Applied Data Science

L9. k nearest neighbours

Irina Mohorianu

Head of Bioinformatics/ Scientific Computing @CSCI

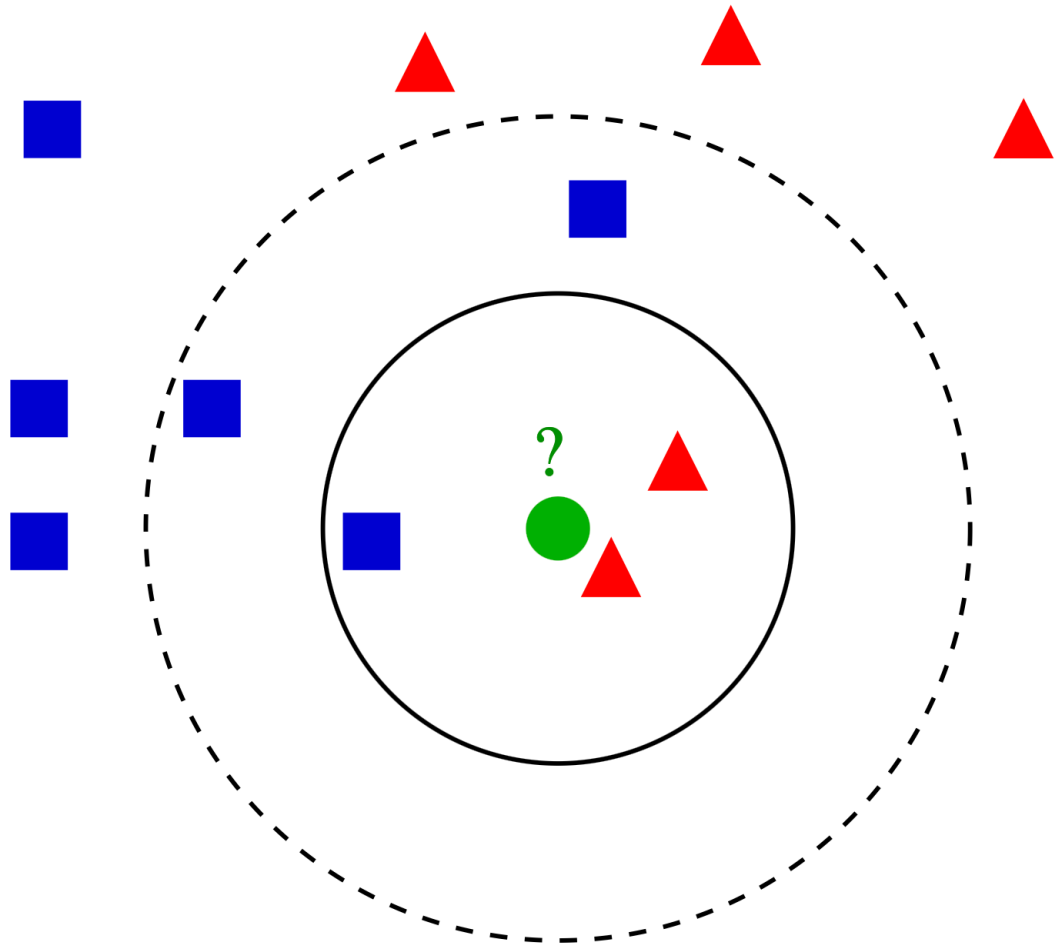
K nearest neighbours. Definitions

k-NN is by far the simplest method of supervised learning we will cover in this course.

It is a non-parametric/ semi-parameteric method that can be used for both classification (predicting class membership) and regression (estimating continuous variables).

k-NN is categorized as instance based (memory based) learning, i.e. all computation is deferred until classification.

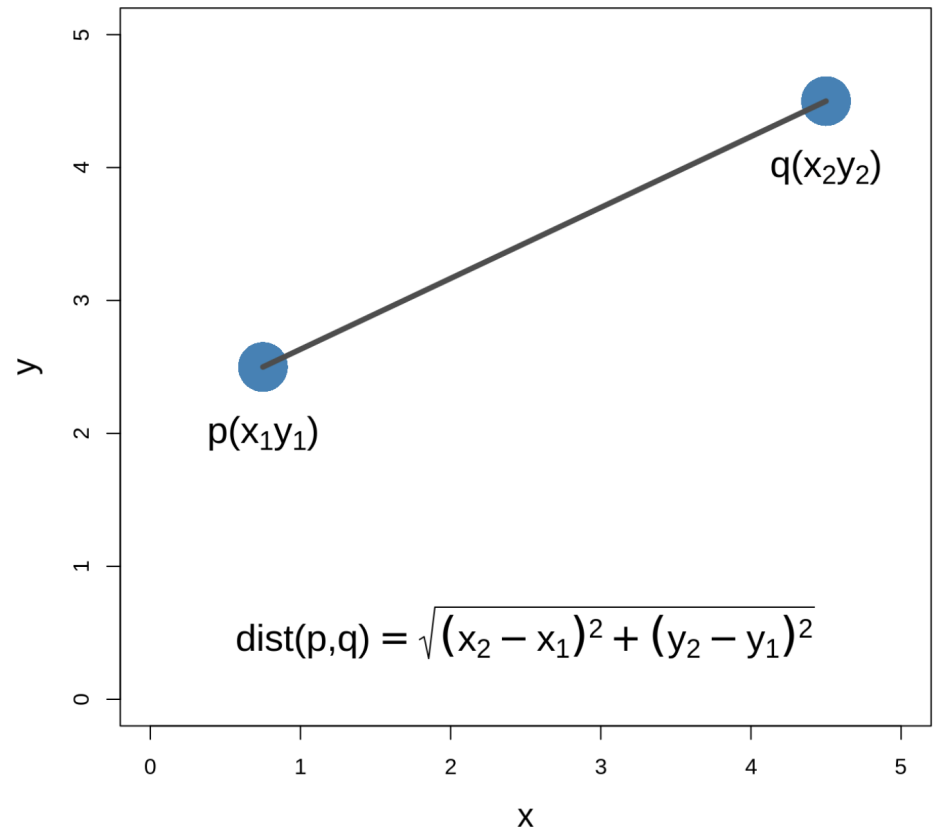
The most computationally demanding aspects of *k*-NN are finding neighbours and storing the entire learning set.



K nearest neighbours. Definitions

Euclidean distance:

$$distance\,(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$



Euclidean Distance

$$dist(p,q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2}$$

Manhattan distance

$$dist(p,q) = \sum_{i=1}^{n}|p_i - q_i|$$

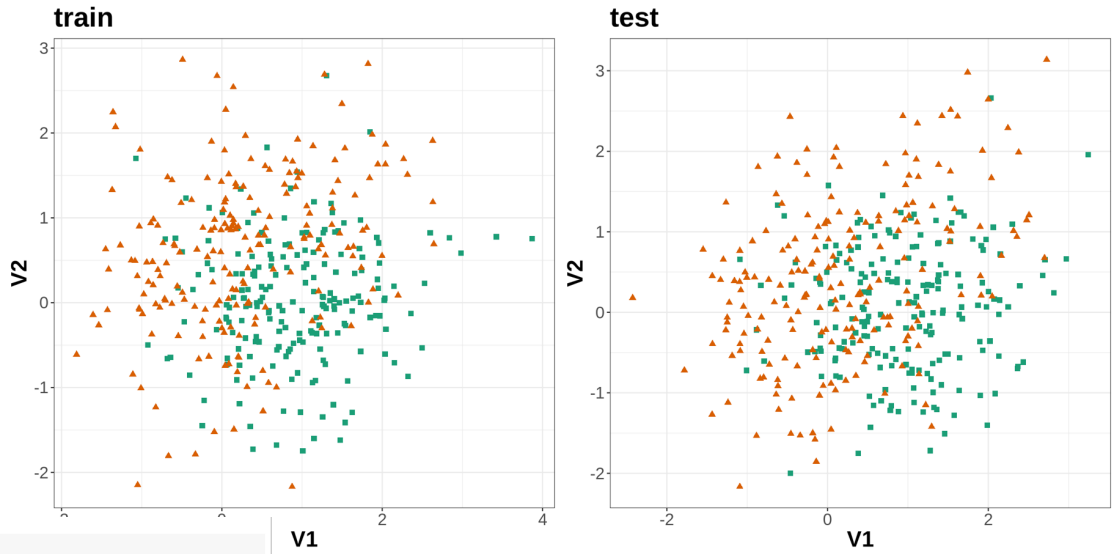
Minkowski distance

$$dist(p,q) = \sqrt[o]{\sum_{i=1}^{n}(p_i - q_i)^o}$$

K nearest neighbours. Algorithm and parameters

A simulated data set will be used to demonstrate:

- bias-variance trade-off
- the knn function in Python
- plotting decision boundaries
- choosing the optimum value of *k*




```
!pip install pyreadr
import pyreadr

data = pyreadr.read_r('bin_class_example.rda')
```

```
[7] str(data)
```

```
'OrderedDict([('xtrain',            V1        V2\n0   -0.222513 -1.152591\n1    0.944435 -0.827482\n2    2.359853 -0.12
8411\n3    1.846116  2.013622\n4    1.731593 -0.574249\n..        ...        ...\n395 -1.025875 -0.079104\n396 -1.050243
-2.148395\n397  0.362549  1.692201\n398 -0.093563  0.878507\n399 -0.989846 -0.133162\n\n[400 rows x 2 columns]), ('xtes
t',            V1        V2\n0    2.085446 -1.008955\n1    2.298474  1.094684\n2    2.071765  0.164437\n3    1.651919
0.324329\n4    1.176965 -0.027658\n..        ...       ...\n395 -0.100715 -1.031759\n396  0.081472  0.578722\n397  0.93
7081  2.439020\n398  2.000068  2.646145\n399 -1.223780  0.120944\n\n[400 rows x 2 columns]), ('ytrain',      ytrain\n0
0.0\n1       0.0\n2       0.0\n3       0.0\n4       0.0\n..      ...\n395     1.0\n396     1.0\n397     1.0\n398     1.
0\n399     1.0\n\n[400 rows x 1 columns]), ('ytest',      ytest\n0      0.0\n1      0.0\n2      0.0\n3      0.0\n4
0.0\n..     ...\n395    1.0\n396    1.0\n…'
```

We illustrate one split into train/ test

K nearest neighbours.
Algorithm and parameters

https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

sklearn.neighbors.KNeighborsClassifier

class `sklearn.neighbors.`**`KNeighborsClassifier`**(*n_neighbors=5, *, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None*) [source]

Parameters:

n_neighbors : *int, default=5*

Number of neighbors to use by default for `kneighbors` queries.

weights : *{'uniform', 'distance'}, callable or None, default='uniform'*

Weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

Refer to the example entitled Nearest Neighbors Classification showing the impact of the `weights` parameter on the decision boundary.

K nearest neighbours.
Algorithm and parameters

algorithm : *{'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'*

Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit` method.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size : *int, default=30*

Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem.

K nearest neighbours.
Algorithm and parameters

The **ball tree nearest-neighbour algorithm** examines nodes in depth-first order, starting at the root.

During the search, the algorithm maintains a max-first priority queue (often implemented with a heap, [Q]), of the k nearest points encountered so far. At each node B, it may perform one of three operations, before finally returning an updated version of the priority queue:

[1] If the distance from the test point t to the current node B is greater than the furthest point in Q,
ignore B and return Q.
[2] If B is a leaf node, scan through every point enumerated in B and
update the nearest-neighbour queue appropriately. Return the updated queue.
[3] If B is an internal node, call the algorithm recursively on B's two children,
searching the child whose center is closer to t first.
Return the queue after each of these calls has updated it in turn.

Performing the recursive search in the order described in point 3 above increases likelihood that the further child will be pruned entirely during the search.

https://en.wikipedia.org/wiki/Ball_tree

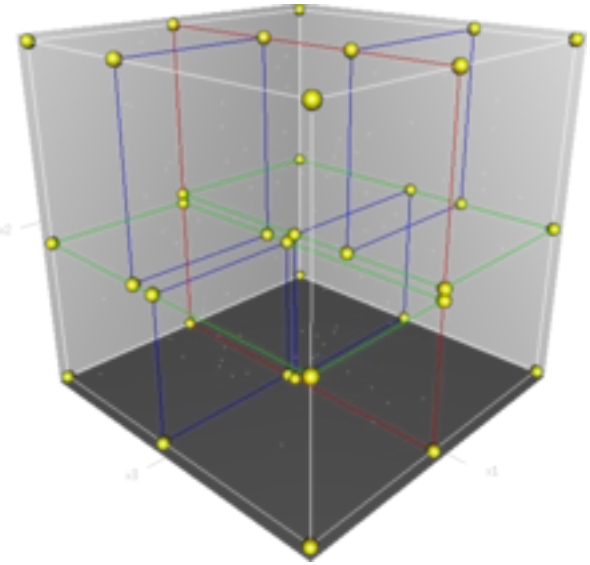
K nearest neighbours. Algorithm and parameters

a k-d tree (short for k-dimensional tree) is a space-partitioning data structure for organizing points in a k-dimensional space.

The k-d tree is a binary tree in which every node is a k-dimensional point.

Every non-leaf node corresponds to a splitting hyperplane that divides the space into two parts, known as half-spaces.

Points to the left of this hyperplane are represented by the left subtree of that node and points to the right of the hyperplane are represented by the right subtree.



https://en.wikipedia.org/wiki/K-d_tree

K nearest neighbours. Algorithm and parameters

p : *float, default=2*
Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric : *str or callable, default='minkowski'*
Metric to use for distance computation. Default is "minkowski", which results in the standard Euclidean distance when p = 2. See the documentation of scipy.spatial.distance and the metrics listed in `distance_metrics` for valid metric values.

If metric is "precomputed", X is assumed to be a distance matrix and must be square during fit. X may be a sparse graph, in which case only "nonzero" elements may be considered neighbors.

If metric is a callable function, it takes two arrays representing 1D vectors as inputs and must return one value indicating the distance between those vectors. This works for Scipy's metrics, but is less efficient than passing the metric name as a string.

metric_params : *dict, default=None*
Additional keyword arguments for the metric function.

n_jobs : *int, default=None*
The number of parallel jobs to run for neighbors search. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details. Doesn't affect `fit` method.

K nearest neighbours. Algorithm and parameters

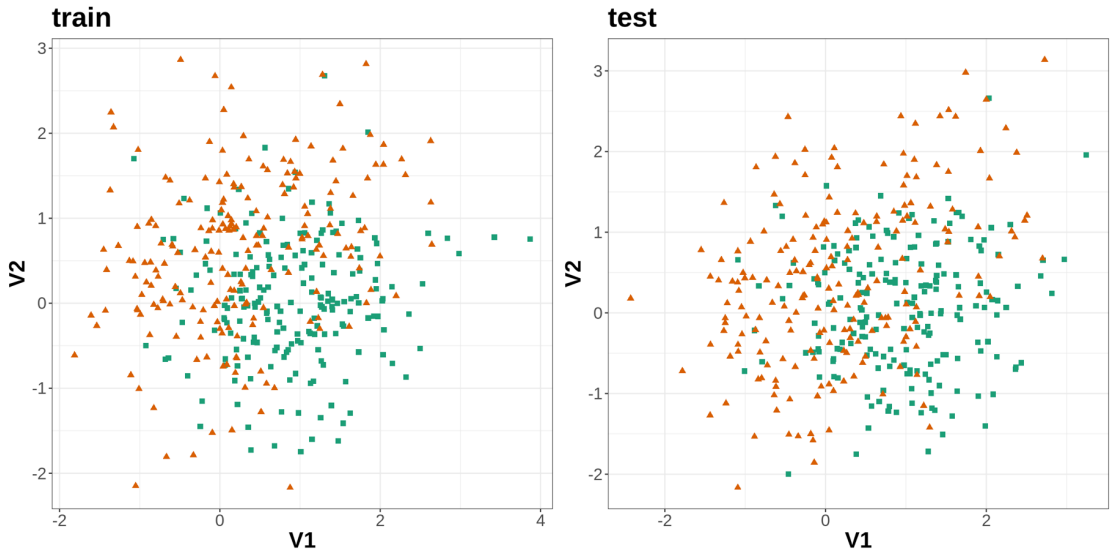
Examples

```
>>> X = [[0], [1], [2], [3]]
>>> y = [0, 0, 1, 1]
>>> from sklearn.neighbors import KNeighborsClassifier
>>> neigh = KNeighborsClassifier(n_neighbors=3)
>>> neigh.fit(X, y)
KNeighborsClassifier(...)
>>> print(neigh.predict([[1.1]]))
[0]
>>> print(neigh.predict_proba([[0.9]]))
[[0.666... 0.333...]]
```

Methods

`fit`(X, y)	Fit the k-nearest neighbors classifier from the training dataset.
`get_metadata_routing`()	Get metadata routing of this object.
`get_params`([deep])	Get parameters for this estimator.
`kneighbors`([X, n_neighbors, return_distance])	Find the K-neighbors of a point.
`kneighbors_graph`([X, n_neighbors, mode])	Compute the (weighted) graph of k-Neighbors for points in X.
`predict`(X)	Predict the class labels for the provided data.
`predict_proba`(X)	Return probability estimates for the test data X.
`score`(X, y[, sample_weight])	Return the mean accuracy on the given test data and labels.
`set_params`(**params)	Set the parameters of this estimator.
`set_score_request`(*[, sample_weight])	Request metadata passed to the `score` method.

K nearest neighbours. Example 1




```
knn_classifier = KNeighborsClassifier(n_neighbors=1)
knn_classifier.fit(x_train, y_train)

# Predict on the training data
knn1_train_predictions = knn_classifier.predict(x_train)

# Compute confusion matrix
conf_matrix = confusion_matrix(y_train, knn1_train_predictions)

print(conf_matrix)
print(classification_report(y_train, knn1_train_predictions))
```

```
[[200   0]
 [  0 200]]
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00       200
         1.0       1.00      1.00      1.00       200

    accuracy                           1.00       400
   macro avg       1.00      1.00      1.00       400
weighted avg       1.00      1.00      1.00       400
```

K nearest neighbours. Example 1

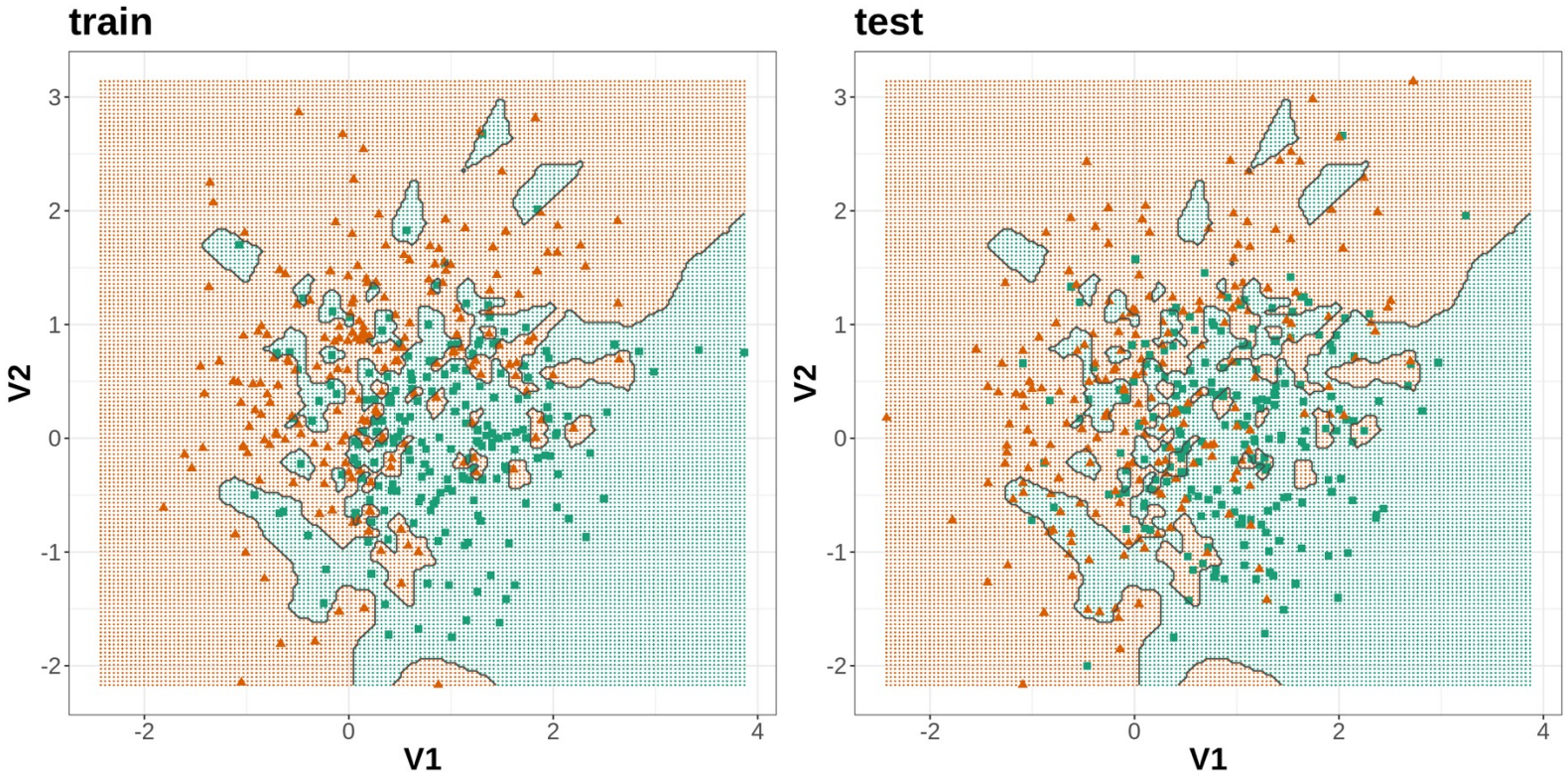
```
knn_classifier = KNeighborsClassifier(n_neighbors=1)
knn_classifier.fit(x_train, y_train)

# Predict on the training data
knn1_train_predictions = knn_classifier.predict(x_train)

# Compute confusion matrix
conf_matrix = confusion_matrix(y_train, knn1_train_predictions)

print(conf_matrix)
print(classification_report(y_train, knn1_train_predictions))
```

```
[[200   0]
 [  0 200]]
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00       200
         1.0       1.00      1.00      1.00       200

    accuracy                           1.00       400
   macro avg       1.00      1.00      1.00       400
weighted avg       1.00      1.00      1.00       400
```

```
# Predict on the testing data
knn1_test_predictions = knn_classifier.predict(x_test)

conf_matrix = confusion_matrix(y_test, knn1_test_predictions)

print(conf_matrix)
print(classification_report(y_test, knn1_test_predictions))
```

```
[[131  69]
 [ 81 119]]
              precision    recall  f1-score   support

         0.0       0.62      0.66      0.64       200
         1.0       0.63      0.59      0.61       200

    accuracy                           0.62       400
   macro avg       0.63      0.62      0.62       400
weighted avg       0.63      0.62      0.62       400
```

K nearest neighbours.
Example 1



Binary classification of the simulated training and test sets with k=1.

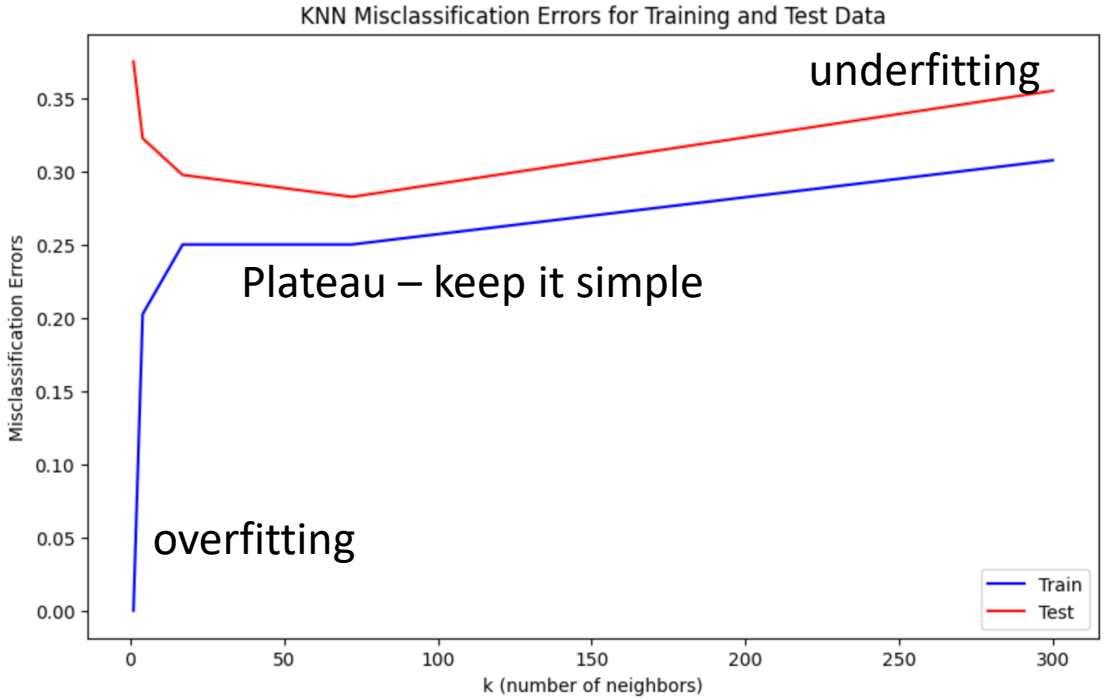
The model overfits on the training data and fails to generalise on the test data

K nearest neighbours. Bias/variance trade-off

The bias–variance tradeoff is the problem of simultaneously minimizing two sources of error that prevent supervised learning algorithms from generalizing beyond their training set

•The bias is error from erroneous assumptions in the learning algorithm. High bias can cause an algorithm to miss the relevant relations between features and target outputs (underfitting).

•The variance is error from sensitivity to small fluctuations in the training set. High variance can cause an algorithm to model the random noise in the training data, rather than the intended outputs (overfitting).
To demonstrate this phenomenon, let us look at the performance of the *k*-nn classifier over a range of values of *k*.

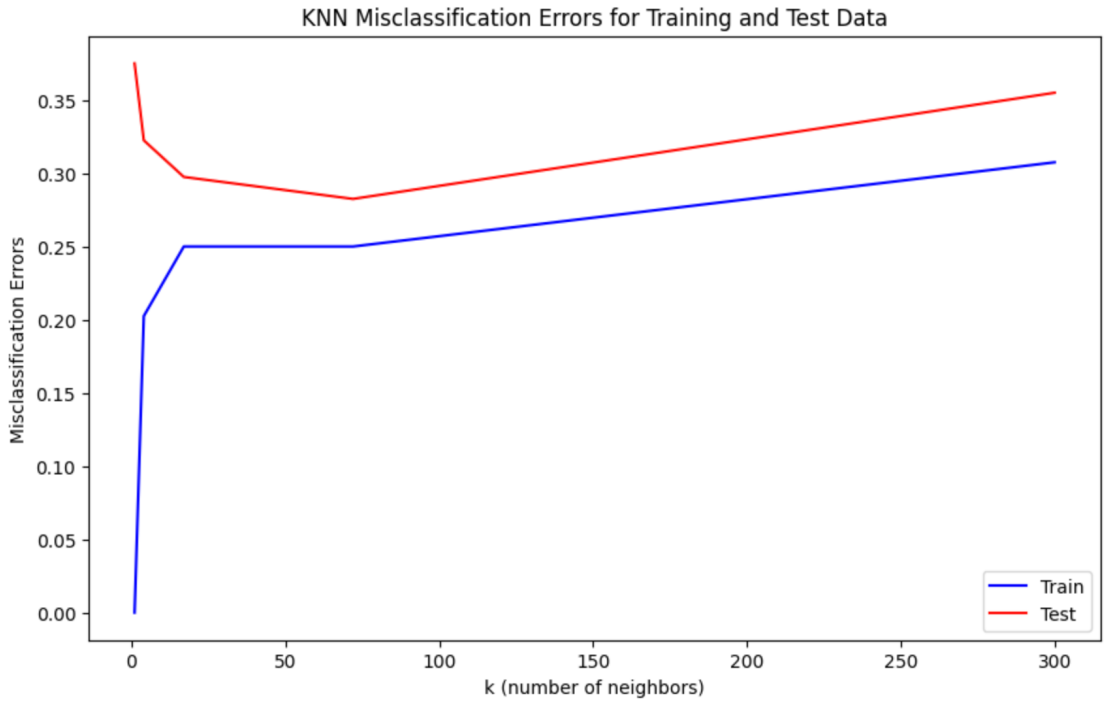


K nearest neighbours. Bias/variance trade-off

```
from sklearn.model_selection import RepeatedKFold

# 10 fold 10 repeats CV
repeated_cv = RepeatedKFold(n_splits=10, n_repeats=10, random_state=42)
```

```
import numpy as np
from sklearn.model_selection import RepeatedKFold, GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import cohen_kappa_score, make_scorer, accuracy_score

np.random.seed(42)

log_spaced_array = np.logspace(np.log10(1), np.log10(300), num=50)
log_spaced_list = sorted(list(set(map(int, log_spaced_array))))

param_grid = {'n_neighbors': log_spaced_list}

# Create custom scorers for grid search
scoring = {
    'accuracy': make_scorer(accuracy_score),
    'kappa': make_scorer(cohen_kappa_score)
}

# Fit model with repeated CV and custom scoring created.
knn = KNeighborsClassifier()

grid_search = GridSearchCV(knn, param_grid, cv=repeated_cv, scoring=scoring, refit='accuracy', return_train_score=True)

grid_search.fit(x_train, y_train.values.ravel())

results = grid_search.cv_results_
```

K nearest neighbours. Bias/variance trade-off



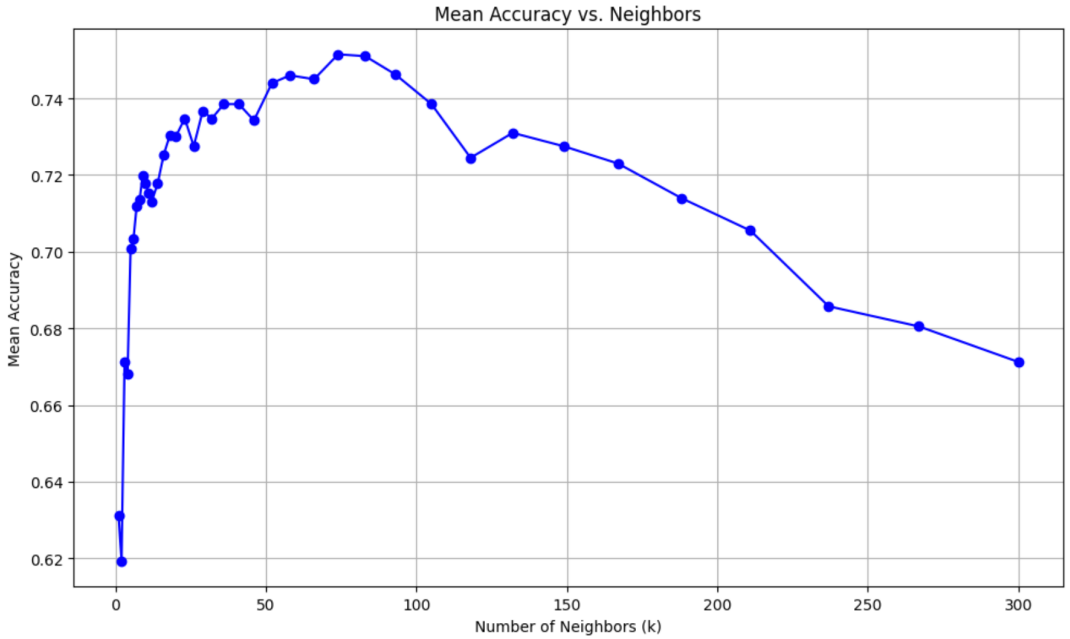
k	Mean Accuracy	Mean Kappa
1	0.63125	0.261172
2	0.61925	0.238327
3	0.67125	0.341646
4	0.66800	0.334981
5	0.70075	0.399034
6	0.70350	0.404591
7	0.71175	0.421534
8	0.71350	0.424377
9	0.71975	0.436896
10	0.71775	0.432499
11	0.71525	0.428060
12	0.71300	0.423903
14	0.71775	0.432876
16	0.72525	0.446702
18	0.73025	0.457874
20	0.73000	0.456828
23	0.73475	0.466782
26	0.72750	0.452070
29	0.73675	0.470500
32	0.73475	0.466615
36	0.73850	0.474087
41	0.73850	0.474532
46	0.73425	0.466406
52	0.74400	0.485740
58	0.74600	0.489606
66	0.74500	0.487576
74	0.75150	0.499835
83	0.75100	0.499246
93	0.74625	0.490176
05	0.73850	0.475364
18	0.72450	0.446889
32	0.73100	0.459514
49	0.72750	0.453482
67	0.72300	0.445095
88	0.71400	0.427814

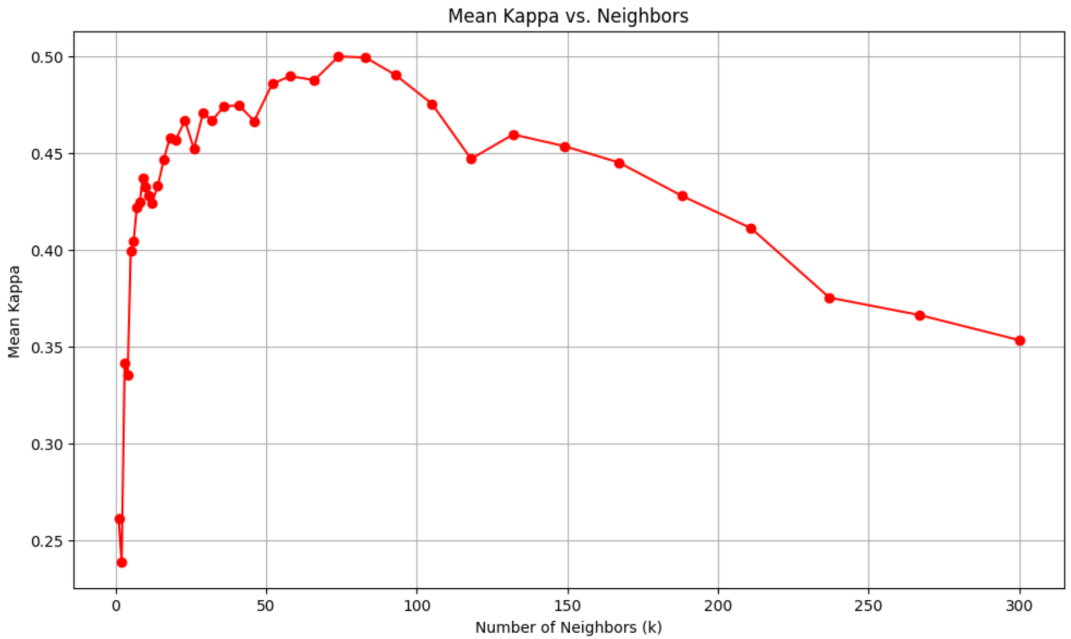
Cohen's Kappa

Cohen's Kappa:
O: observed accuracy, E: expected accuracy based on marginal totals of the confusion matrix.
Values between -1 .. 1;
0 corresponds to a random assignment

$$Kappa = \frac{O - E}{1 - E}$$

K nearest neighbours.
Bias/variance trade-off

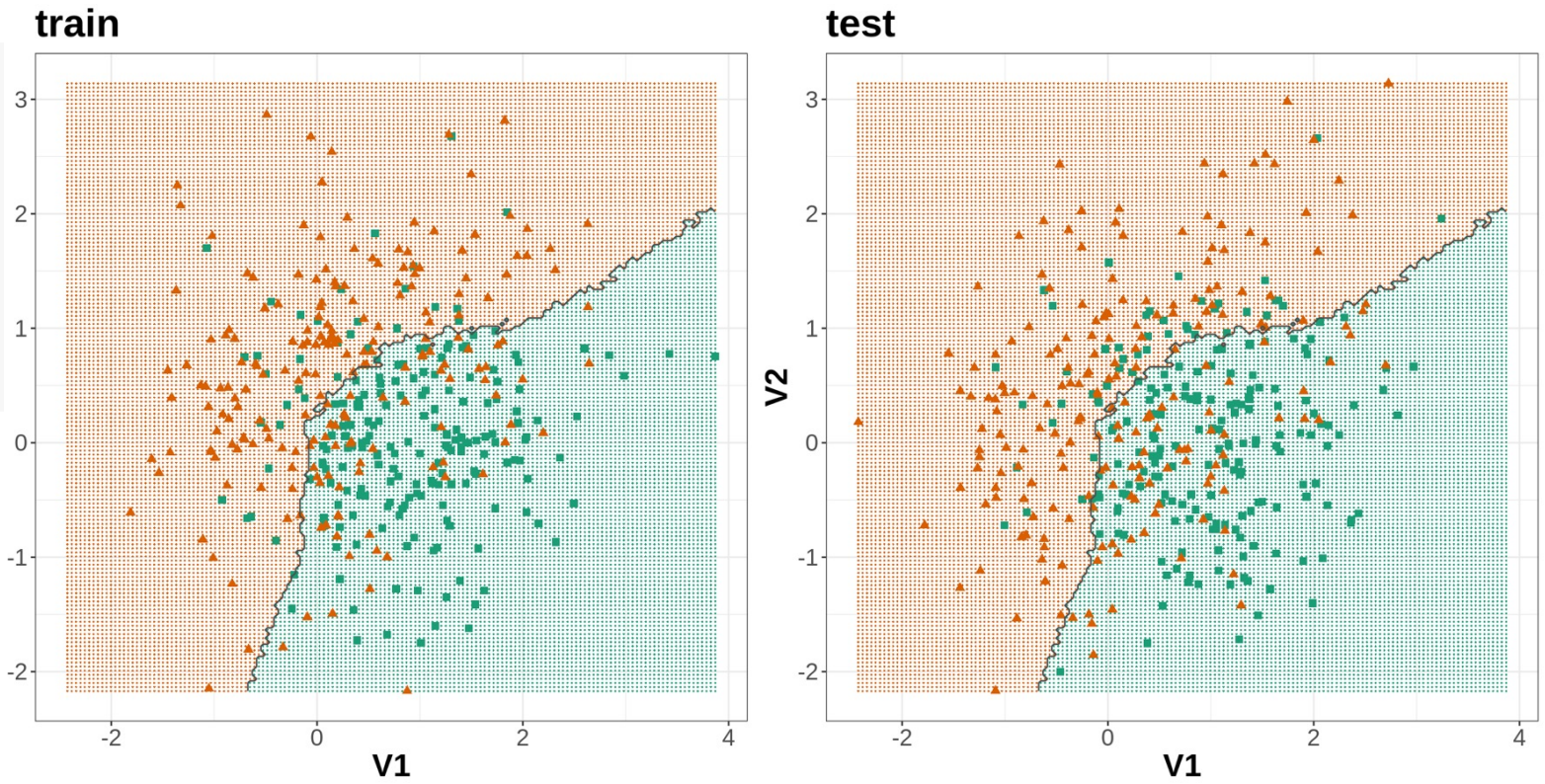




K nearest neighbours. Bias/variance trade-off

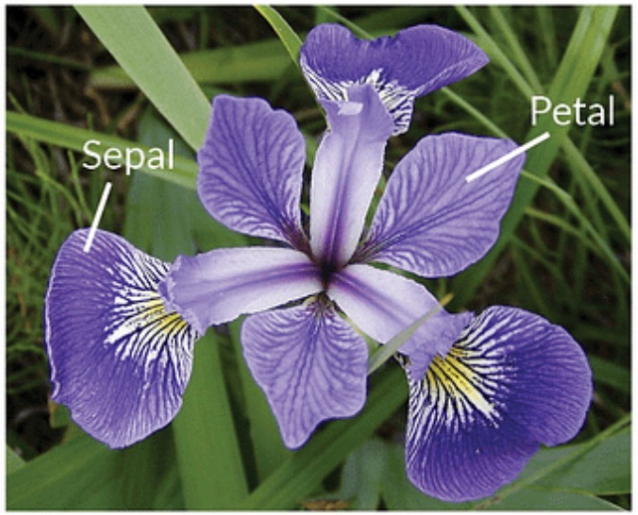
```python
from sklearn.metrics import confusion_matrix, classification_report

test_pred = grid_search.best_estimator_.predict(x_test)

# Generate the confusion matrix
conf_matrix = confusion_matrix(y_test, test_pred)

print(conf_matrix)

report = classification_report(y_test, test_pred)
print(report)
```

```
[[154  46]
 [ 69 131]]
              precision    recall  f1-score   support

           0       0.69      0.77      0.73       200
           1       0.74      0.66      0.69       200

    accuracy                           0.71       400
   macro avg       0.72      0.71      0.71       400
weighted avg       0.72      0.71      0.71       400
```



Binary classification of the simulated training and test sets with k=83.

kNN on the iris dataset



Iris Versicolor

Iris Setosa

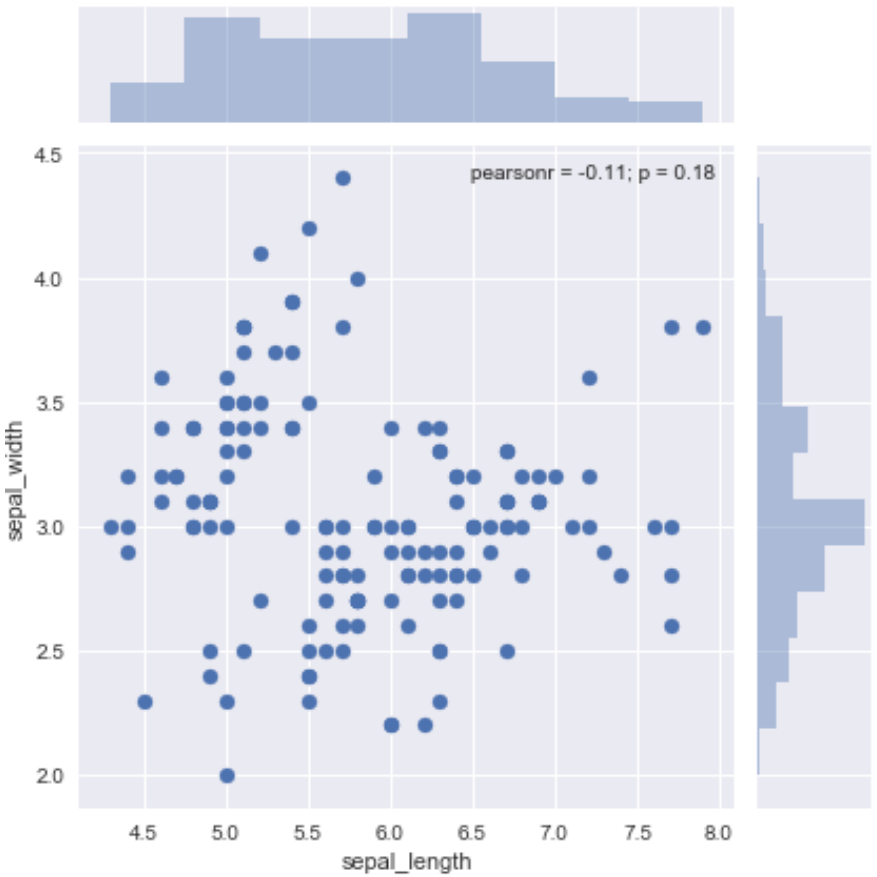
Iris Virginica

```
from sklearn.datasets import load_iris

# Load the iris dataset
data = load_iris()

# Convert to pandas DataFrame
iris_df = pd.DataFrame(data.data, columns=data.feature_names)
iris_df['species'] = data.target
```

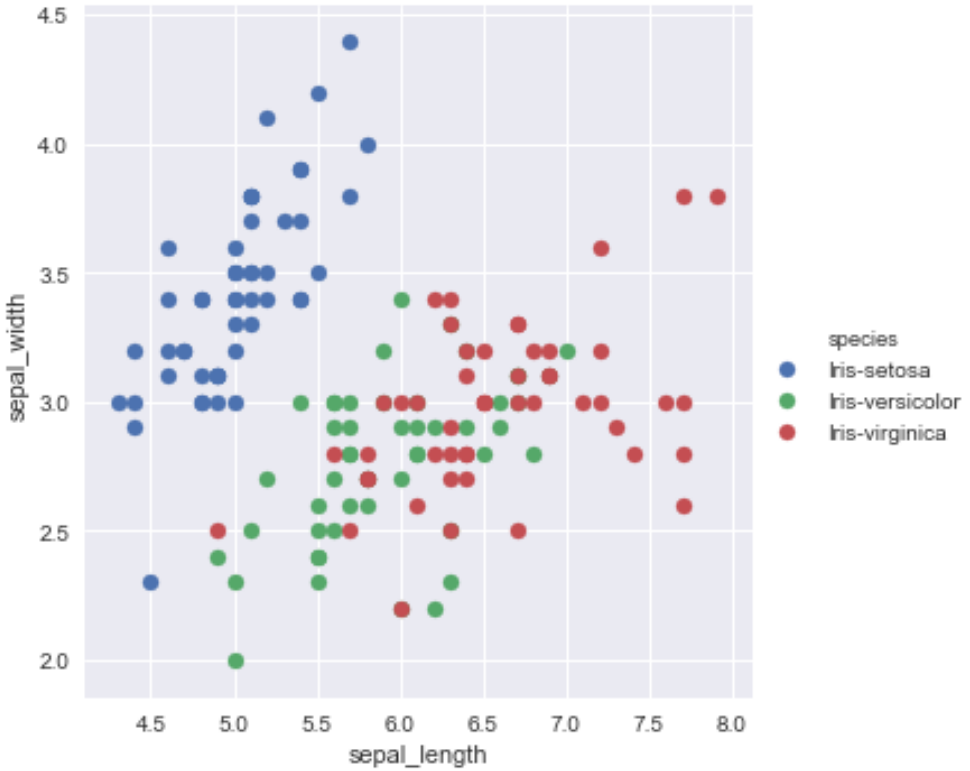
kNN on the iris dataset

```
sns.jointplot(x = 'sepal_length', y = 'sepal_width', data = df)
```

```
<seaborn.axisgrid.JointGrid at 0x4d74940>
```



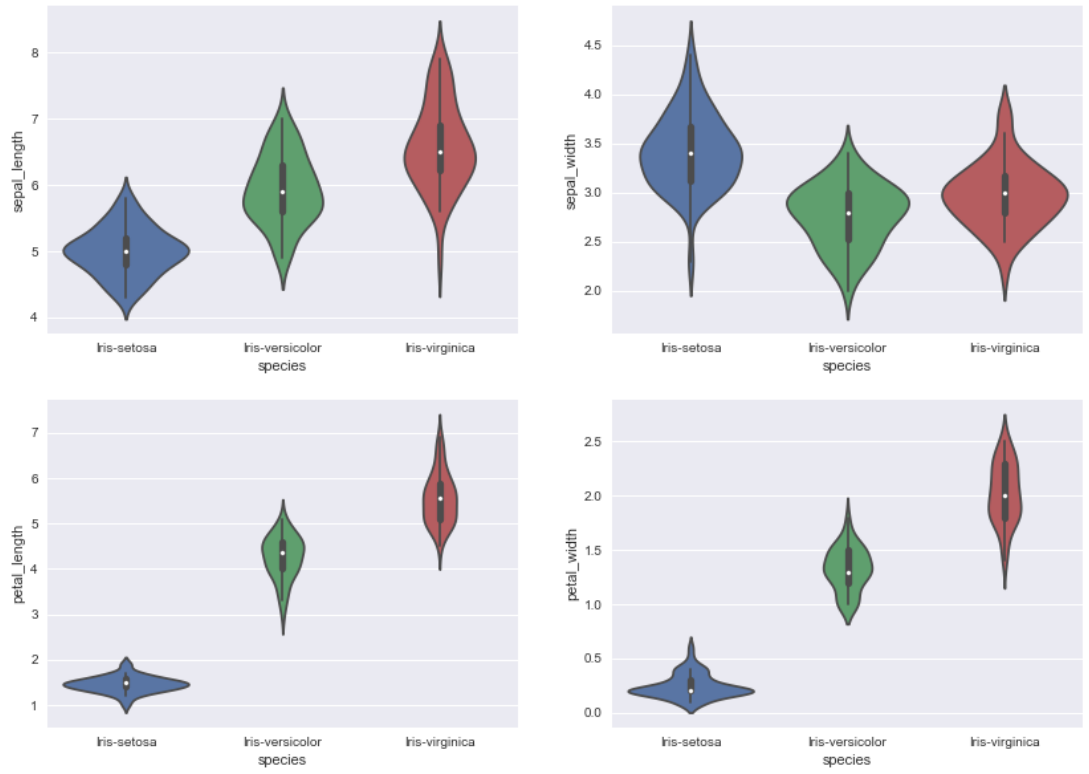

```
#seaborn scatterplot by species on sepal_length vs sepal_width
g = sns.FacetGrid(df, hue='species', size=5)
g = g.map(plt.scatter, 'sepal_length', 'sepal_width').add_legend()
```



kNN on the iris dataset

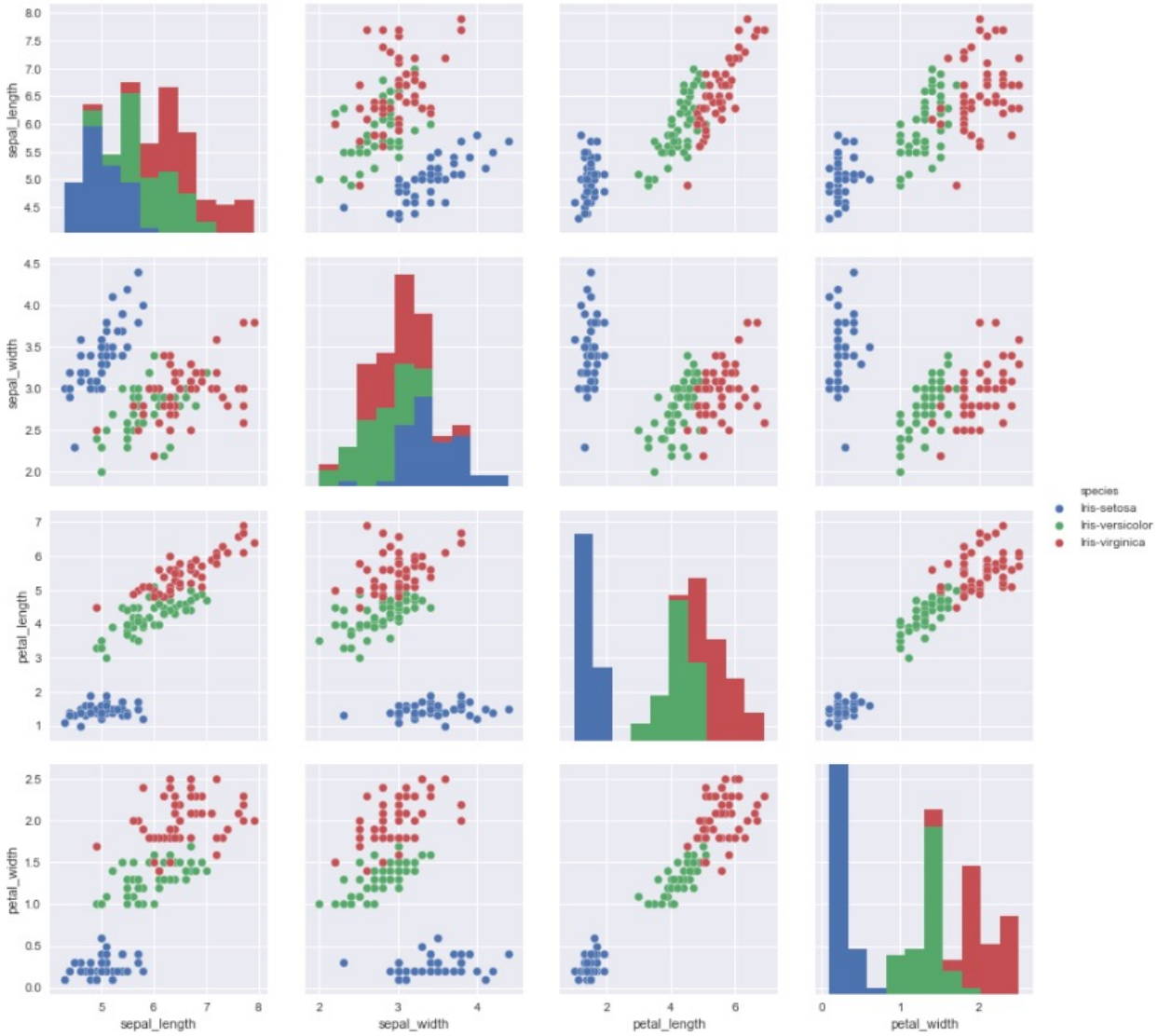
```
# The violinplot shows density of the length and width in the species
# Denser regions of the data are fatter, and sparser thiner in a violin plot
plt.figure(figsize=(14,10))
plt.subplot(2,2,1)
sns.violinplot(x='species', y='sepal_length', data=df, size=5)
plt.subplot(2,2,2)
sns.violinplot(x='species', y='sepal_width', data=df, size=5)
plt.subplot(2,2,3)
sns.violinplot(x='species', y='petal_length', data=df, size=5)
plt.subplot(2,2,4)
sns.violinplot(x='species', y='petal_width', data=df, size=5)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0xc4a4da0>
```




```
sns.pairplot(data = df, hue = 'species', size = 3)
```

```
<seaborn.axisgrid.PairGrid at 0xc74a390>
```



kNN on the iris dataset

```
X = df.drop("species", axis=1)
X.head()
```

	sepal_length	sepal_width	petal_length	petal_width
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

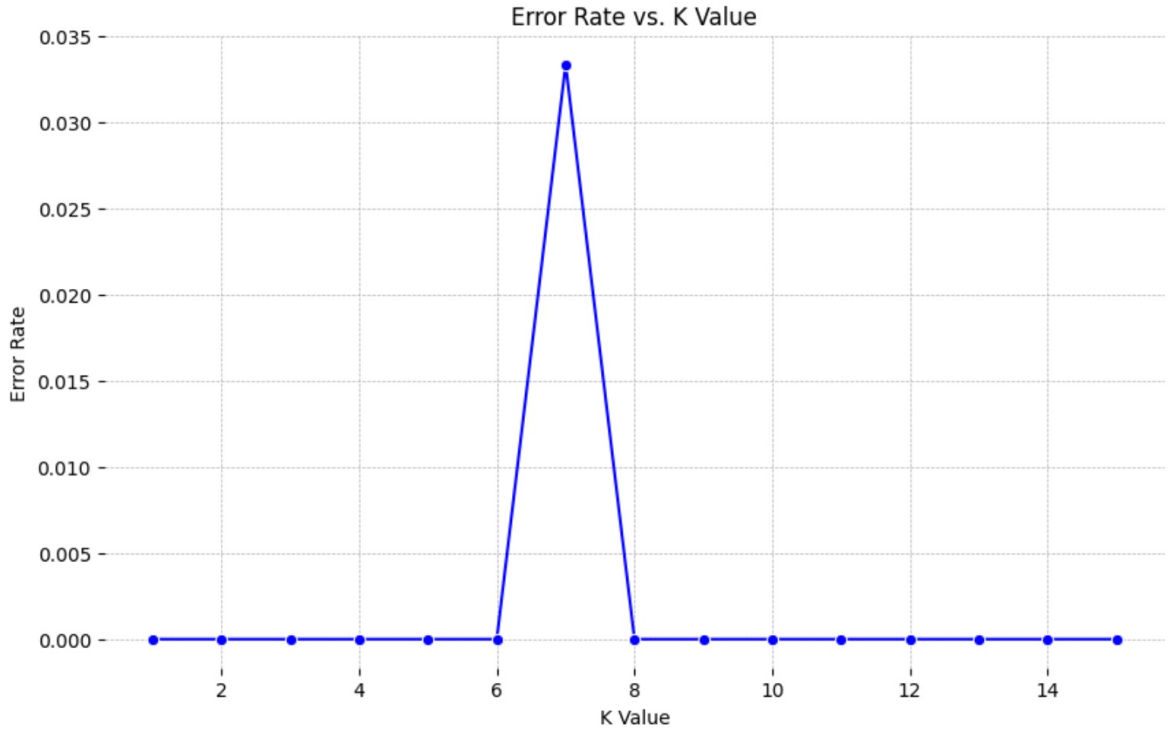
```
y = df["species"]
y.head()
```

```
0    Iris-setosa
1    Iris-setosa
2    Iris-setosa
3    Iris-setosa
4    Iris-setosa
Name: species, dtype: object
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
classifier = KNeighborsClassifier(n_neighbors=5)
classifier.fit(X_train, y_train)
```

kNN on the iris dataset

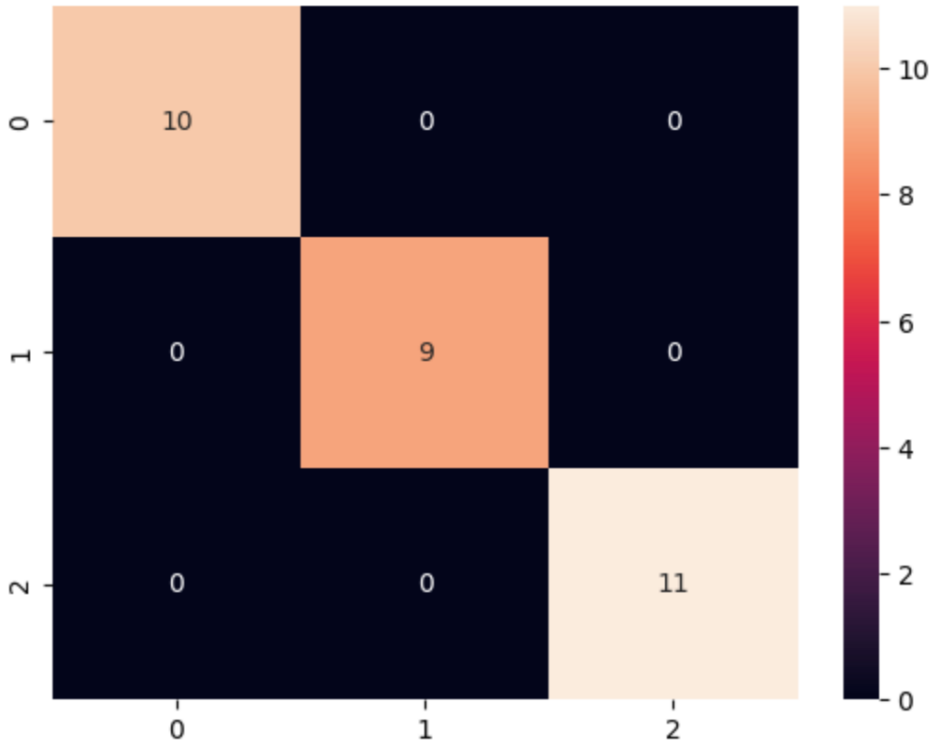
```
errors = []

# Loop over k values
for i in range(1, 16):
    knn = KNeighborsClassifier(n_neighbors=i)
    knn.fit(X_train, y_train)
    score = knn.score(X_test, y_test)
    errors.append(1 - score)  # 1 - accuracy is the error

# Plotting
plt.figure(figsize=(10, 6))
sns.lineplot(x=range(1, 16), y=errors, marker='o', color='blue')
plt.title('Error Rate vs. K Value')
plt.xlabel('K Value')
plt.ylabel('Error Rate')
plt.grid(True, which='both', linestyle='--', linewidth=0.5)
sns.despine(left=True, bottom=True)
plt.show()
```



kNN on the iris dataset

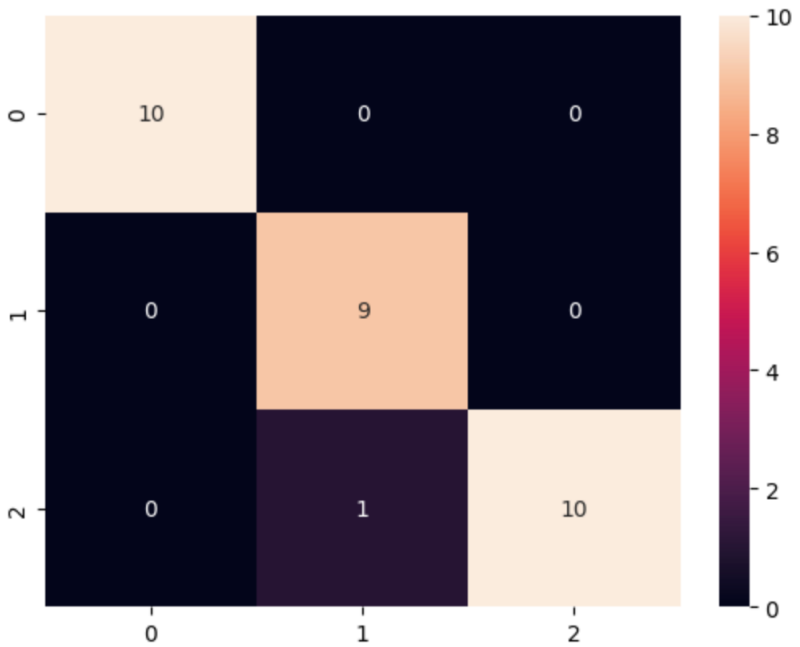
```
cm = confusion_matrix(y_test, y_pred)
print(cm)
sns.heatmap(cm, annot=True)
```

```
[[10  0  0]
 [ 0  9  0]
 [ 0  0 11]]
 <Axes: >
```




```
cm2 = confusion_matrix(y_test, y_pred2)
print(cm2)
sns.heatmap(cm2, annot=True)
```

```
[10  0  0]
 [ 0  9  0]
 [ 0  1 10]]
 <Axes: >
```



More kNN examples.
Cell Segmentation example [classification]

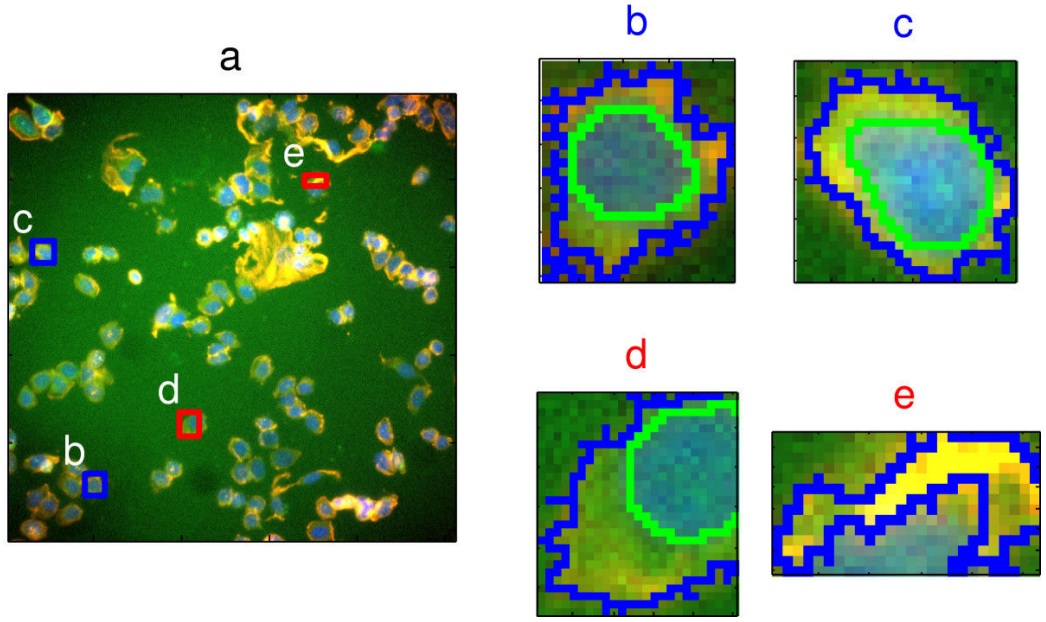


Image segmentation in high content screening. Images b and c are examples of well-segmented cells; d and e show poor-segmentation.
Source: Hill et al (2007)
https://doi.org/10.1186/1471-2105-8-340

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

from sklearn.model_selection import RepeatedKFold, GridSearchCV
from sklearn.feature_selection import VarianceThreshold
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import PowerTransformer, StandardScaler
from sklearn.pipeline import Pipeline
from yellowbrick.model_selection import ValidationCurve
import numpy as np

from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import matplotlib.pyplot as plt
```

```
segmentationData = pd.read_csv("segDF.csv")
segmentationData = segmentationData.drop("Unnamed: 0", axis=1)
```

More kNN examples.
Cell Segmentation example [classification]

	AngleCh1	AreaCh1	AvgIntenCh1	AvgIntenCh2	AvgIntenCh3	AvgIntenCh4	ConvexHullAreaRatioCh1	ConvexHullPerimRatioCh1	DiffIntenDensityCh1	DiffIntenDensityCh3
0	143.247705	185	15.711864	4.954802	9.548023	2.214689	1.124509	0.919683	29.519231	13.775641
1	133.752037	819	31.923274	206.878517	69.916880	164.153453	1.263158	0.797080	31.875000	43.122283
2	106.646387	431	28.038835	116.315534	63.941748	106.696602	1.053310	0.935475	32.487710	35.985770
3	69.150325	298	19.456140	102.294737	28.217544	31.028070	1.202625	0.865829	26.732283	22.917323
4	2.887837	285	24.275735	112.415441	20.474265	40.577206	1.109333	0.956812	31.580645	21.709677
...	...	...	...	...	...	...	...	...	...	...
2014	85.481047	302	42.041522	24.467128	143.532872	233.325260	1.509709	0.783153	51.529183	93.762646
2015	99.049010	607	38.803448	21.587931	78.372414	82.200000	1.683524	0.784737	46.776930	65.911344
2016	83.319801	204	37.866667	6.994872	59.789744	148.235897	1.160976	0.943333	48.594203	44.942029
2017	116.473894	390	45.643432	170.123324	147.498660	338.117962	1.057605	0.937941	53.972740	93.992826
2018	150.820416	441	35.641330	119.372922	96.738717	191.783848	1.109348	0.882904	48.688378	66.264368

2019 rows × 57 columns

```
segClass = segmentationData['Class']
segData = segmentationData.iloc[:, 1:-1]

# Splitting data into training and testing sets
trainIndex, testIndex = train_test_split(segData.index, test_size=0.5, stratify=segClass, random_state=42)

segDataTrain = segData.loc[trainIndex]
segDataTest = segData.loc[testIndex]
segClassTrain = segClass.loc[trainIndex]
segClassTest = segClass.loc[testIndex]
```

More kNN examples.
Cell Segmentation example [classification]
Near Zero variance

```
[ ] # Get features with near-zero variance
    selector = VarianceThreshold(threshold=1e-5)
    segDataTrain_selected = selector.fit_transform(segDataTrain)
```

```
[ ] # Get boolean mask of retained columns
    retained_columns_mask = selector.get_support()

    # Identify the columns that were dropped
    dropped_columns = segDataTrain.columns[~retained_columns_mask]

    print("Columns removed due to near-zero variance:")
    print(dropped_columns)
```

```
Columns removed due to near-zero variance:
Index([], dtype='object')
```

No features had nzv.

More kNN examples.
Cell Segmentation example [classification]

Scaling

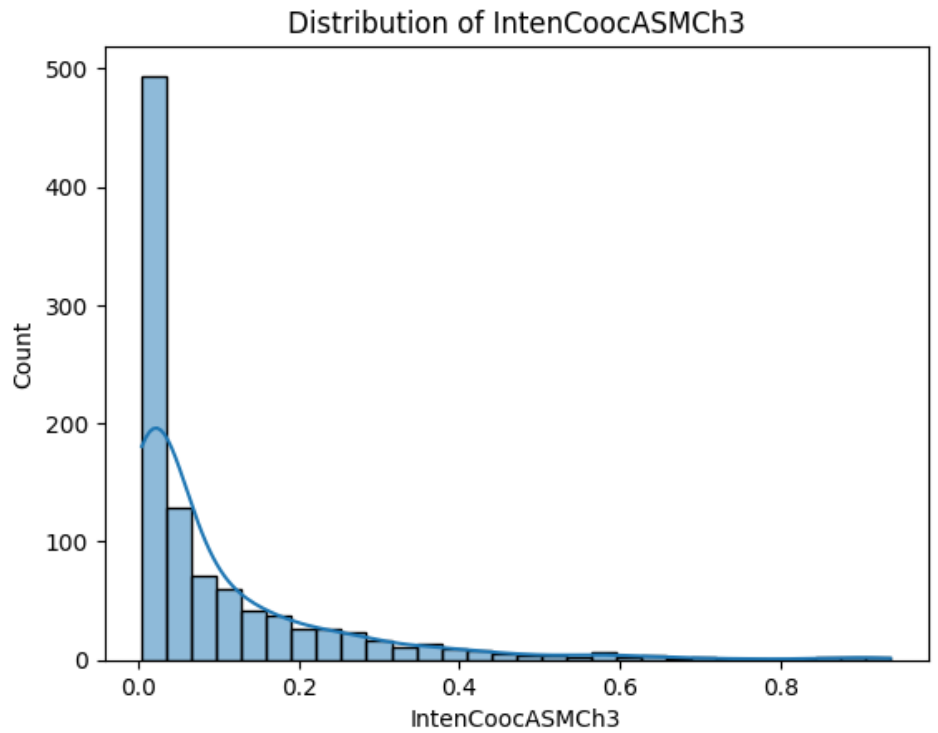
```
[ ] # Get summary of distirbution before scaling.
    print(segDataTrain['IntenCoocASMCh4'].describe())
    print(segDataTrain['TotalIntenCh2'].describe())
```

```
count    1009.000000
mean        0.099101
std         0.131095
min         0.004514
25%         0.018337
50%         0.050857
75%         0.114867
max         0.881854
Name: IntenCoocASMCh4, dtype: float64
count      1009.000000
mean      53278.230922
std       45686.265260
min           1.000000
25%       17357.000000
50%       50798.000000
75%       73474.000000
max      302485.000000
Name: TotalIntenCh2, dtype: float64
```

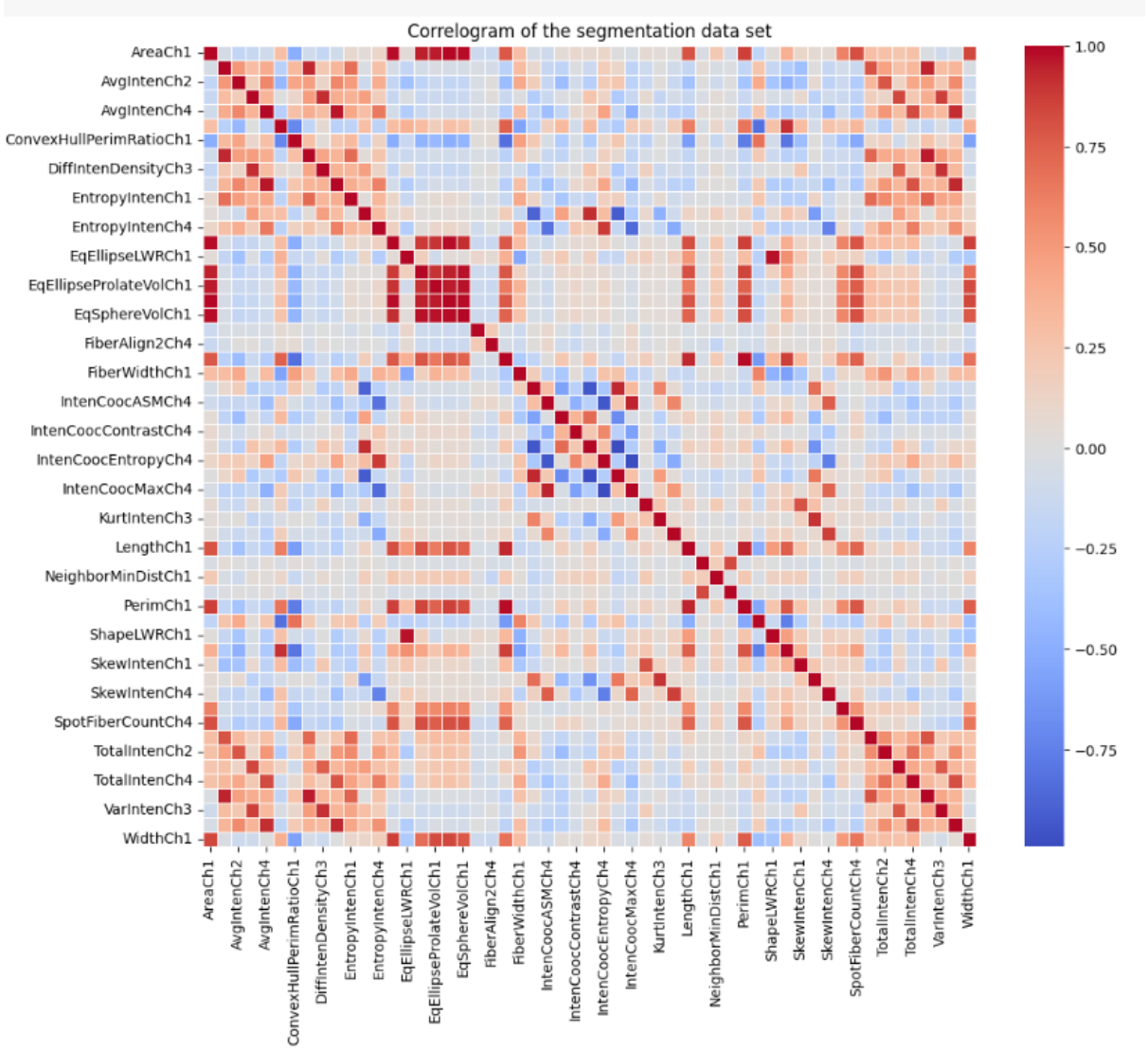
```
[ ] from sklearn.preprocessing import StandardScaler

    # Scaling
    scaler = StandardScaler()

    segDataTrain_scaled = pd.DataFrame(scaler.fit_transform(segDataTrain), columns=segDataTrain.columns)
    segDataTest_scaled = pd.DataFrame(scaler.transform(segDataTest), columns=segDataTest.columns)
```

Cell Segmentation example [classification] Skewness

```python
import seaborn as sns
import matplotlib.pyplot as plt

# Plotting the distribution of a variable to check skewness
sns.histplot(segDataTrain['IntenCoocASMCh3'], bins=30, kde=True)
plt.xlabel("IntenCoocASMCh3")
plt.title("Distribution of IntenCoocASMCh3")
plt.show()
```



Cell Segmentation example [classification]
Collinearity [assessment of correlations]



More kNN examples.
Cell Segmentation example [classification]

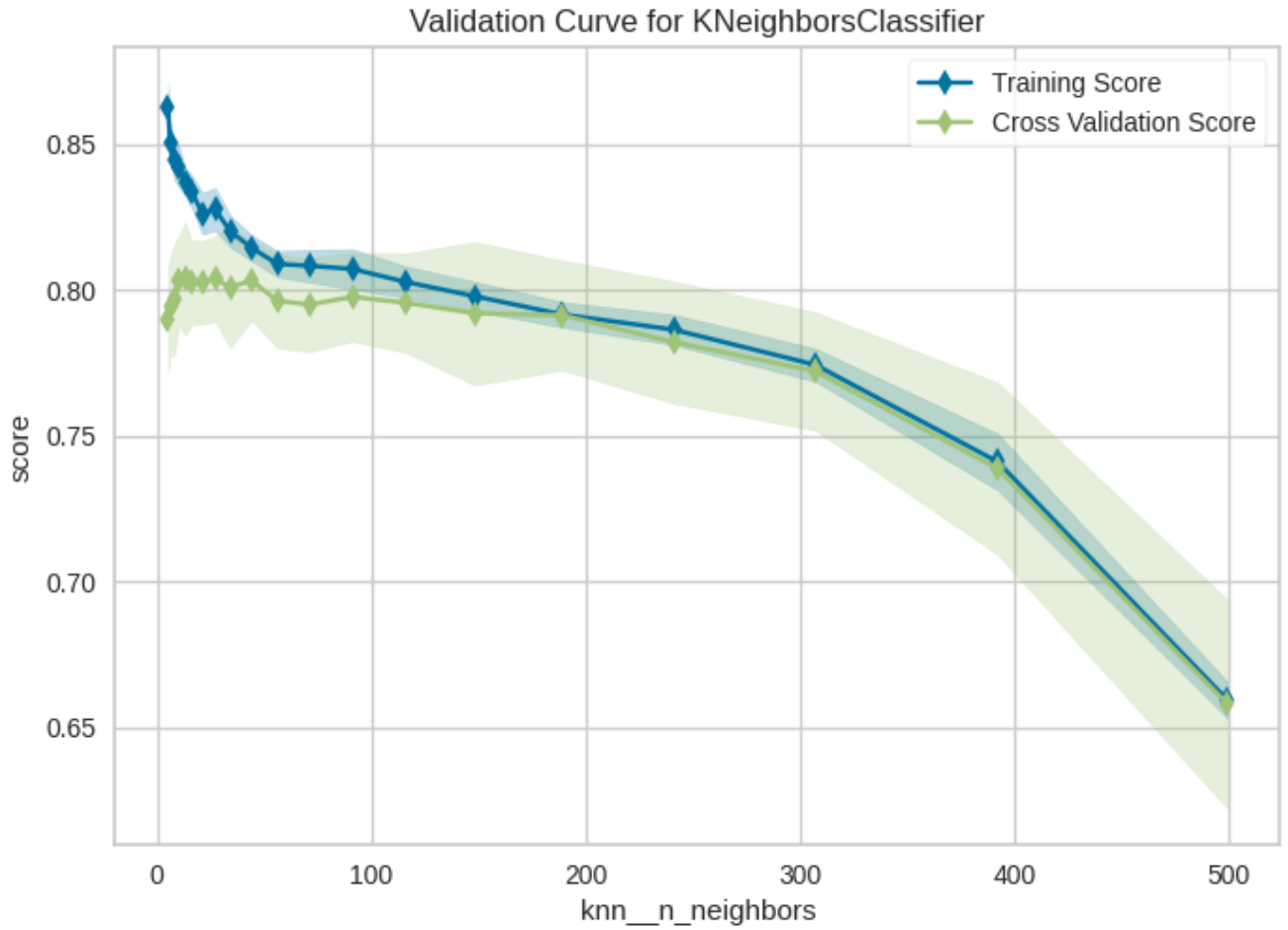
Fit model

```
cv = RepeatedKFold(n_splits=5, n_repeats=3, random_state=42)

tuneParam = {'knn__n_neighbors': list(np.logspace(np.log10(5), np.log10(500), 20).astype(int))}

# Pipeline for Preprocessing and Model Training
pipeline = Pipeline([
    ('yeojohnson', PowerTransformer(method='yeo-johnson')),
    ('scaler', StandardScaler()),
    ('knn', KNeighborsClassifier())
])

# Training the Model using GridSearchCV
grid = GridSearchCV(pipeline, param_grid=tuneParam, cv=cv, n_jobs=-1, return_train_score=True)
grid.fit(segDataTrain, segClassTrain)

results = pd.DataFrame(grid.cv_results_)

# Create a summary table - similar to what we get in R.
k_values = results['param_knn__n_neighbors']
mean_test_scores = results['mean_test_score']
std_test_scores = results['std_test_score']

summary_table = pd.DataFrame({
    'k_value': k_values,
    'mean_accuracy': mean_test_scores,
    'std_accuracy': std_test_scores
})

print(summary_table)
```

More kNN examples.
Cell Segmentation example [classification]

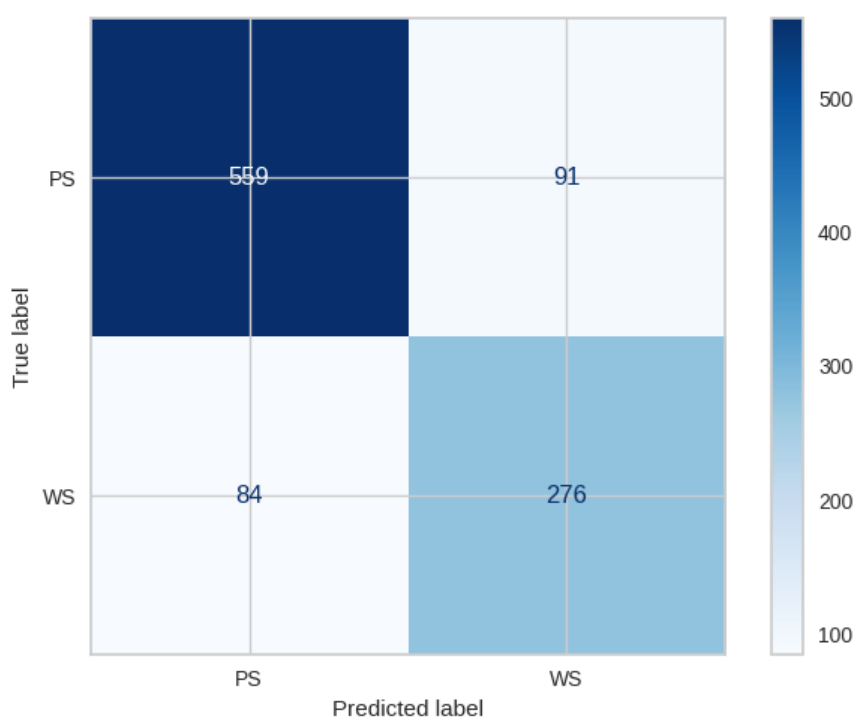
	k_value	mean_accuracy	std_accuracy
0	5	0.789557	0.017736
1	6	0.794184	0.016865
2	8	0.796828	0.019137
3	10	0.803105	0.015283
4	13	0.803752	0.019368
5	16	0.802430	0.014434
6	21	0.802430	0.014317
7	27	0.803767	0.014669
8	34	0.800798	0.020817
9	44	0.803100	0.013542
10	56	0.796171	0.016127
11	71	0.794844	0.016099
12	91	0.797485	0.015196
13	116	0.795501	0.017009
14	148	0.791876	0.024576
15	189	0.791216	0.018826
16	241	0.781961	0.020935
17	307	0.772056	0.020249
18	392	0.738699	0.029494
19	499	0.658087	0.035785

TRAINING SET



More kNN examples.
Cell Segmentation example [classification]

```
# Predict on test set
test_pred = grid.predict(segDataTest)

# Confusion matrix
cm = confusion_matrix(segClassTest, test_pred)

disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=grid.best_estimator_.named_steps['knn'].classes_)
disp.plot(cmap=plt.cm.Blues)
plt.show()
```



More kNN examples.
Blood/brain barrier example [regression]

Regression

```
[ ]  bloodBrainDf = pd.read_csv("bloodbrainDF.csv")
     bloodBrainDf = bloodBrainDf.drop("Unnamed: 0", axis = 1)
```

	tpsa	nbasic	negative	vsa_hyd	a_aro	weight	peoe_vsa.0	peoe_vsa.1	peoe_vsa.2	peoe_vsa.3	...	ctdh	ctaa	mchg	achg	rdta	n_sp2	n_sp3
0	12.030000	1	0	167.06700	0	156.293	76.94749	43.44619	0.00000	0.000000	...	1	1	0.9241	0.9241	1.0000	0.000000	6.0255
1	49.330002	0	0	92.64243	6	151.165	38.24339	25.52006	0.00000	8.619013	...	2	2	1.2685	1.0420	1.0000	0.000000	6.5681
2	50.529999	1	0	295.16700	15	366.485	58.05473	124.74020	21.65084	8.619013	...	1	4	1.2562	1.2562	0.2500	26.973301	10.8567
3	37.389999	0	0	319.11220	15	382.552	62.23933	124.74020	13.19232	21.785640	...	1	3	1.1962	1.1962	0.3333	21.706499	11.0017
4	37.389999	1	0	299.65800	12	326.464	74.80064	118.04060	33.00190	0.000000	...	1	3	1.2934	1.2934	0.3333	24.206100	10.8109
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
203	32.700001	1	0	233.69200	6	250.362	51.28292	109.50990	13.19232	0.000000	...	1	3	0.8717	0.8717	0.3333	0.000000	3.1150
204	3.240000	0	0	343.24290	10	292.446	40.59702	104.67720	13.19232	0.000000	...	0	1	0.0000	0.0000	0.0000	0.000000	4.6658
205	32.340000	1	0	234.79200	6	261.389	75.11627	86.66676	0.00000	8.619013	...	1	2	1.2276	1.2276	0.5000	0.000000	16.1765
206	37.299999	0	0	124.25940	0	143.206	63.98079	14.70850	0.00000	0.000000	...	1	2	0.9082	0.9082	0.5000	0.000000	0.0000
207	64.430000	0	0	224.75310	10	319.385	74.42332	43.65297	29.68691	6.699551	...	0	4	0.0000	0.0000	0.0000	26.665400	2.0261

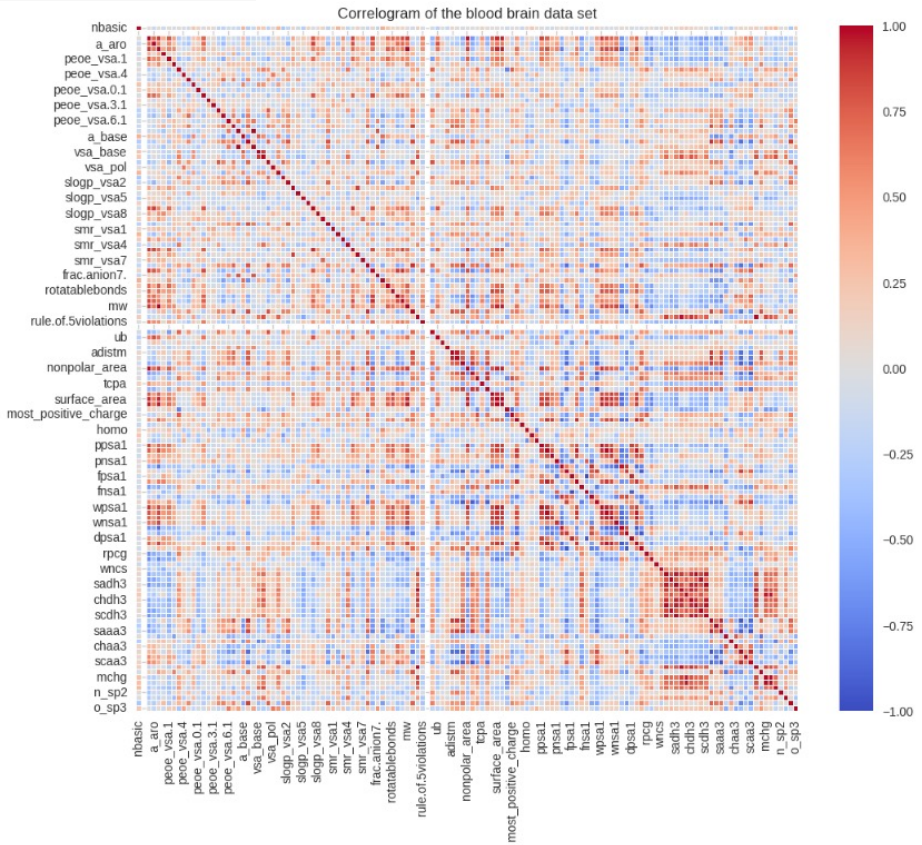
208 rows × 135 columns

More kNN examples.
Blood/brain barrier example [regression]

```
[ ]  bloodBrainClass = bloodBrainDf['logBBB']
     bloodBrainData = bloodBrainDf.iloc[:, 1:-1]

     # Splitting data into training and testing sets
     trainIndex, testIndex = train_test_split(bloodBrainData.index, test_size=0.8, random_state=42)

     bloodBrainDataTrain = bloodBrainData.loc[trainIndex]
     bloodBrainDataTest = bloodBrainData.loc[testIndex]
     bloodBrainClassTrain = bloodBrainClass.loc[trainIndex]
     bloodBrainClassTest = bloodBrainClass.loc[testIndex]
```

```
def find_correlation(data, threshold):
    corr_mat = data.corr()
    corr_mat = abs(corr_mat)
    upper = corr_mat.where(np.triu(np.ones(corr_mat.shape), k=1).astype(np.bool))
    to_drop = [column for column in upper.columns if any(upper[column] > threshold)]
    return to_drop

highCorr = find_correlation(bloodBrainDataTrain, 0.75)

print("Number of highly correlated features: ",len(highCorr))
```

```
Number of highly correlated features:  71
```



More kNN examples.
Blood/brain barrier example [regression]

```
# Get features with near-zero variance
selector = VarianceThreshold(threshold=1e-5)
bloodBrainDataTrain_selected = selector.fit_transform(bloodBrainDataTrain)

retained_columns_mask = selector.get_support()

dropped_columns = bloodBrainDataTrain.columns[~retained_columns_mask]

print("Columns removed due to near-zero variance:")
print(dropped_columns)
```

```
Columns removed due to near-zero variance:
Index(['negative', 'alert', 'tcsa', 'chdh3', 'chaa3'], dtype='object')
```

More kNN examples.
Blood/brain barrier example [regression]

```
from sklearn.base import BaseEstimator, TransformerMixin
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import make_scorer, mean_squared_error, r2_score, mean_absolute_error

tuneParam = {'knn__n_neighbors': list(range(1, 10))}

# I created a custom transformer for removing highly correlated features - couldn't find one in sklearn.
class RemoveCorrelatedFeatures(BaseEstimator, TransformerMixin):
    def __init__(self, threshold=0.9):
        self.threshold = threshold

    def fit(self, X, y=None):
        if not isinstance(X, pd.DataFrame):
            X = pd.DataFrame(X)

        corr_matrix = X.corr().abs()
        upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(np.bool))
        self.to_drop = [column for column in upper.columns if any(upper[column] > self.threshold)]
        return self

    def transform(self, X):
        if not isinstance(X, pd.DataFrame):
            X = pd.DataFrame(X)
        return X.drop(self.to_drop, axis=1)



pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('remove_nzv', VarianceThreshold(threshold=0.01)),
    ('remove_corr', RemoveCorrelatedFeatures(0.9)),
    ('knn', KNeighborsRegressor())
])
```

More kNN examples.
Blood/brain barrier example [regression]

```
pipeline = Pipeline([
    ('scaler', StandardScaler()),
    ('remove_nzv', VarianceThreshold(threshold=0.01)),
    ('remove_corr', RemoveCorrelatedFeatures(0.9)),
    ('knn', KNeighborsRegressor())
])

# Define custom RMSE scorer
def rmse(y_true, y_pred):
    return np.sqrt(mean_squared_error(y_true, y_pred))

rmse_scorer = make_scorer(rmse, greater_is_better=False)

# Define scoring metrics
scoring = {
    'RMSE': rmse_scorer,
    'R2': 'r2',
    'MAE': 'neg_mean_absolute_error'
}

# Modify the GridSearchCV to compute the specified metrics
grid = GridSearchCV(pipeline, param_grid=tuneParam, cv=cv, n_jobs=-1, scoring=scoring, refit='RMSE', return_train_score=True)

grid.fit(bloodBrainDataTrain, bloodBrainClassTrain)

results = pd.DataFrame(grid.cv_results_)

# Extracting RMSE, R2, and MAE for each k value
metrics_summary = results[['param_knn__n_neighbors', 'mean_test_RMSE', 'mean_test_R2', 'mean_test_MAE']]
print(metrics_summary)
```

More kNN examples.
Blood/brain barrier example [regression]

```
# Modify the GridSearchCV to compute the specified metrics
grid = GridSearchCV(pipeline, param_grid=tuneParam, cv=cv, n_jobs=-1, scoring=scoring, refit='RMSE', return_train_score=True)

grid.fit(bloodBrainDataTrain, bloodBrainClassTrain)

results = pd.DataFrame(grid.cv_results_)

# Extracting RMSE, R2, and MAE for each k value
metrics_summary = results[['param_knn__n_neighbors', 'mean_test_RMSE', 'mean_test_R2', 'mean_test_MAE']]
print(metrics_summary)
```

```
  param_knn__n_neighbors  mean_test_RMSE  mean_test_R2  mean_test_MAE
0                      1       -0.914251     -0.916686      -0.726741
1                      2       -0.797283     -0.367570      -0.624412
2                      3       -0.716902     -0.035597      -0.549222
3                      4       -0.686389      0.044503      -0.514361
4                      5       -0.692614      0.053903      -0.521463
5                      6       -0.696653      0.034224      -0.526472
6                      7       -0.690733      0.052964      -0.521163
7                      8       -0.674976      0.104897      -0.515214
8                      9       -0.678923      0.098550      -0.526797
```

More kNN examples.
Blood/brain barrier example [regression]