list of length |X_query|          # outputs align 1:1 with X_query

FOR i = 1 TO |X_query| DO
    x* ← X_query[i]
    d  ← distances from x* to all X_train using metric
    J  ← indices of the k smallest values in d
    # majority vote = class with the highest count among y_train[J];
    # break ties by any fixed rule (e.g., alphabetical order of class name)
    ŷ[i] ← majority_vote(y_train[J])
END FOR

RETURN ŷ
```

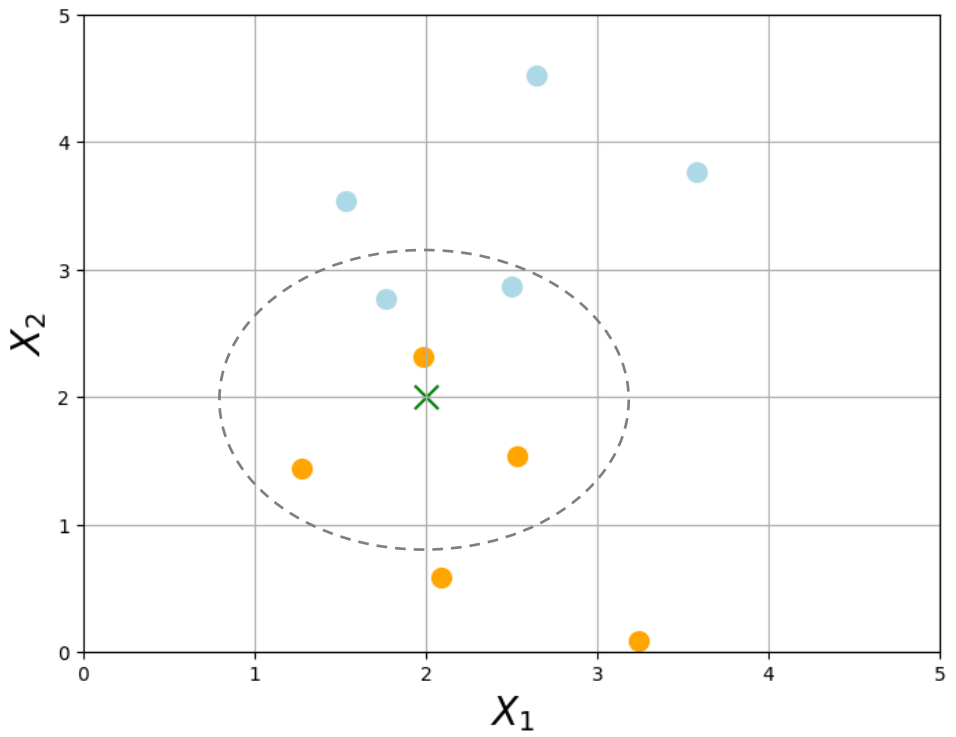
KNN from Scratch

```
import numpy as np
from sklearn.base import BaseEstimator, ClassifierMixin
from sklearn.metrics import pairwise_distances

class MyKNNClassifier(BaseEstimator, ClassifierMixin):
    def __init__(self, k=3, metric="Euclidean") :
        self.k = k
        self.metric = metric

    def fit(self, X, y) :
        self.X_train_ = np.asarray(X)
        self.y_train_ = np.asarray(y)
        return self

    def predict(self, X) :
        X = np.asarray(X)
        y_pred = np.empty(X.shape[0], dtype=self.y_train_.dtype)

        for i, x in enumerate(X):
            d = pairwise_distances(self.X_train_, x[None, :], metric=self.metric).ravel()
            J = np.argsort(d)[:self.k]
            labels, counts = np.unique(self.y_train_[J], return_counts=True)
            y_pred[i] = labels[np.argmax(counts)]  # majority vote; ties break by smallest label

        return y_pred
```

Usage Example

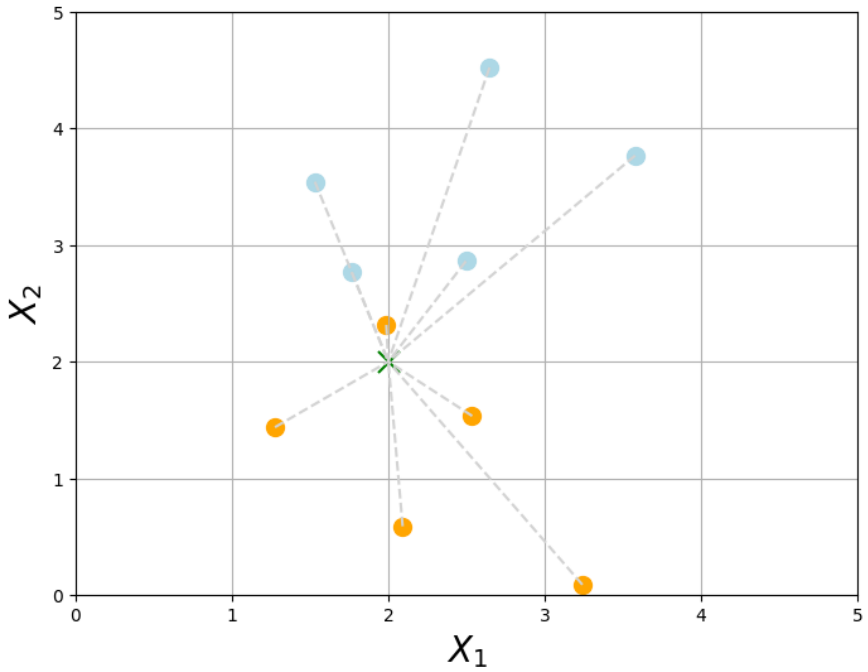
```
# Test data point
X_test = np.array([[2,2]])

# Example usage
K = 5
distance_metric = “euclidean” # Can be “euclidean”, “Manhattan”, “cosine”, etc.

knn = MyKNNClassifier(K=5, metric=distance_metric)
knn.fit(X_train, y_train)

Y_pred = knn.predict()
print("Predicted Class:", y_pred)
```

```
Predicted Class: [1.0]
```



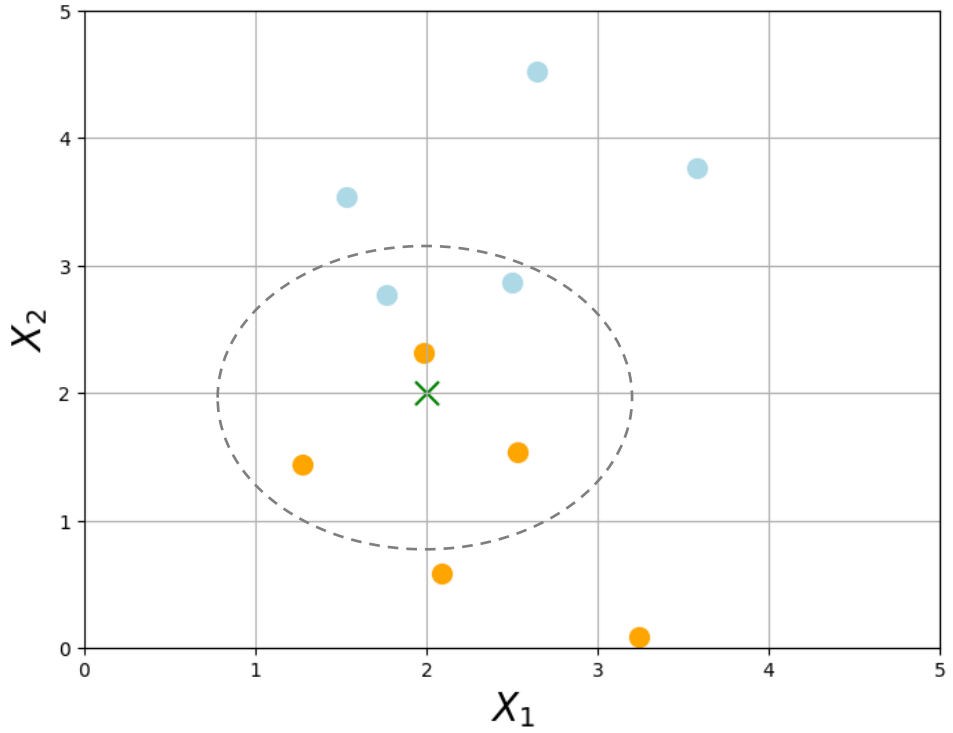
Step 1: Pairwise Distance Calculation

```
# Calculate distances using pairwise_distances
d = pairwise_distances(self.X_train_, x[None, :], metric=self.metric).ravel()
```



Step 2 & 3: Finding k Nearest Neighbours

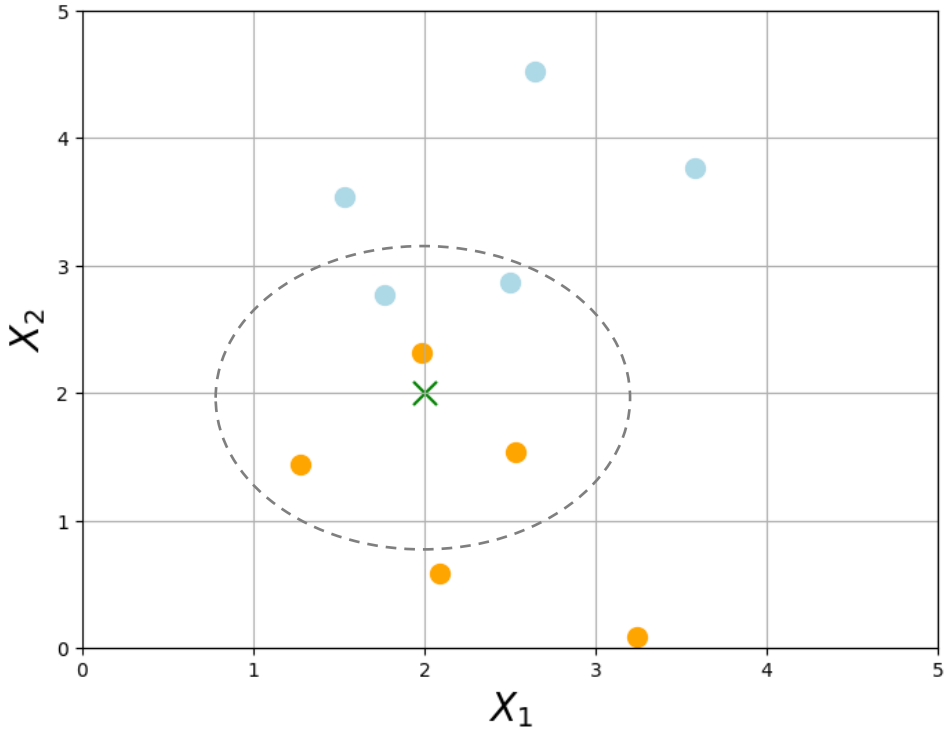
```
# Sort distances and select K nearest neighbors
J = np.argsort(d)[:self.k]
```



K = 5

Step 4 & 5: Prediction based on Majority Voting

```
# Get the labels of the K nearest neighbors
labels, counts = np.unique(self.y_train_[J], return_counts=True)

# Find the most common label among the K neighbors
y_pred[i] = labels[np.argmax(counts)]  # majority vote; ties break by smallest label
```



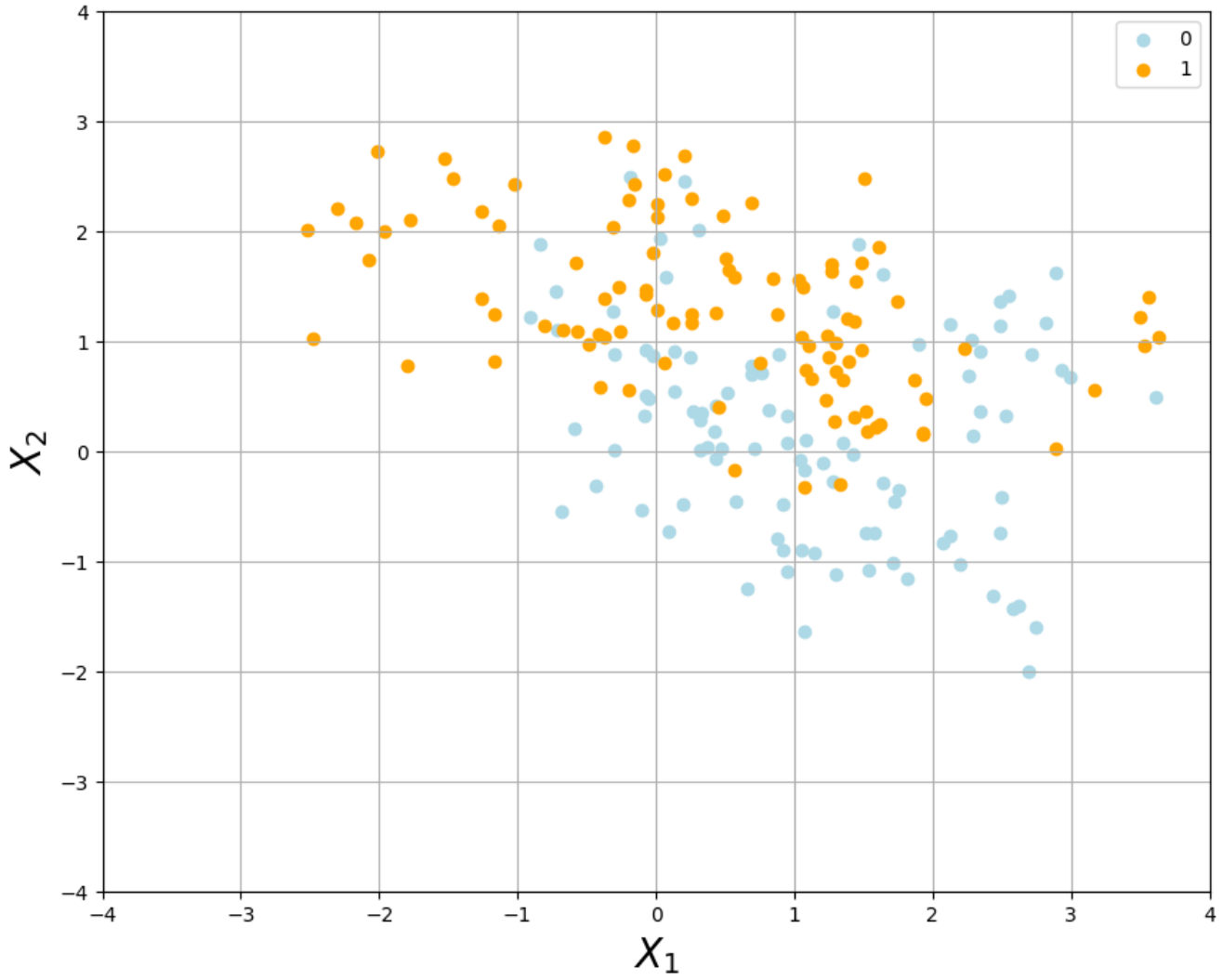
K = 5

# of 0 = 2

# of 1 = 3

Prediction: 1

R Mixture Example Dataset



ElemStatLearn Package:

- Simulated Mixture Guassian Dataset
- 2 Features, 2 Classes

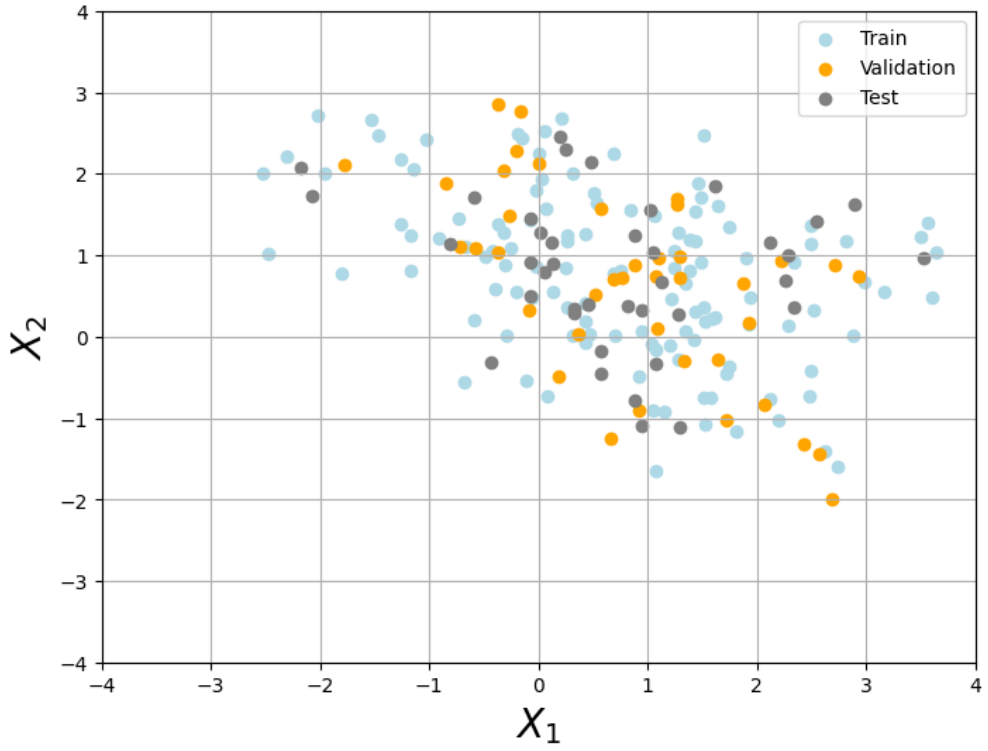
What Makes a Good ML Model?

- We learn through tutorials and practice problems. Hopefully, we develop understanding of the subject.
- To pass an exam, we are likely to be tested on unseen exam questions, so memorising the practice ones will not work.

- Like wise, a good classifer is the one that consistently performs well on both trainning data and also an unseen dataset.
- The model must learn underlying relationships in the data, instead of memorising it.

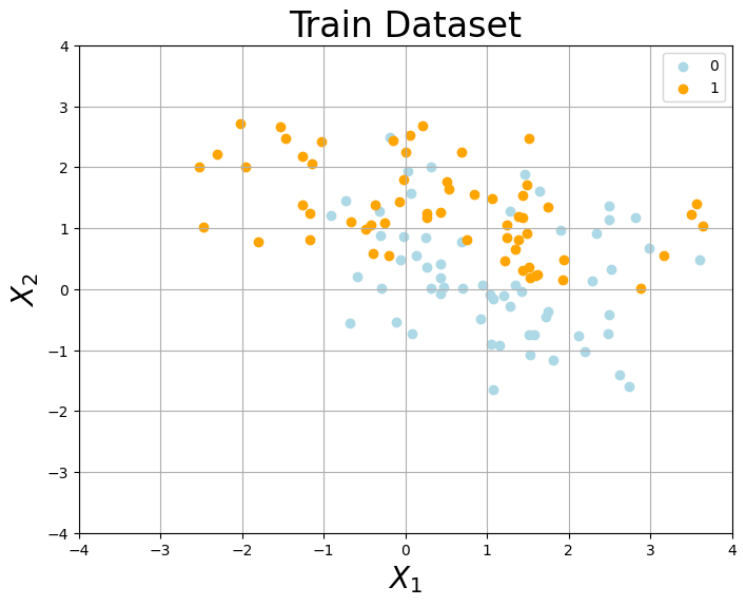
Train, Validation and Test Datasets

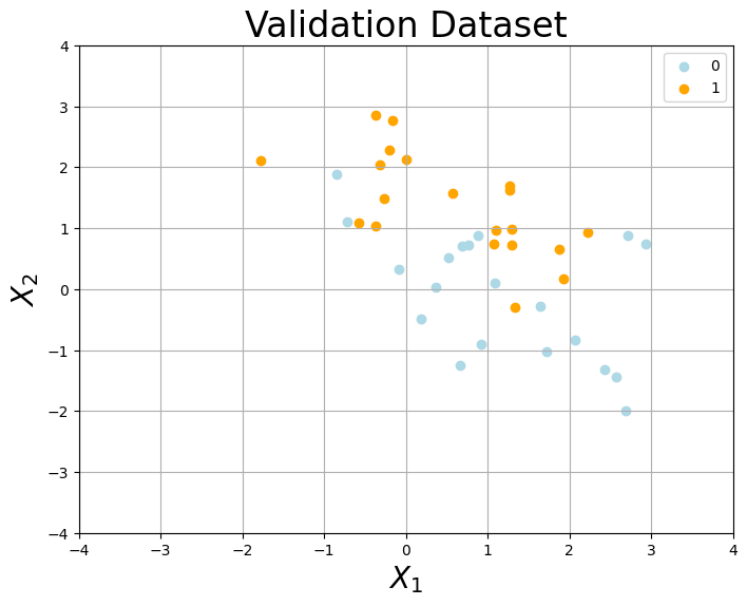
```
from sklearn.model_selection import train_test_split

# First, split the data into train (60%) and temp (40%)
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.4, random_state=42, stratify=y)

# Then split the temp data into validation (50% of 40% ⟹ 20% of total) and test (remaining 50% of 40% ⟹ 20% of total)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=0.5, random_state=42, stratify=y_temp)
```



- (Train+Validation):Test is typicaly either 0.8:0.2 or 0.7:0.3.
- **Test** dataset is a proxy of unseen data, and it will only be used in the final evaluation.
- We train our ML model on the **Train** dataset.
- **Validation** datest is used to fine-tune or optimise the ML model. Here, we find a set of hyper-parameters, e.g. k in the kNN context, that will result in the best performance on the Validation dataset.

Train, Validation and Test Datasets (cont.)



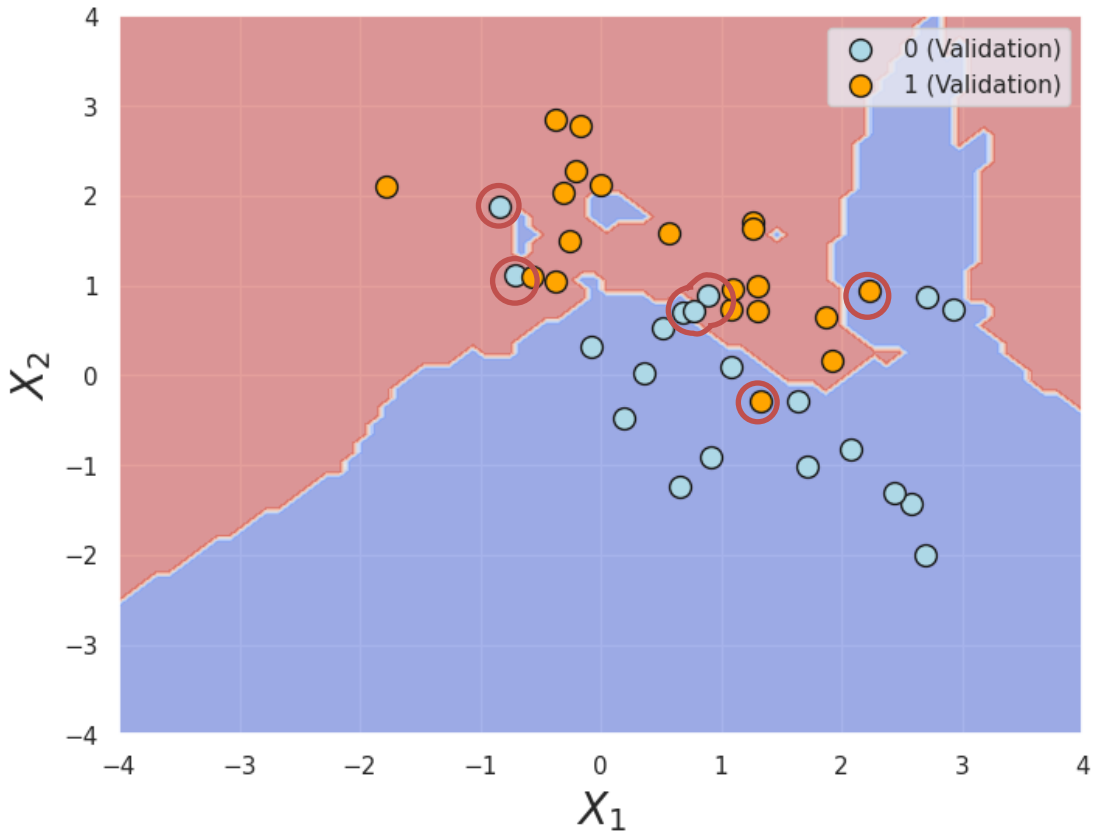


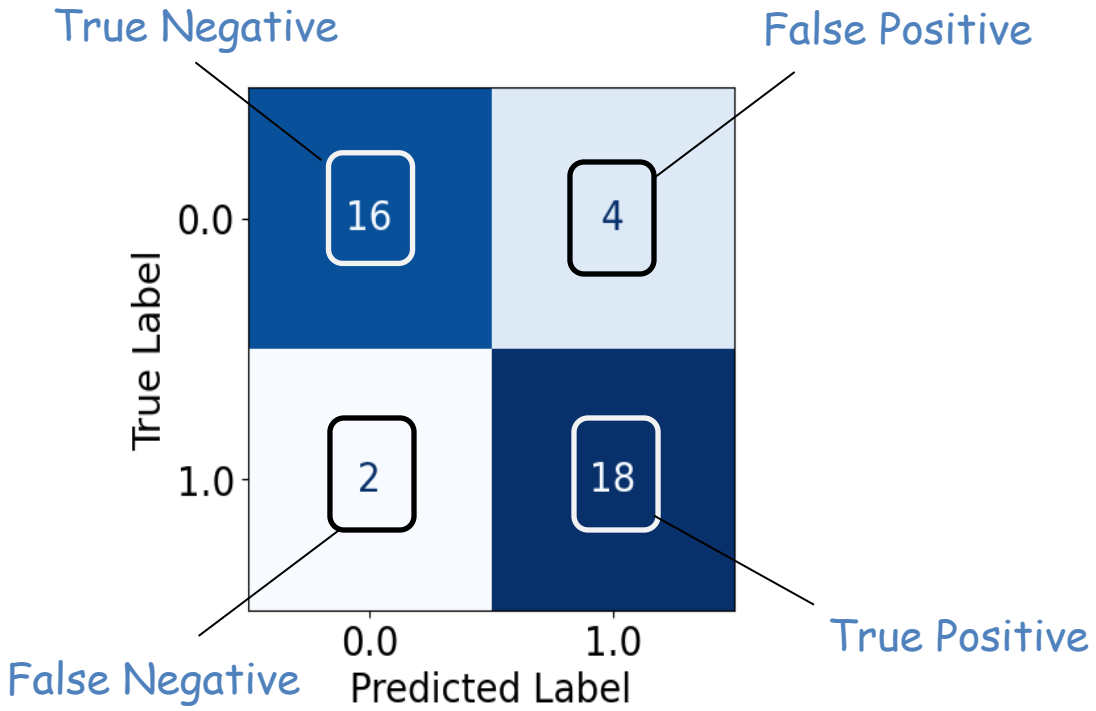
scikit-learn: KNeighborsClassifier

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

# Define k and the distance metric
k = 5
distance_metric = “Euclidean”

# Create an instance of KNeighborsClassifier
knn = KNeighborsClassifier(n_neighbors=k, metric=distance_metric)

# Fit the model on the training data
knn.fit(X_train, y_train)

# Predict the labels on the validation set
predictions = knn.predict(X_val)

# Calculate the accuracy of the predictions
accuracy = accuracy_score(y_val, predictions)
print("Accuracy of the k-NN classifier on the validation set:", accuracy)
```

```
Accuracy of the KNN classifier on the validation set: 0.85
```

Decision Boundary





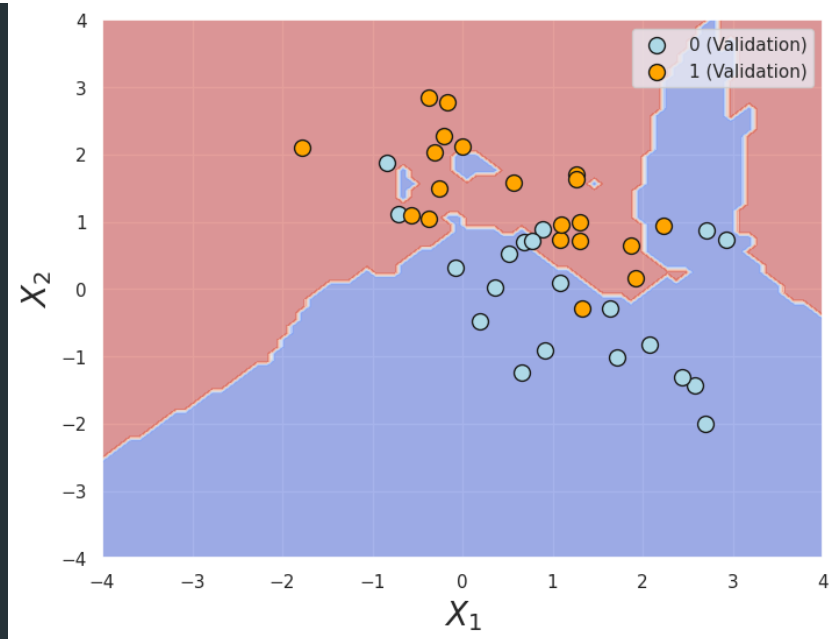
$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

$$= \frac{16 + 18}{16 + 4 + 2 + 18}$$

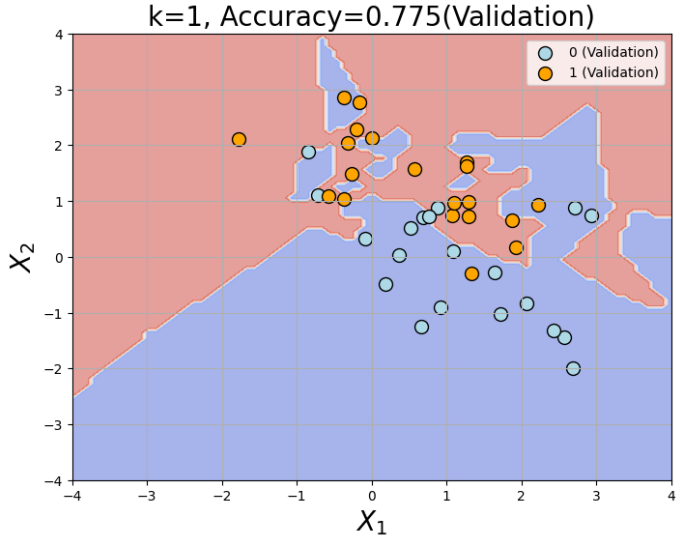
$$= 0.85$$

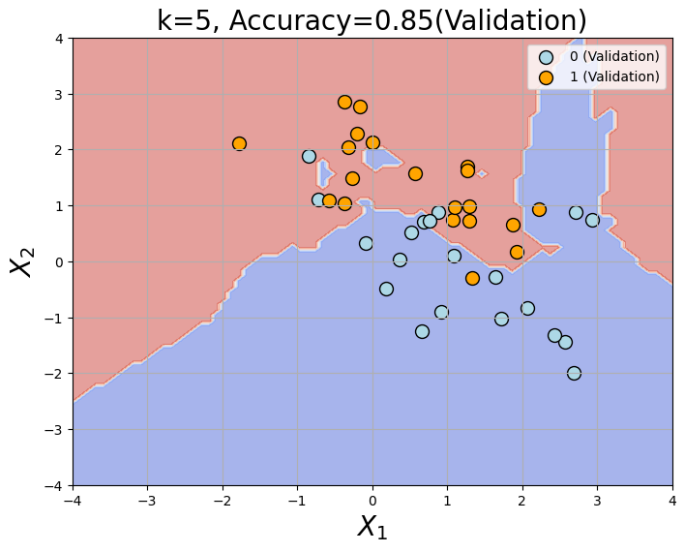
Decision Boundary

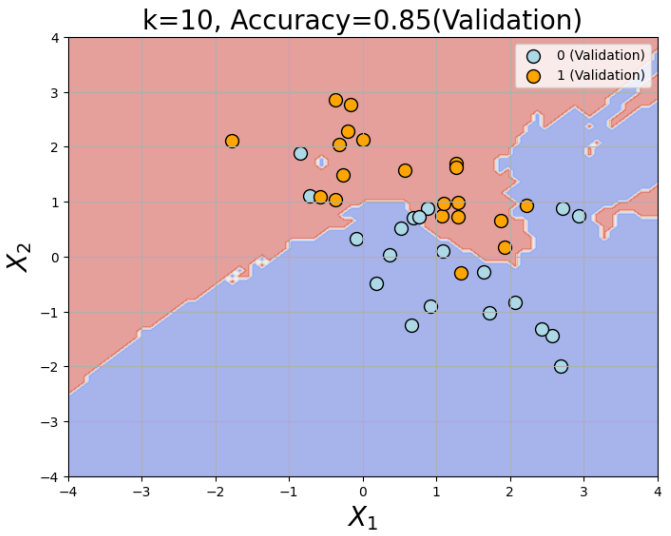
```
from sklearn.inspection import DecisionBoundaryDisplay

# Train the classifier
knn = MyKNNClassifier(k=5, distance_metric=”euclidean”)
knn.fit(X_train, y_train)

# Plotting decision boundaries
fig, ax = plt.subplots(figsize=(8, 6))
db_display = DecisionBoundaryDisplay.from_estimator(
    estimator=knn, X=X_train, response_method="predict",
    alpha=0.5, ax=ax, cmap='coolwarm'
)

# Scatter plot of the validation data with class labels
class_names = ['0', '1']
colors = ['lightblue', 'orange']  # Define colors for different classes
for i, color in enumerate(colors):
    idx = np.where(y_val == i)
    ax.scatter(X_val[idx, 0], X_val[idx, 1], c=color, edgecolor='k', s=100, label=f'{class_names[i]}
(Validation)')

# Adding labels, title and custom legend for validation data
ax.set_xlabel('$X_1$', fontsize=20)
ax.set_ylabel('$X_2$', fontsize=20)
ax.legend(loc='best')  # Position the legend to not overlap with data
ax.grid()
plt.axis([-4, 4, -4, 4])
plt.show()
```

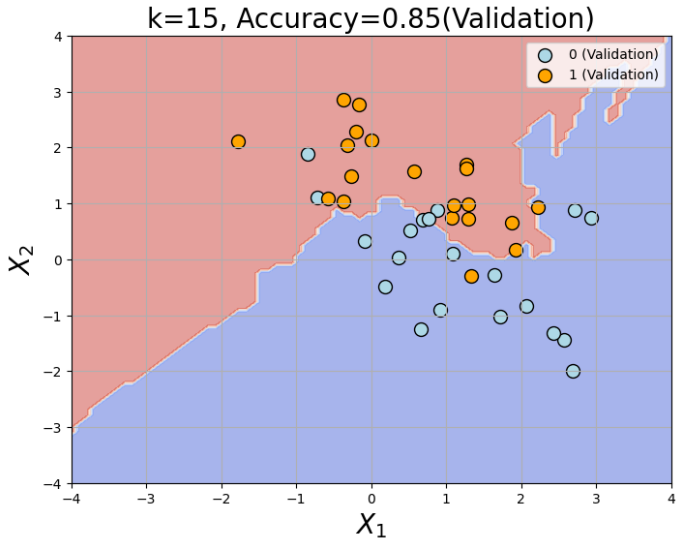


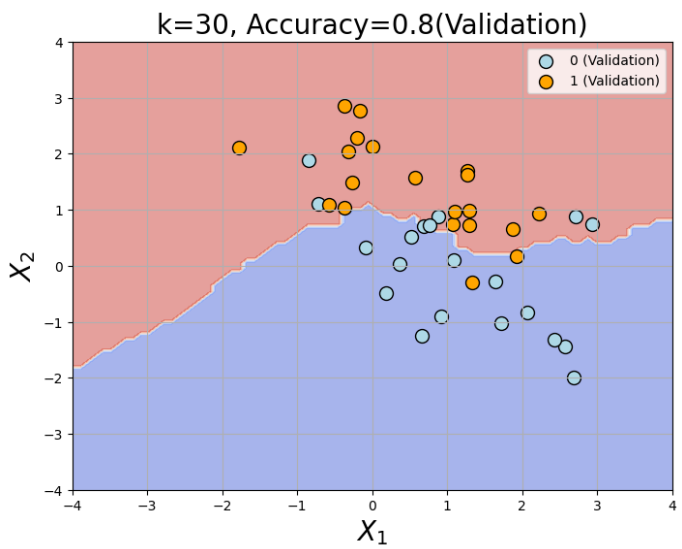
Model Complexity

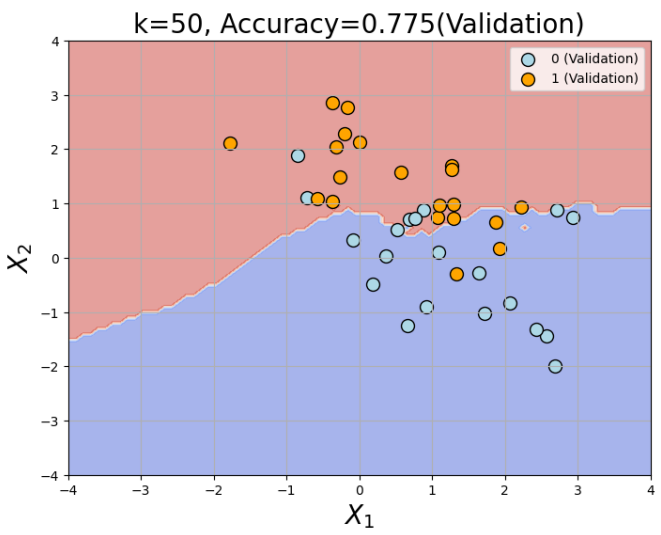




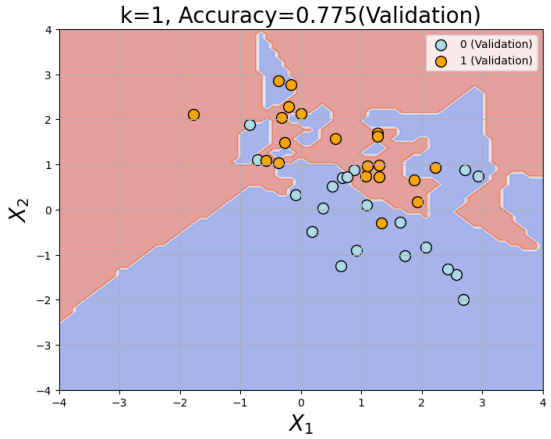




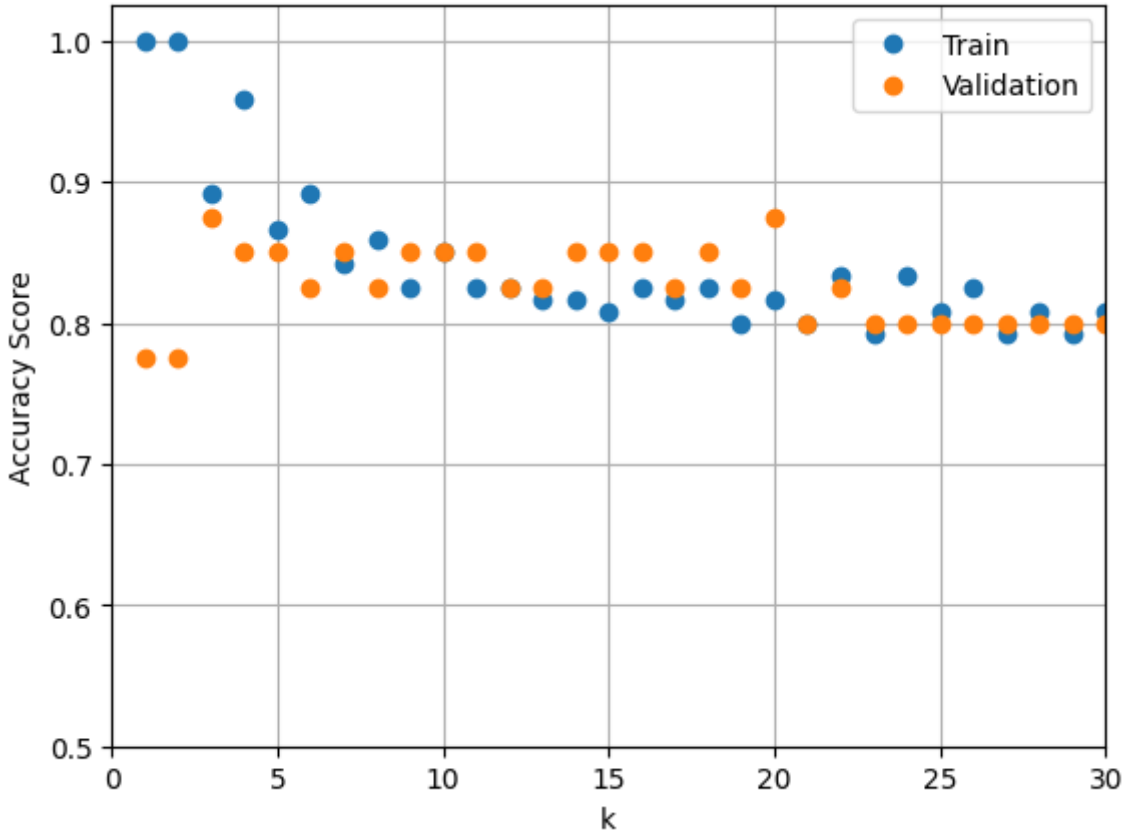


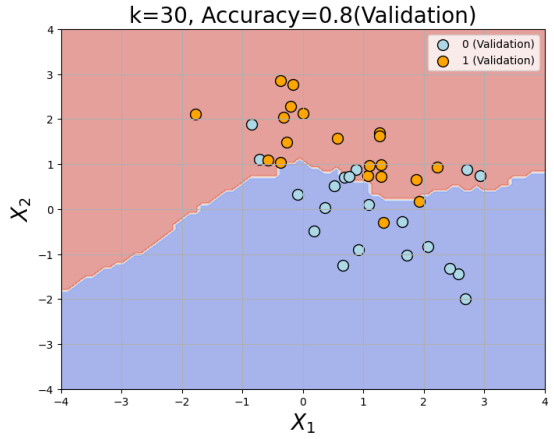


Bias-Variance Trade-Offs



- High Variance, Low Bias
- Complex Decision Boundary
- High Train Accuracy
- Low Validation Accuracy

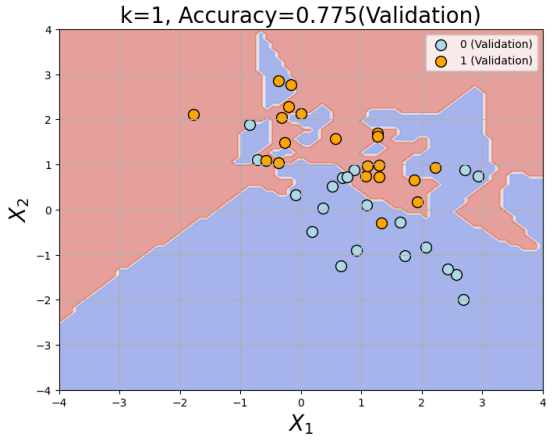




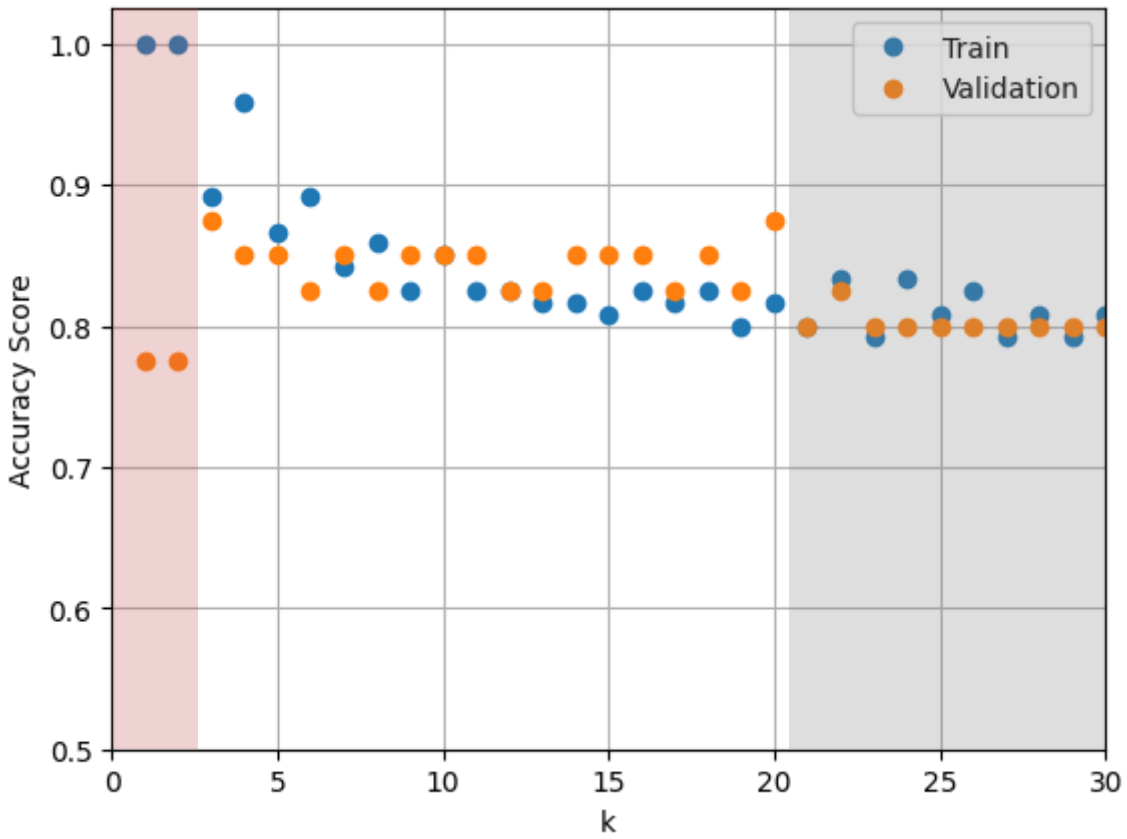
- High Bias, Low Variance
- Simple Decision Boundary
- Low Train Accuracy
- Low Validation Accuracy

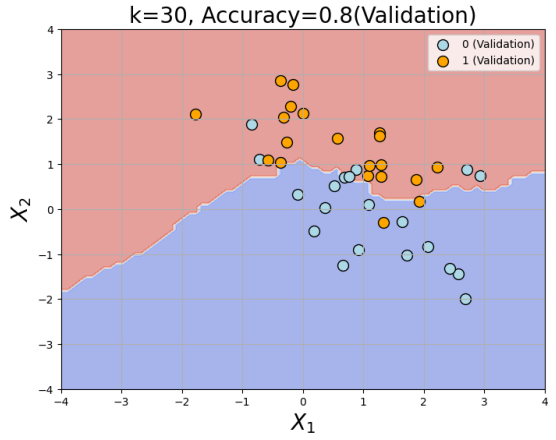
Overfitting vs Underfitting

- **Overfitting** happens when a small k leads to high training accuracy but poor validation accuracy due to overly complex decision boundaries that fail to generalise.
- **Underfitting** occurs when a large k leads to overly simple decision boundaries, resulting in low training and validation accuracies.



- High Variance, Low Bias
- Complex Decision Boundary
- High Train Accuracy
- Low Validation Accuracy

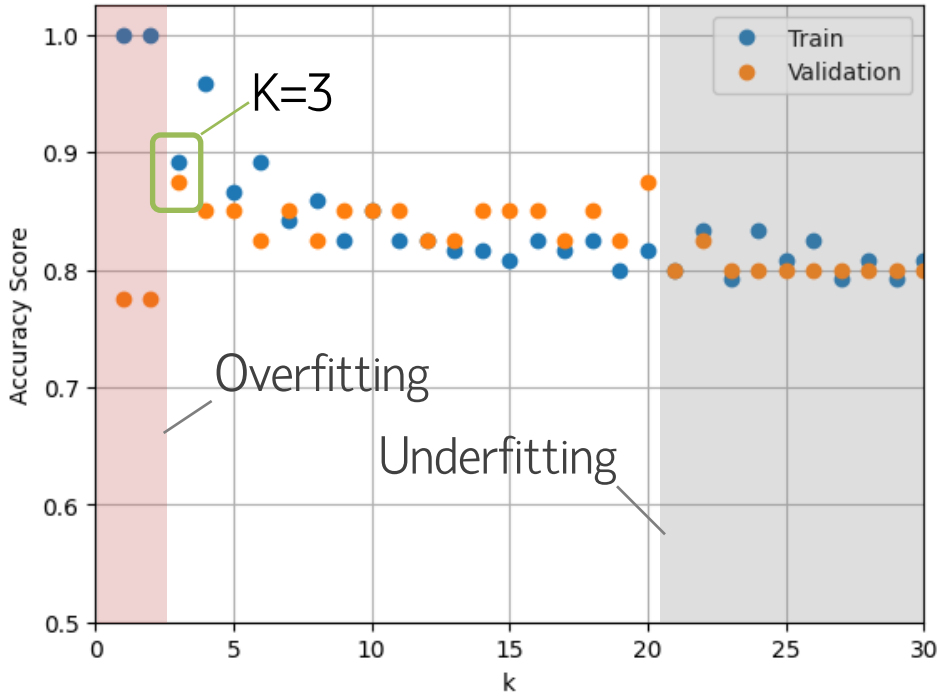


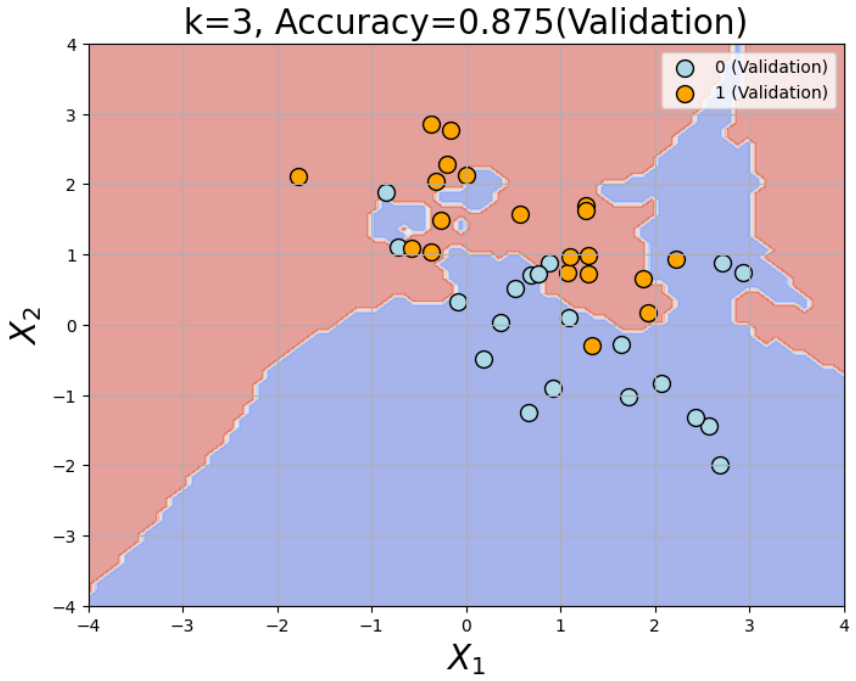


- High Bias, Low Variance
- Simple Decision Boundary
- Low Train Accuracy
- Low Validation Accuracy

Optimal Model

- An optimal KNN model is one where the choice of K balances the complexity of decision boundaries to achieve high validation accuracy while maintaining good training accuracy, indicating effective generalisation without overfitting or underfitting.





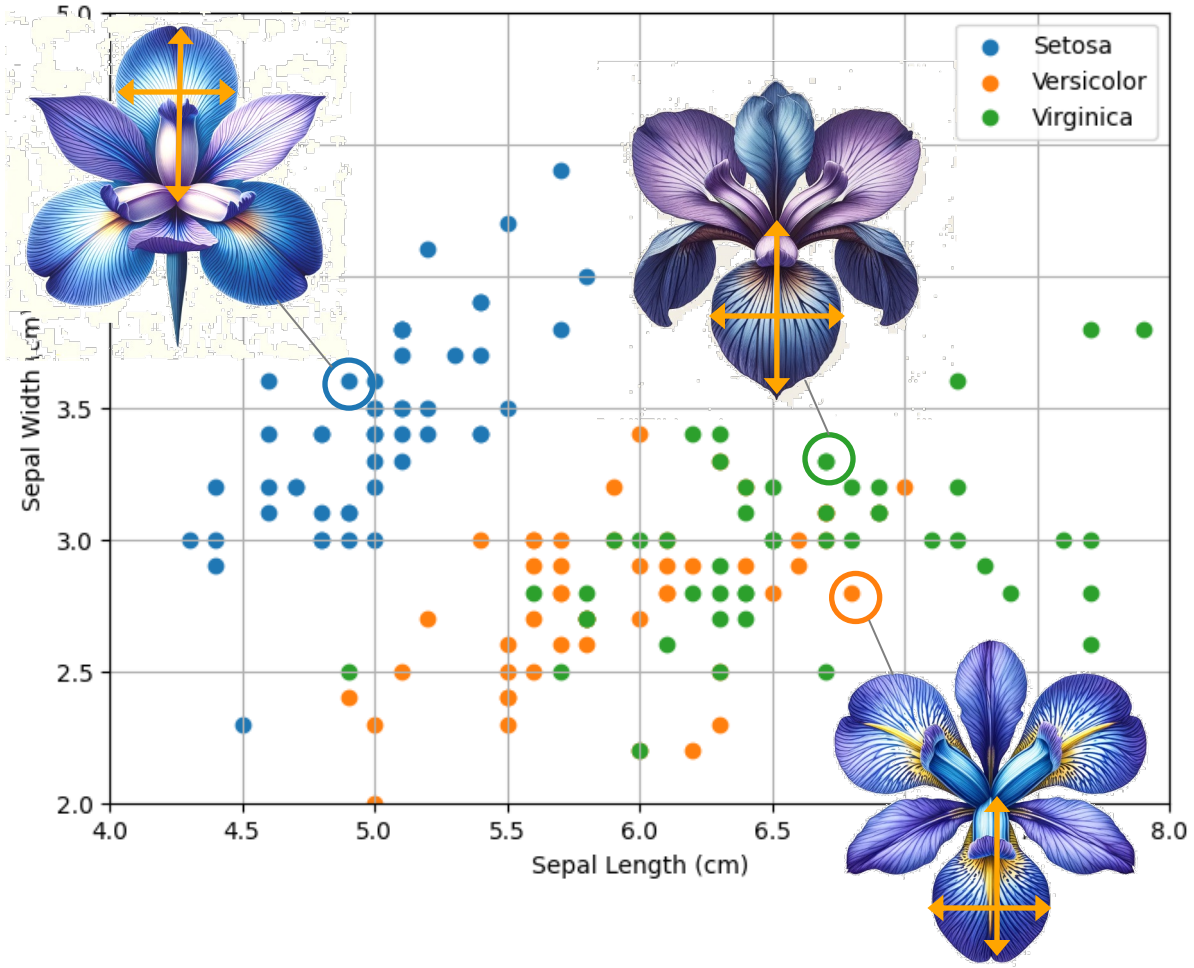

```
Accuracy of the KNN classifier on the test set: 0.75
```

>10% drop is BAD. Our validation and test sets are not from the same population. This happened when the dataset is small. K-fold cross-validation addresses this shortcoming, more of it in our Logistic Regression lecture.

Until then, prefer a slightly higher-bias model (larger k) to reduce variance; with k=5 the test accuracy is 85%.

What is Nearest?

- For KNN to function, we must define the notation of 'Nearest'.
- A: Distance (or Similarity) Measure



Versicolor ⇔ Virginica

- Less Distance implies Near (High Similarity).

Setosa ⇔ Virginica

- Further Distance implies Far (Low Similarity).

What is Nearest? (cont.)

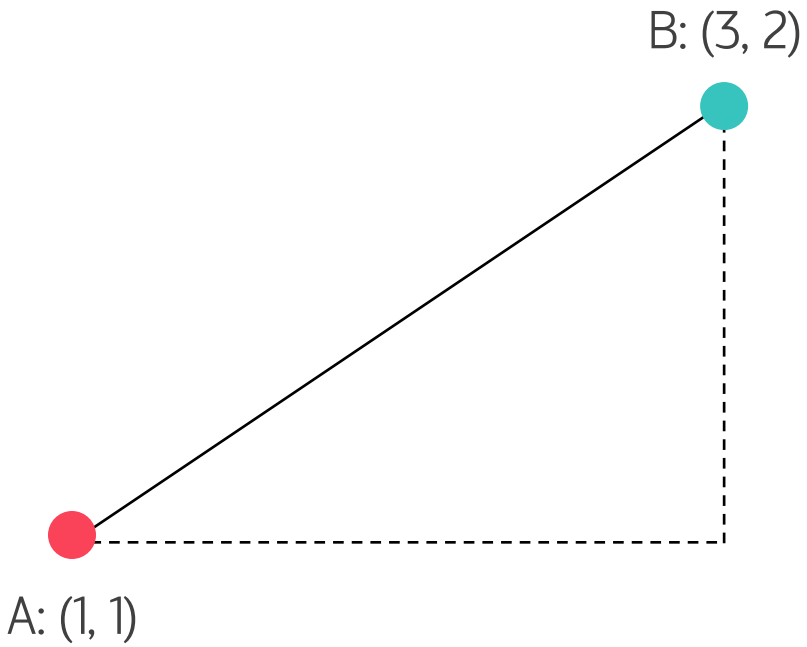
- For KNN to function, we must define the notation of 'Nearest'.
- A: Distance (or Similarity) Measure

High Similarity implies Near (or Less Distance).

Low Similarity implies Far (or Further Distance).

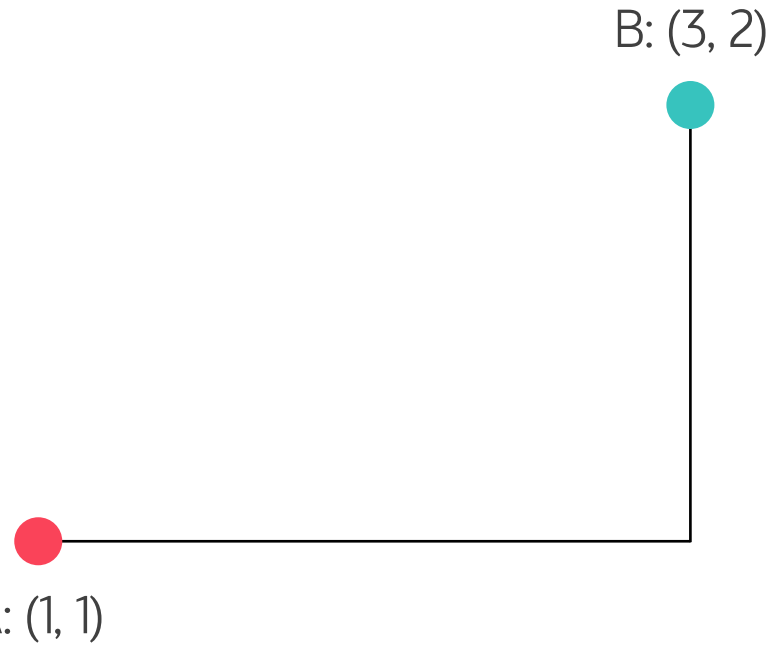
Distance Metrics

L_2 Norm (Euclidean Distance)



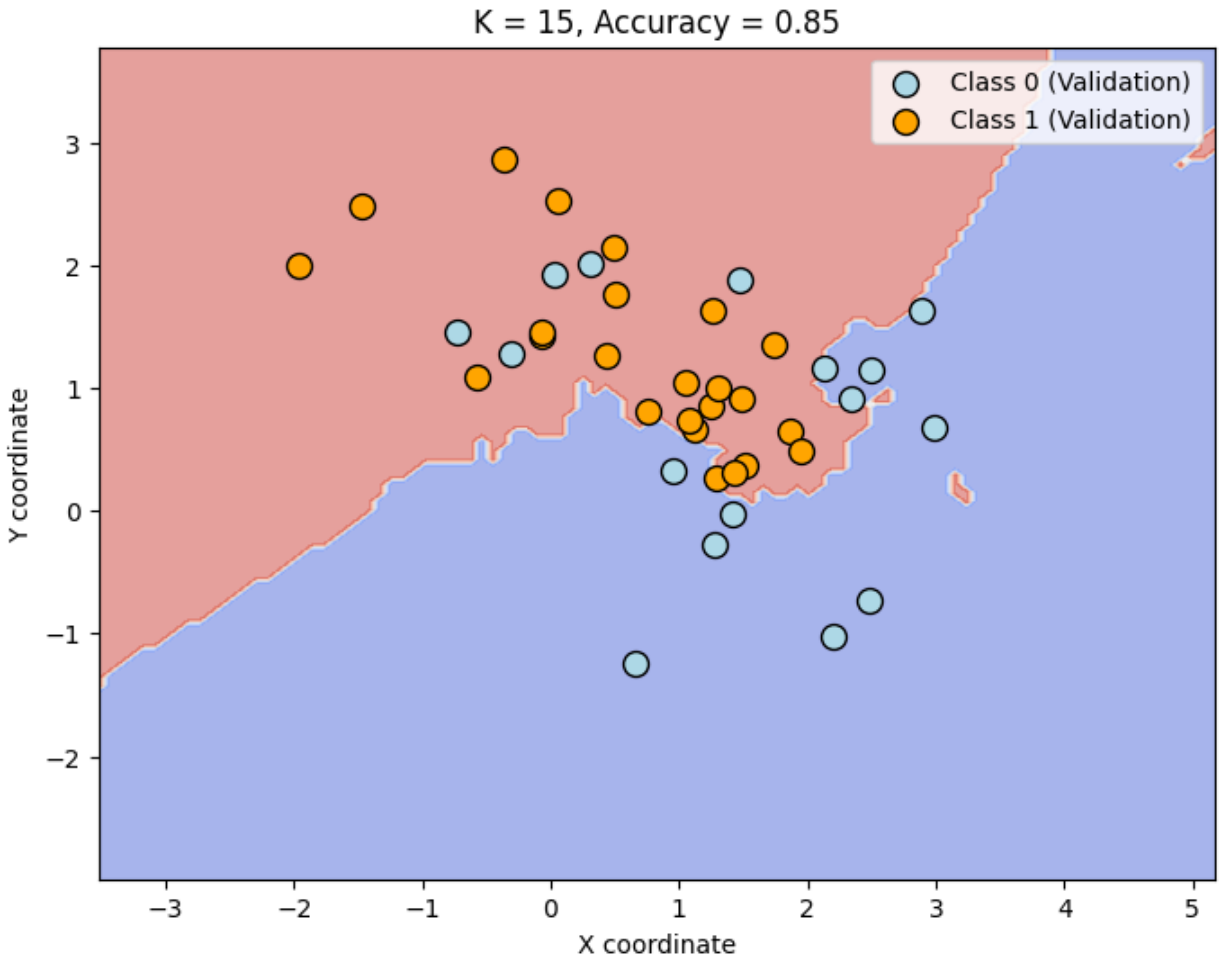
$$\|A-B\|_2 = \sqrt{(x_1^{(B)}-x_1^{(A)})^2+(x_2^{(B)}-x_2^{(A)})^2}$$
$$= \sqrt{(3-1)^2+(2-1)^2}$$
$$= \sqrt{2^2+1^2} = \sqrt{5}\,(\sim 2.24)$$

L_1 Norm (Manhattan or Taxicab Distance)

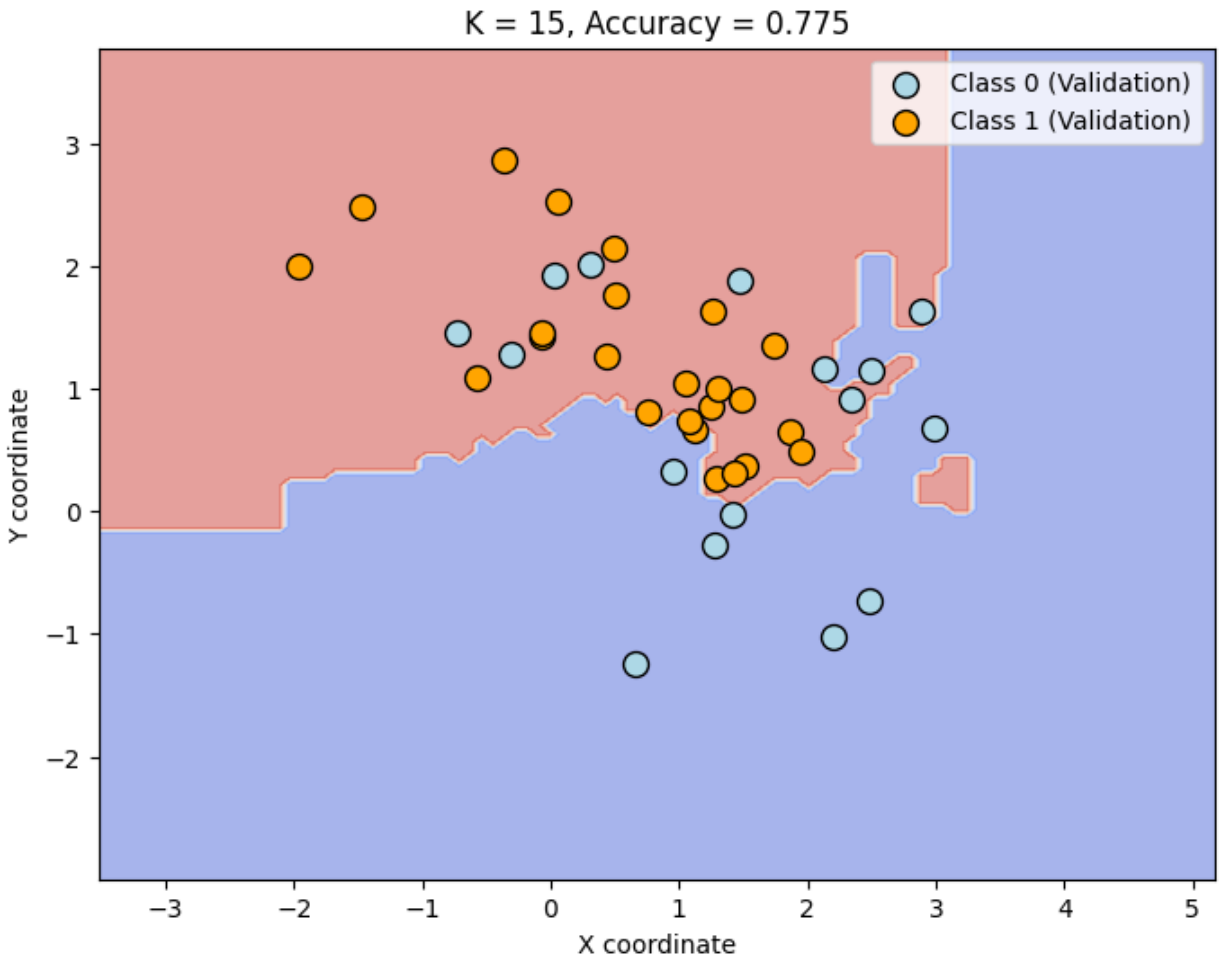


$$\|A-B\|_1 = \left|x_1^{(B)}-x_1^{(A)}\right| + \left|x_2^{(B)}-x_2^{(A)}\right|$$
$$= |3-1|+|2-1|$$
$$= 2+1 = 3$$

Decision Boundary



Euclean Distance



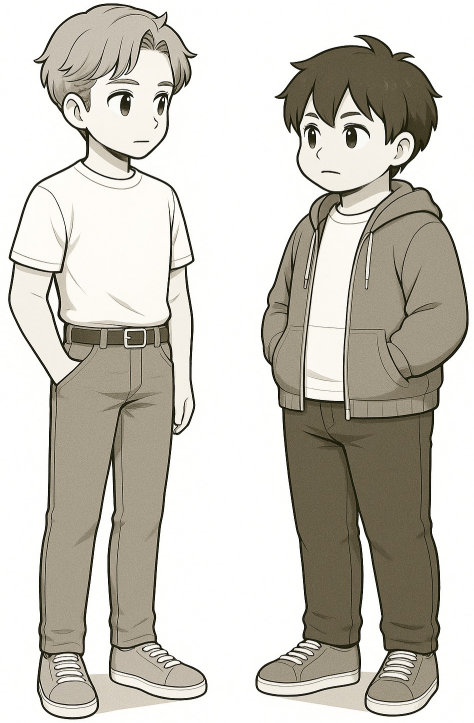
Manhattan Distance

Optimal Model (Revised)

- An optimal KNN model is one where the choice of **K** and **distance metric** balances the complexity of decision boundaries to achieve high validation accuracy while maintaining good training accuracy, indicating effective generalisation without overfitting or underfitting.

K	Distance	Train Accuracy	Validation Accuracy
1	Manhattan	1	0.775
1	Euclidean	1	0.775
2	Manhattan	1	0.775
2	Euclidean	1	0.775
3	Manhattan	0.892	0.85
3	Euclidean	0.892	0.875
4	Manhattan	0.95	0.85
4	Euclidean	0.958	0.85
5	Manhattan	0.875	0.85
5	Euclidean	0.867	0.85
...	...	...	...

Scaling Effects

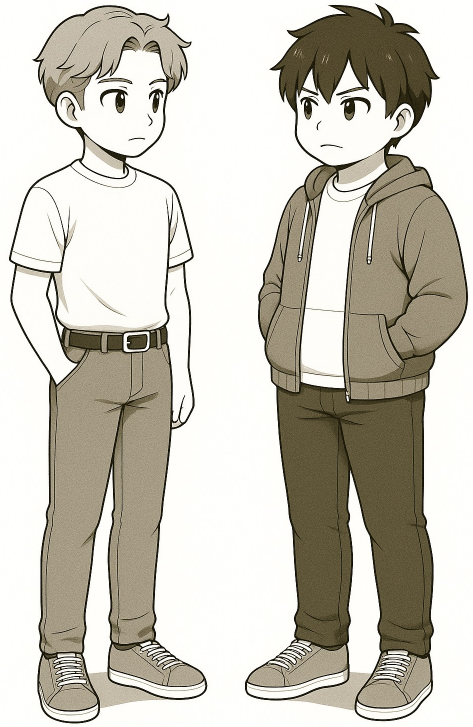


A: (170 cm, 60 kg), B: (160 cm, 60 kg)

Distance(A, B) = 10

C: (170 cm, 60 kg), D: (170 cm, 70 kg)

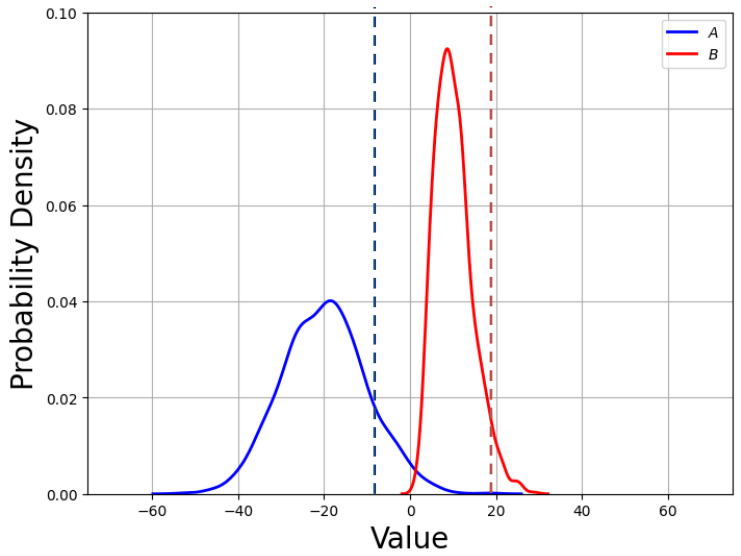
Distance(C, D) = 10



- Distance(A, B) = Distance(C, D). Eh?
- We are able to differentiate A form B quite easilty. This will not be so obvious between C and D.
- Where possible, we must ensure different features are fairly contributing to the machine learning (ML) model.

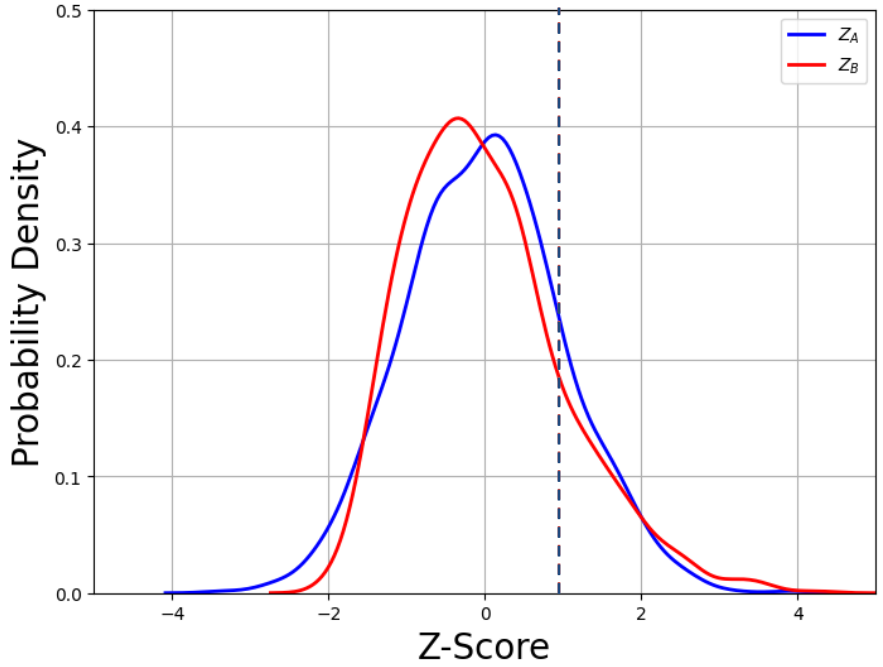
Standardising with Z Scores

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



$$\mu_A = -19.81$$
$$\sigma_A = 9.88$$

$$\mu_B = 10.16$$
$$\sigma_B = 4.40$$



$$Z_A = \frac{-9.93 - (-19.81)}{9.88}$$
$$= 1$$

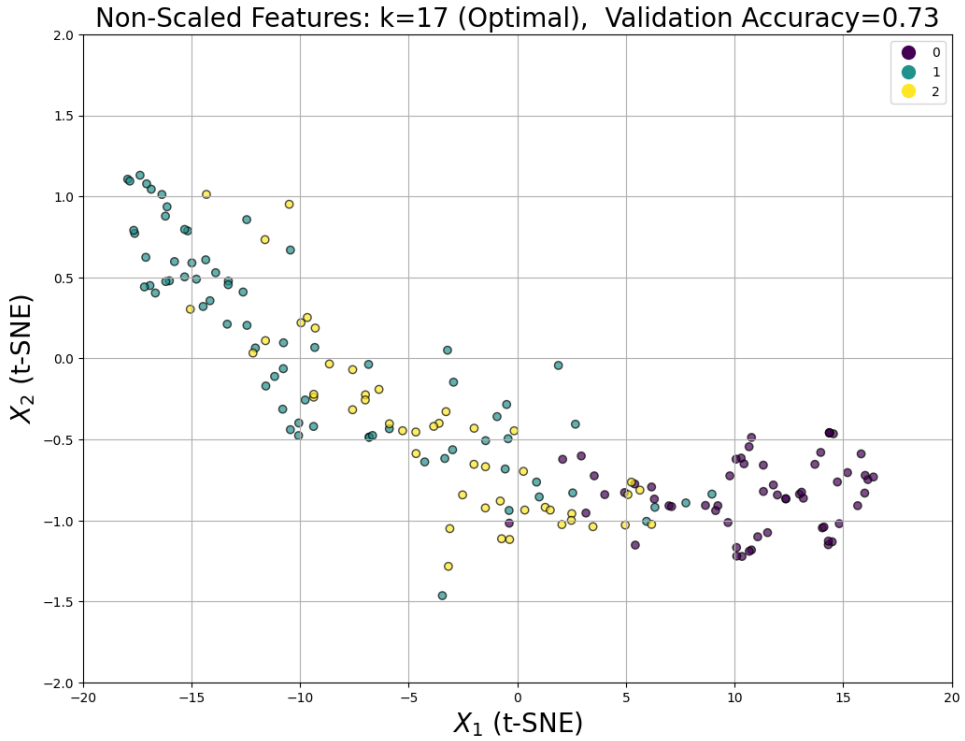
$$Z_B = \frac{14.56 - 10.16}{4.40}$$
$$= 1$$

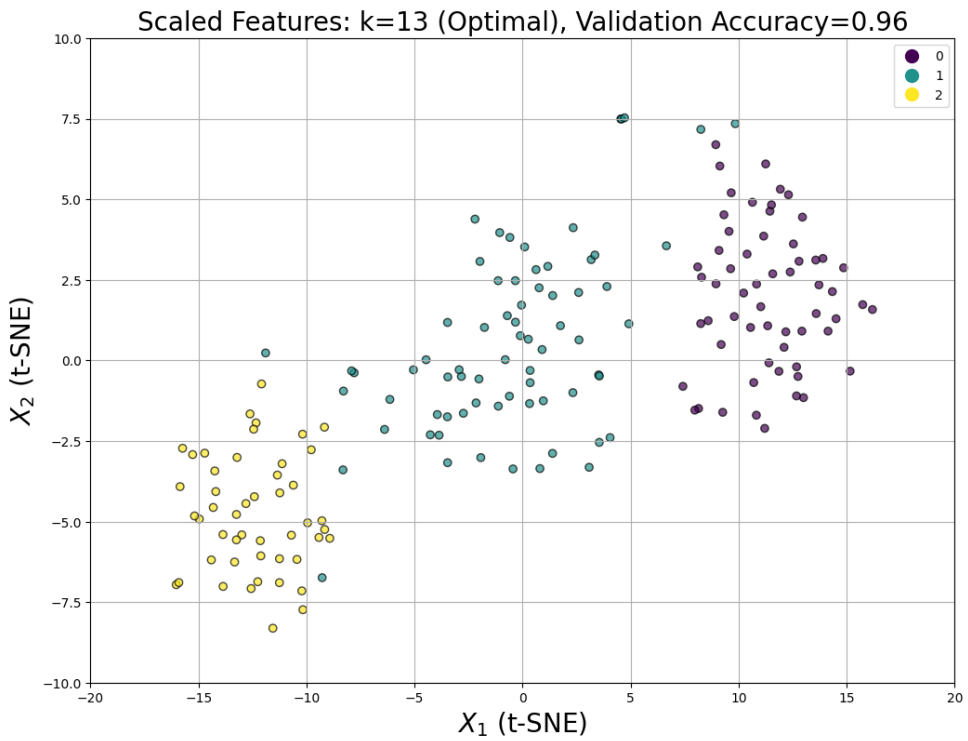
scikit-learn: StandardScaler

```
from sklearn.datasets import load_wine
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
import numpy as np

# Load the dataset
data = load_wine()
X = data.data
y = data.target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42, stratify=y)

# Create a StandardScaler object
scaler = StandardScaler()

# Fit the scaler to the training data only
scaler.fit(X_train)

# Transform both the training data and testing data
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

- Fit the scaler only on the training data to avoid data leakage. We then use this fitted scaler to transform both the training and test datasets. This ensures that the model is tested on truly unseen data, maintaining the integrity of the test set and enabling an accurate evaluation of model performance.

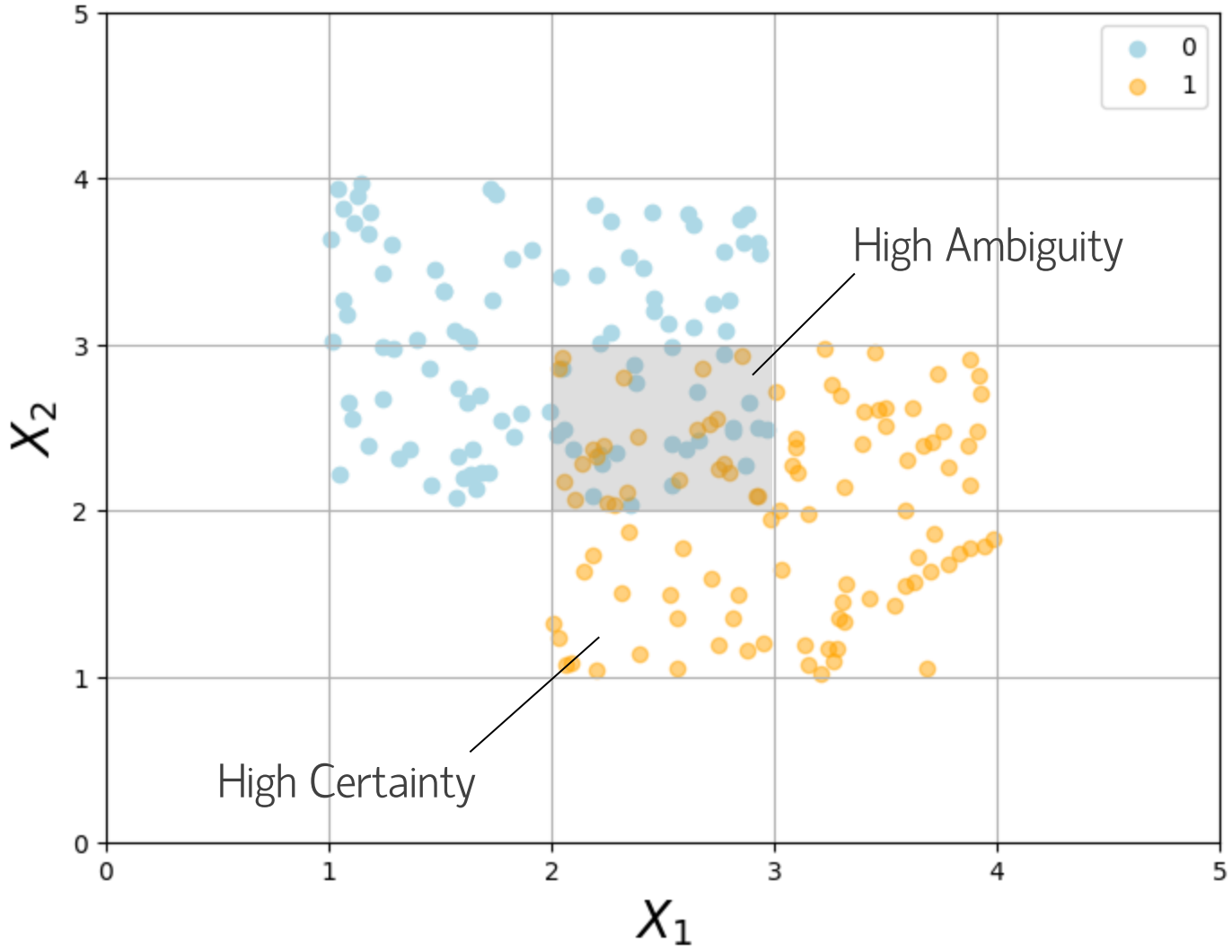
Wine Dataset: Better Prediction Performance with Z-Score Features





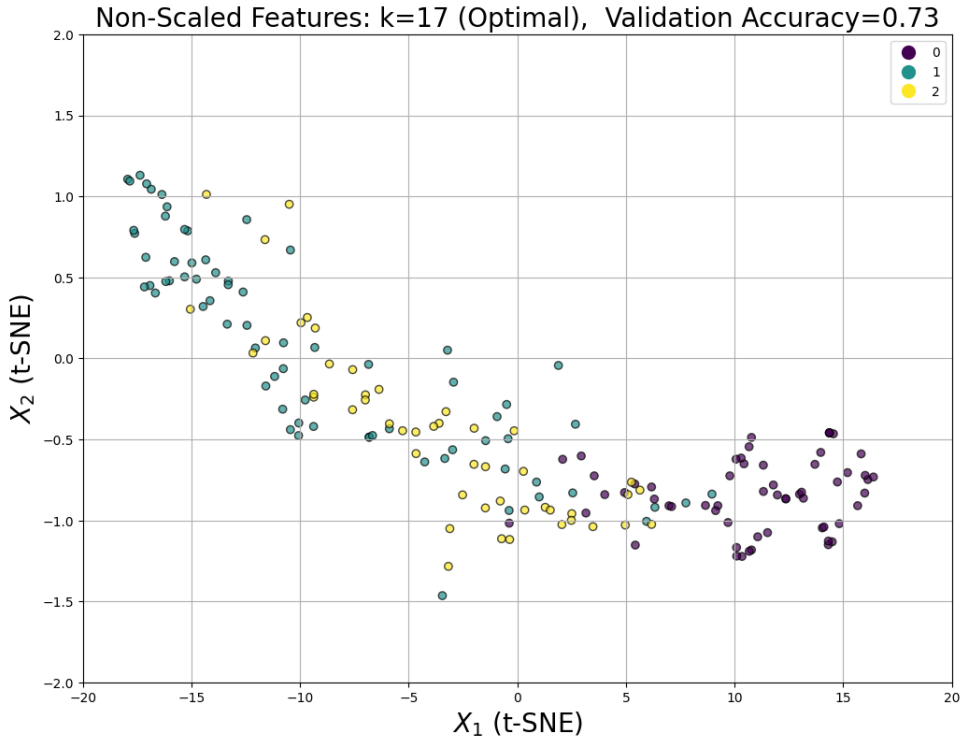
- Wine Dataset: 13 Features. These features are chemical constituents found in wine, specifically from three different cultivars of wine grown in the same region in Italy.
- Standardisation ensures features with different units and scales are contributing fairly into the ML model. This results in better separation between the classes.

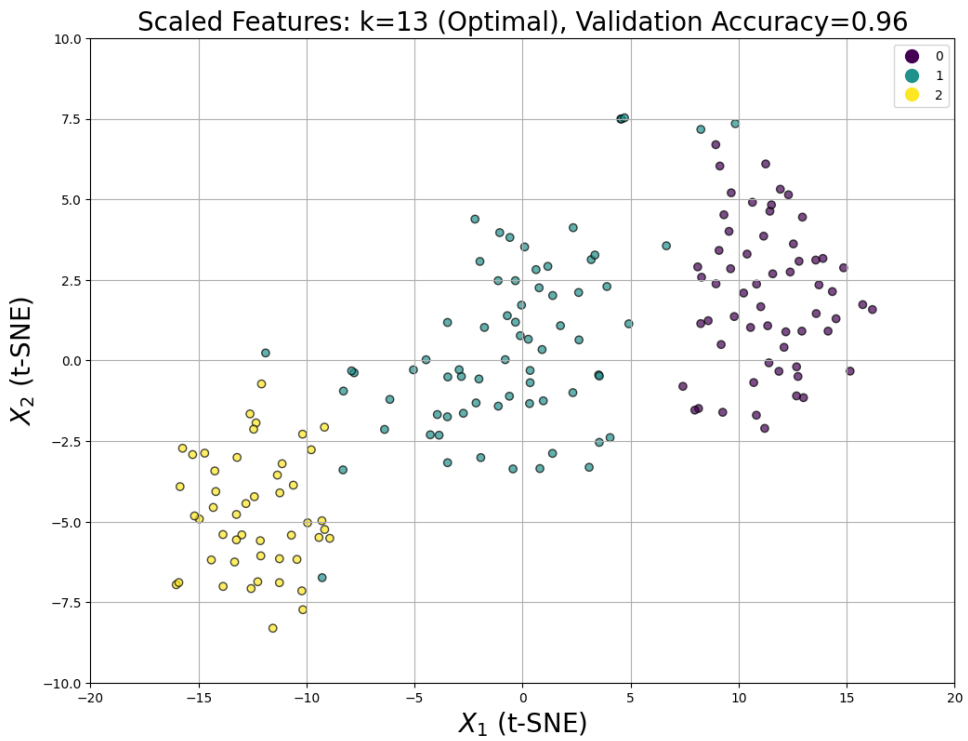
Where Do Errors Come From?



- The area, where the classes are overlapping. is where we are likely to make incorrect predictions. Hence, this is where classification errors are coming from.
- On the contrary, we can predict with high accuracy in the area where this is no (or little) overlap between the classes.

Wine Dataset: Z-Score Features (Revisit)





- When features are not scaled, there are large overlapping areas between the classes. We will make a lot of mistakes, and hence low validation accuracy.
- With standardisation, the classes are well separated. The overlapping areas are small, and hence we can predict with high certainty. This results in a high validation accuracy when we standardise the features.

scikit-learn: t-SNE

```
from sklearn.manifold import TSNE

# t-SNE on the original data
tsne = TSNE(n_components=2, random_state=42)
X_tsne = tsne.fit_transform(X)

# Mapping colors to labels
cmap = plt.cm.viridis
norm = plt.Normalize(y.min(), y.max())

# Figure: t-SNE on Original Data
plt.figure(figsize=(12, 9))
scatter = plt.scatter(X_tsne[:, 0], X_tsne[:, 1], c=y, cmap=cmap, edgecolor='k', alpha=0.7, norm=norm)
plt.xlabel('$X_1$ (t-SNE)', fontsize=20)
plt.ylabel('$X_2$ (t-SNE)', fontsize=20)
plt.axis([-20, 20, -2, 2])
plt.grid()

# Create a legend:
handles = [plt.Line2D([], [], marker='o', color=cmap(norm(i)), linestyle='', markersize=10, label=name[-
1]) for i, name in enumerate(target_names)]
plt.legend(handles=handles)
plt.show()
```

- t-SNE is a statistical method for visualizing high-dimensional data by reducing the number of dimensions to 2 or 3 while trying to preserve the relative distances between points. t-SNE is effective for exploring the structure of data and for identifying clusters or groups within the data.

Summary

- The notion of distance (or similarity) underpins how K-Nearest Neighbours (KNN) works. It defines nearness between the data points, which is then used to make predictions. The distance metric will affect KNN's prediction perfomances. The optimal choice is usecase specific and to be found through grid search and cross-validation.
- KNN models, or ML in general, are considered good if they consistently perform well on the training dataset and also on an unseen data. We need to find a model that is neither overfitting nor underderfitting.
- Low K generally leads to a complex decision boundary, i.e. high variance (or low bias). In reverse, high K will lead to a simple one, i.e. high bias (or low variance). We need to find K which will result in high validation accuracy and still maintaining high train accuracy.
- The test dataset is a proxy of an unseen (or real world) data. We only use it for final evaluation of the trained model, never use it during the training. To deploy the model or not, we often based on whether the validation and test performances are consistent with each other.
- Features must be in the same scale, either through standard scaling or normalisation, to ensure they are contributing fairly to the KNN (or ML) model. In practice, we always scale our features before training a model, and often this leads to a better prediction performance.