K-NEAREST NEIGHBOURS

Dr. Brian Mc Ginley

KNN BASICS

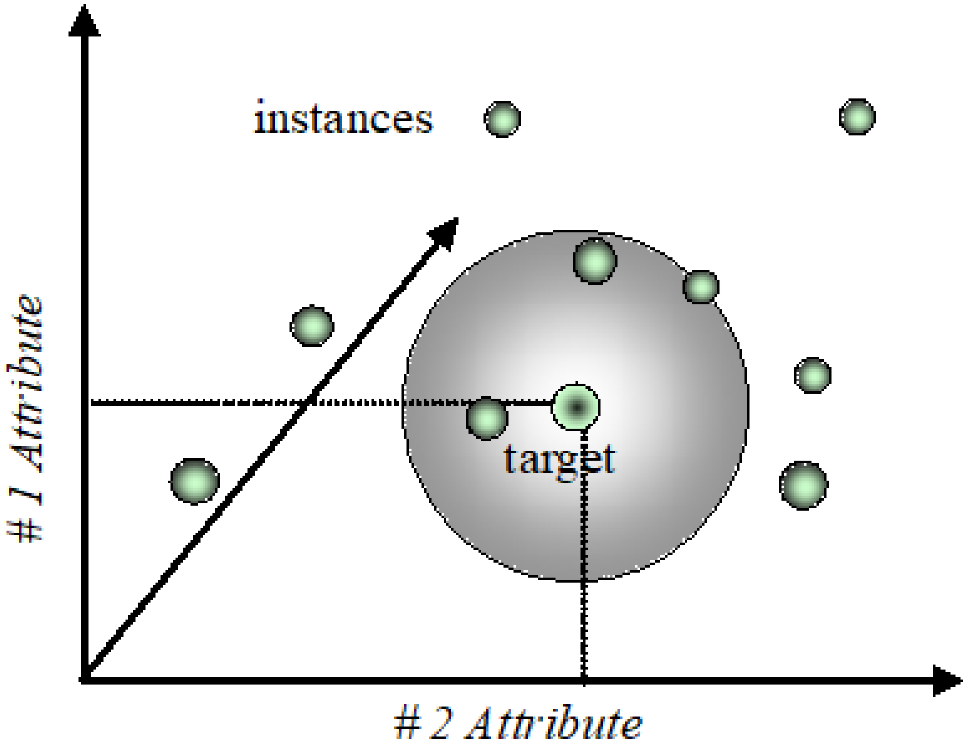
- kNN is basically a two-part algorithm

1. Retrieve the *k* most similar recorded cases
2. Classification or regression
   - "Average" across these *k* cases (regression)
   - Take the max of *k* votes for ordinal types (classification)

- So, kNN can be used for either regression or classification. However, let's ignore regression and stick to classification

DISTANCE

- May require examining all instances



DISTANCE

- We typically need a notion of "distance" to calculate the "nearest" neighbour.
- Often use Euclidean distance i.e. (with p features/predictor values, ***t*** is a target and ***s*** is a source, $t1$ is the first attribute for ***t*** etc.

$$d(t,s) = \sqrt{(t_1 - s_1)^2 + \cdots + (t_p - s_p)^2}$$

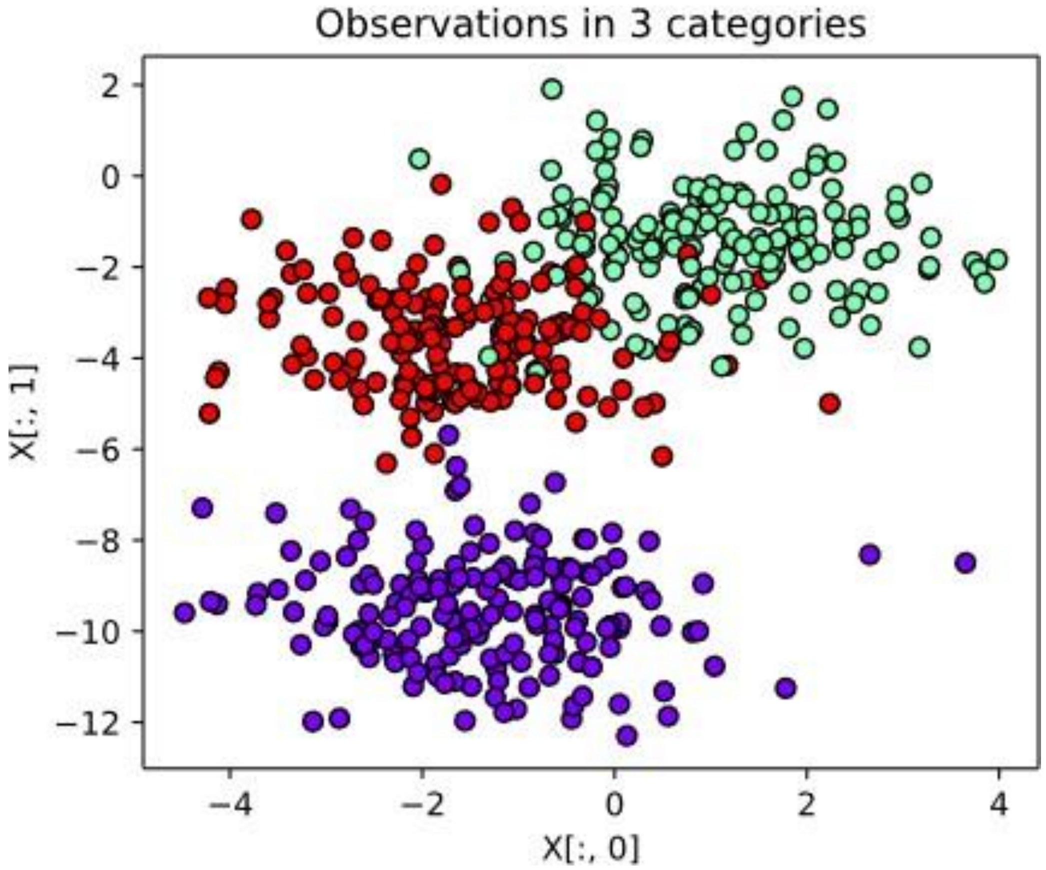
- Caution: Scale matters. If one predictor has is a much larger value than another, it will contribute more to the distance measurement.

FORMALISING EVERYTHING

1. If you have a training set of data $(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_m)$.
2. Given a target $\boldsymbol{x}_j$.
3. Calculate all $d(\boldsymbol{x}_j, \boldsymbol{x}_i)$ and select the k smallest ones.
4. Classify/Approximate $\boldsymbol{x}_j$ based on the k nearest $\boldsymbol{x}_i$ you found.

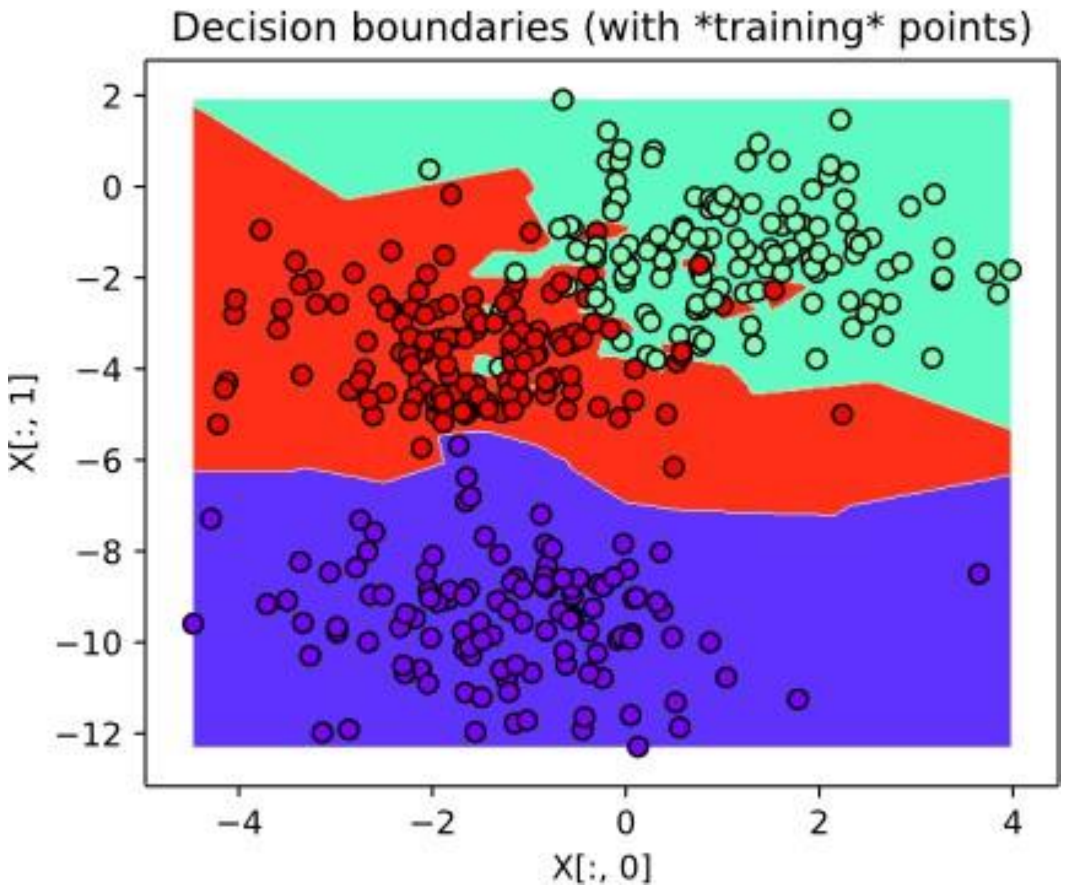
NEAREST NEIGHBOUR

- What if the prediction is made by just looking at which training point is it closest to?



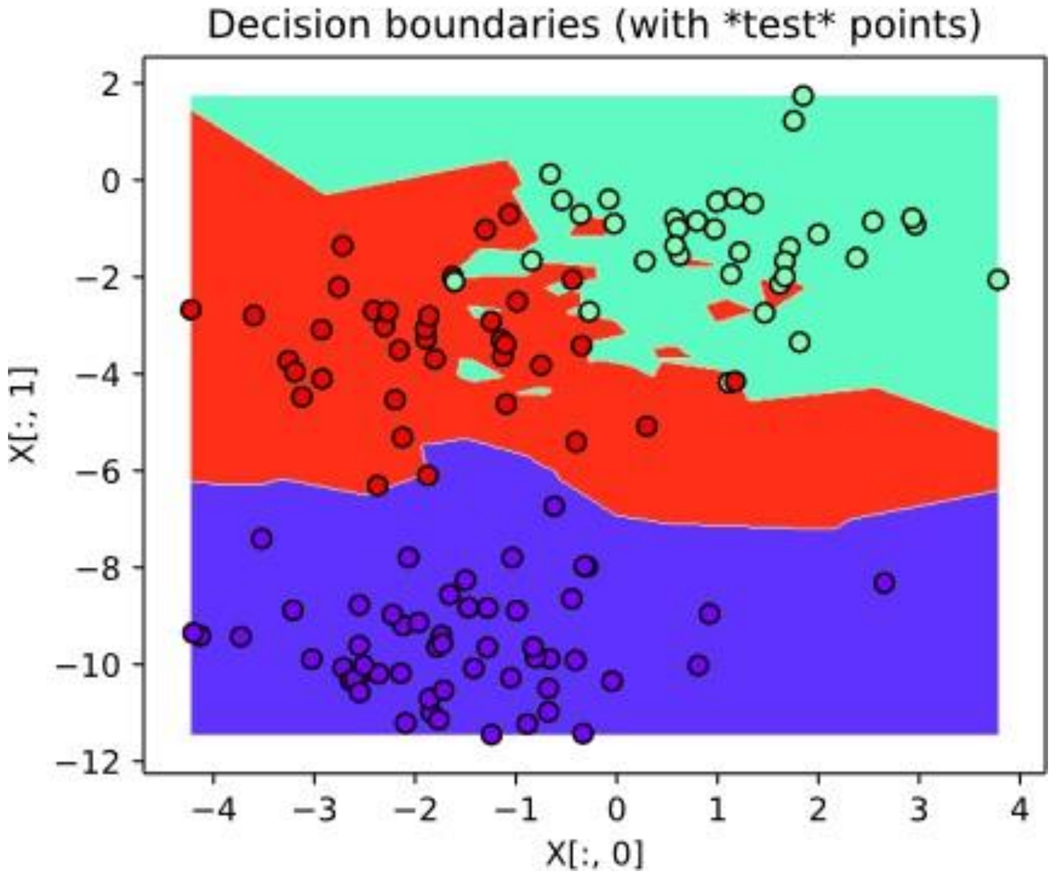
NEAREST NEIGHBOUR

- Not too bad. Looks like overfitting the training data.



NEAREST NEIGHBOUR

- With the test data, we can see noise.

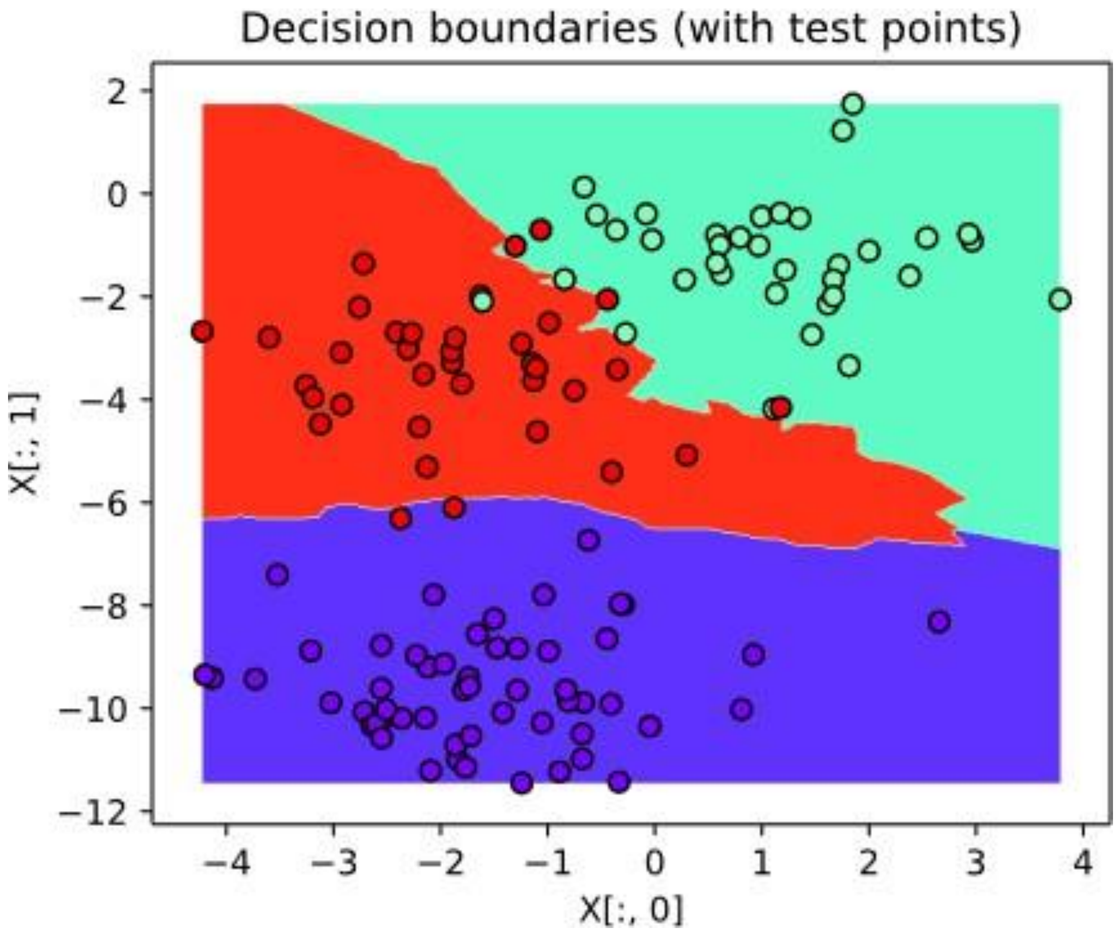


k-NEAREST NEIGHBOUR

- Instead of looking at the one nearest training point, we could look at k and have a vote: a k-nearest neighbours classifier.
- e.g. with $k = 5$, we might want to predict a category for a point and find that the nearest five training points are green, green, red, green, red.
  - Take the most common and predict green.

K-NEAREST NEIGHBOUR

- Smooths out some of the overfitting



SKLEARN

- Of course we want to do this by programming. It follows the exact same procedure as the others - model, fit, evaluate.

```
from sklearn.neighbors import KNeighborsClassifier
model = KNeighborsClassifier(n_neighbors = 5)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))

0.944
```

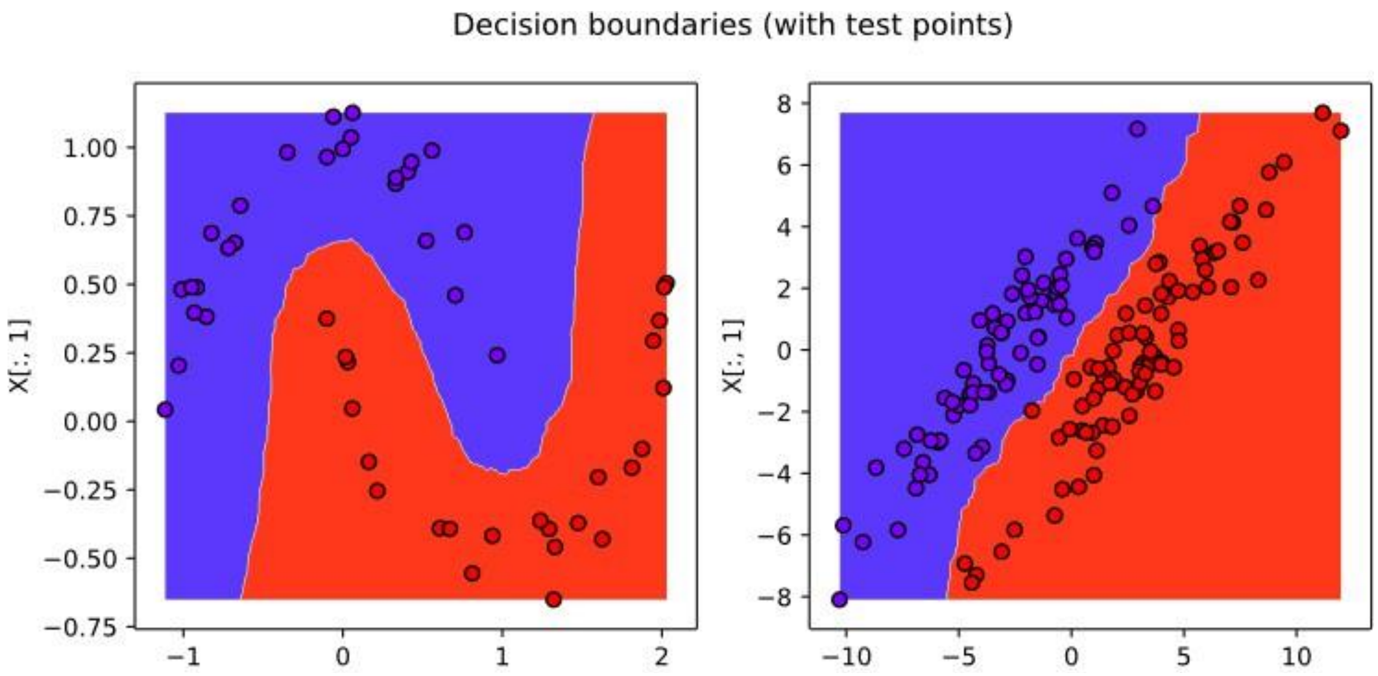
- Offers good results for such a simple technique

CHOOSING *K*

- There's tradeoff. If *k* is too small, you'll overfit the training data. If it's too large, you'll underfit reality.
- Any guesses?
  - Cross-validation. Experiment.
- So you can use *k*-fold cross validation to choose the *k* in *k*-nearest neighbours….this is confusing. They are two different *k*'s!
- We are choosing the hyperparameter *k* for *k*-nearest neighbours and you apply say a 10-fold cross validation procedure to choose it

KNN

- kNN can create a complicated boundary shapes.



CURSE OF DIMENSIONALITY

- Imagine instances described by 20 attributes, but only 2 are relevant to target function.
- Curse of dimensionality: nearest-neighbour is easily mis-led with high-dimensional X.
- High-dimensionality increases sparsity. The more features, the further the distances apart.

KNN OTHER

- There are other ways of applying distance metrics
  - Weighted kNN
  - Distance Weighted kNN (Shepard's Method)
- But we will not discuss them

FURTHER REMARKS ON NEAREST NEIGHBOURS

- Since the calculations are done when classifying a new data point, the model must store all the training information (Lazy).
- Storing all the training information can be quite wasteful, better generalisation models would not have to store the full training set.
- *k*-nearest neighbours is very quick to train (it does not do anything except store the set), however prediction is slow.

LAZY AND EAGER LEARNING

- Two contrasting approaches in machine learning, primarily referring to the handling of model construction and prediction.
  - Lazy learning defers the computation of predictions until needed, relying on storing instance-specific information.
  - Eager learning precomputes a model during training, making predictions faster but potentially requiring more memory.

Aspect	Lazy Learning	Eager Learning
Timing of Model Building	The model is built during prediction.	The model is built before prediction.
Data Dependency	Relies heavily on the training data during prediction.	Less dependent on training data during prediction.
Computational Efficiency	Faster during training, but slower during prediction due to real-time model building.	Slower during training, but faster during prediction due to pre-built model.
Example	k-Nearest Neighbors (KNN)	Decision Trees, Support Vector Machines (SVM), Neural Networks
Memory Usage	Less memory usage during training, but more during prediction.	More memory usage during training, but less during prediction.

FURTHER DIMENSIONS

- It's easy to show examples that are 2D points: they make nice figures and are easy to visualize, but they aren't so realistic.
- Let's look at some more realistic data: there are lots of example datasets out there for ML experimentation: mldata.org, UCI Machine Learning Repository, KDnuggets dataset list.
- Scikit-learn has some datasets built-in (like the diabetes and iris datasets).

SKLEARN DATASETS

```
from sklearn.datasets import load_breast_cancer
bc = load_breast_cancer()
print(bc.data.shape)
print(bc.target.shape, np.unique(bc.target))
print(bc.feature_names)
X = bc.data
y = bc.target
```

```
(569, 30)
(569,)[01]
['mean radius' 'mean texture' 'mean perimeter' 'mean area' 'mean smoothness'
'mean compactness' 'mean concavity' 'mean concave points' 'mean symmetry' 'mean
fractal dimension' 'radius error' 'texture error' 'perimeter error' 'area error'
'smoothness error' 'compactness error' 'concavity error' 'concave points error'
'symmetry error' 'fractal dimension error' 'worst radius' 'worst texture' 'worst
perimeter' 'worst area' 'worst smoothness' 'worst compactness' 'worst concavity'
'worst concave points' 'worst symmetry' 'worst fractal dimension']
```

SKLEARN DATASETS

- Apply a randomly-chosen classifier and it does quite well

```
model = GaussianNB()
model.fit(X_train, y_train)
print(model.score(X_test, y_test))
print(classification_report(y_test, model.predict(X_test)))

0.944055944056
             precision    recall  f1-score   support

          0       0.94      0.91      0.93        55
          1       0.94      0.97      0.96        88

avg / total       0.94      0.94      0.94       143
```

- Although the 6% inaccuracy may be too much for this type of information. Ignore this and just think of modelling.

SKLEARN DATASETS - KNN

- kNN seems to do worse

```
model = KNeighborsClassifier(n_neighbors=9)
model.fit(X_train, y_train)
print(model.score(X_test, y_test))

0.923076923077
```

- Any ideas why?
- Ask questions about the dataset - how it fits into the model and how the model reacts. Remember I said about distance metrics being heavily influenced by larger values, so we may need some form of scaling.

SKLEARN DATASETS - KNN

- The scales are completely different. Measuring "nearest" counts "mean radius" as much more important than "worst concavity".

```
bc_df = pd.DataFrame(bc.data, columns=bc.feature_names)
print(bc_df[['mean radius', 'texture error', 'worst concavity']].
    describe())

       mean radius  texture error  worst concavity
count   569.000000     569.000000       569.000000
mean     14.127292       1.216853         0.272188
std       3.524049       0.551648         0.208624
min       6.981000       0.360200         0.000000
25%      11.700000       0.833900         0.114500
50%      13.370000       1.108000         0.226700
75%      15.780000       1.474000         0.382900
max      28.110000       4.885000         1.252000
```