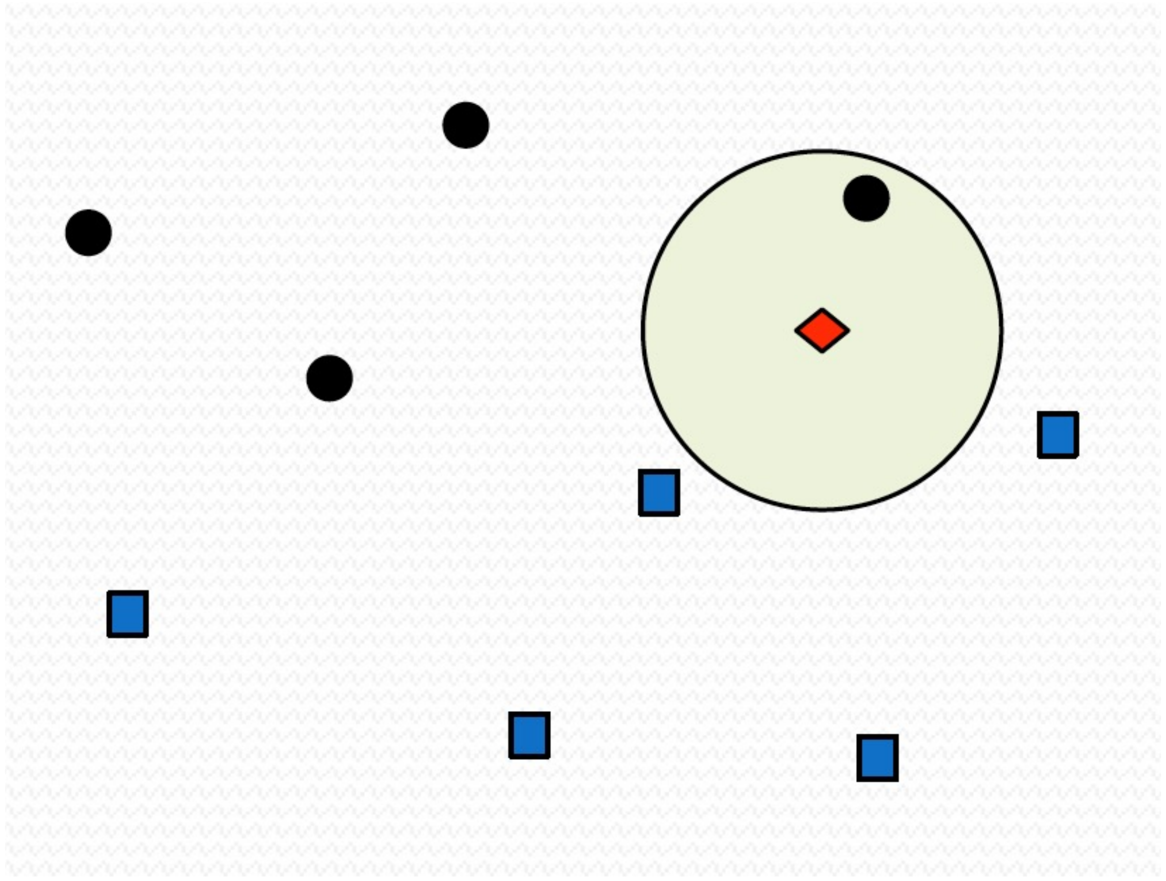
# *k-Nearest Neighbors (k-NN)*

## Instance-Based Learning

- Knn works like a classifier in supervised mode.
  - Have training examples: $(x_i, y_i)$, i=1, …, $N$
    - $x_i$ could have discrete or real value
  - Try to predict the class for new example $x$
    - $y=f(x)\in\{C_1, \cdots, C_c\}$
- The main idea to determine the class
  - Similar examples have similar label
  - Algorithm:
    1. Find most similar training examples $x_n$
    2. Classify $x$ "like" these most similar examples
- Questions:
  - How to determine similarity?
  - How many similar training examples to consider?
  - How to resolve in consistencies among the training examples?

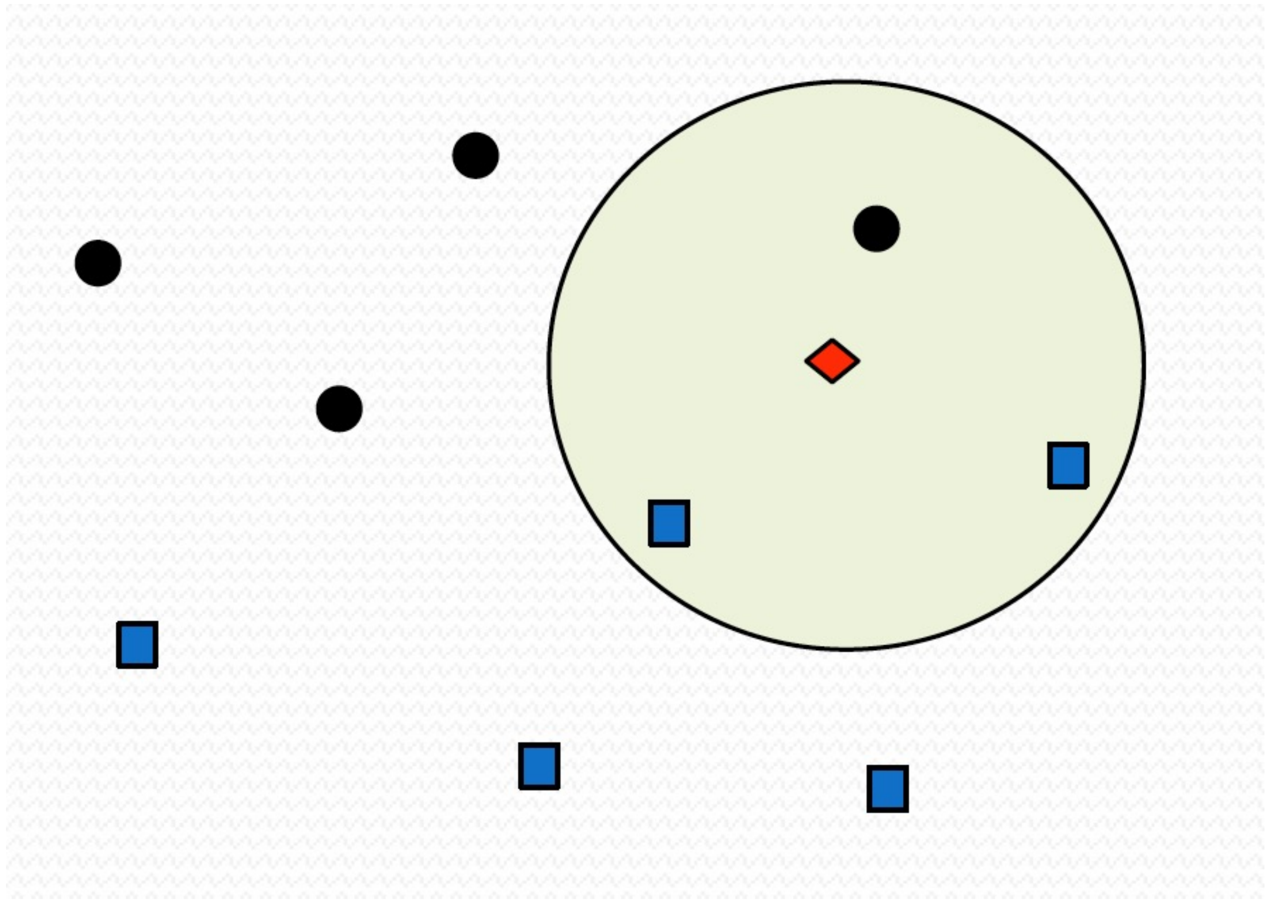
## 1-Nearest Neighbor

- One of the simplest of all machine learning classifiers
- Simple idea: label a new point the same as the closest known point



## 3-Nearest Neighbors

- Generalizes 1-NN to smooth away noise in the labels
- A new point is now assigned the most frequent label of its k nearest neighbors



# K-Nearest Neighbors (KNN)

- K-Nearest neighbour:
  - Given a query instance $x$,
  - First locate the k nearest training examples $x_1, x_2, \ldots, x_k$

  - Classification:
    - Discrete values target function
    - Take vote among its $k$ nearest neighbors
  - Regression
    - Real valued target function
    - Take the mean of the f values of the $k$ nearest neighbors

- Remember. We have to answer to:
  1. How to determine similarity?
  2. How many similar training examples to consider?
  3. How to resolve in consistencies among the training examples?

## 1. How to determine similarity?

It is possible to use any function that respects the following principles

- It's from 'distance properties'
  - Non-negative: $d(i, j) > 0$
  - $d(i, i) = 0$
  - Symmetry: $d(i, j) = d(j, i)$
  - Triangle inequality: $d(i, k) \leq d(i, j)+d(j, k)$

- Some distance
  - Euclidian distance: $d(\mathrm{x}, \mathrm{y}) = \sqrt{\sum(x_i - yi)^2}$
  - Manhattan distance ("city-block"): $d(\mathrm{x}, \mathrm{y}) = \sum|x_i - yi|$

  - Uniform or weighted distance
    - Weigted: assign weights to the neighbors based on their "distance" from the query point
      - Generally weight $= {}^{1}/_{distance}$

# Knn need to normalize each feature

- The distance measure is influenced by the units of the different variables, especially if there is a wide variation in units.
  - Variables with "larger" units will influence the distances more than others.
  - $di, j = \sqrt{\sum(xi - xj)2}$
- An example

| | Income in $ | Age |
|---|---:|---:|
| Carry | $31 779 | 36 |
| Sam | $32 739 | 40 |
| Miranda | $33 880 | 38 |

- d(Carry, Sam) = $((31779 - 32739)^2 + (36 - 40)^2)^{1/2}$
  **= $((960)^2 + (4)^2)^{1/2}$ = $(921600 + 16)^{1/2}$ = 960,008**
  **± difference of income**
- In order to take into account all the features, the dataset must be standardized.

## Knn need to normalize each feature

| | Income in $ | Age | Normalized income | Normalized Age |
|---|---|---|---|---|
| Carry | $31 779 | 36 | 0 | 0 |
| Sam | $32 739 | 40 | 0,46 | 1 |
| Miranda | $33 880 | 38 | 1 | 0,5 |

**With un-normalized features**

| | distance | rank |
|---|---|---|
| d(Carry,Sam) | **960** | **1** |
| d(Sam,Miranda) | 1 141 | 2 |
| d(Miranda,Carry) | 2 101 | 3 |

**With normalized features**

| | distance | rank |
|---|---|---|
| d(Carry,Sam) | 1,1 | **3** |
| d(Sam,Miranda) | **0,73** | **1** |
| d(Miranda,Carry) | 1,12 | 2 |

## 2. How many similar training examples to consider?

Selecting the Number of Neighbors

- Increase $k$:
  - Makes KNN less sensitive to noise
- Decrease $k$:
  - Allows capturing finer structure of space
- Hard to tune!

## 3. How to resolve in consistencies among the training examples?

- Try to use more neighbours
- But give less weight to the far neighbours compared to the close neighbours

- Hard to tune to!

# K-Nearest Neighbors in python

- from sklearn.neighbors import KNeighborsClassifier
  - 3 main parameters
    - Choose the neighbors: n_neighbors (k)
    - Choose the distance: p (power): $(\sum|a_i - bi|^p)^{1/p}$ for Minskowski distance
      - p==1: Manhattan
      - p==2: Euclidian
    - Choose the proximity weight
      - with weight ('distance') or without ('uniform')

- clf = KNeighborsClassifier(n_neighbors=5, weights='uniform', p=2)

- clf.fit(X_train, y_train)
- y_pred = clf.predict(X_test) or clf.predict_proba(X_test)

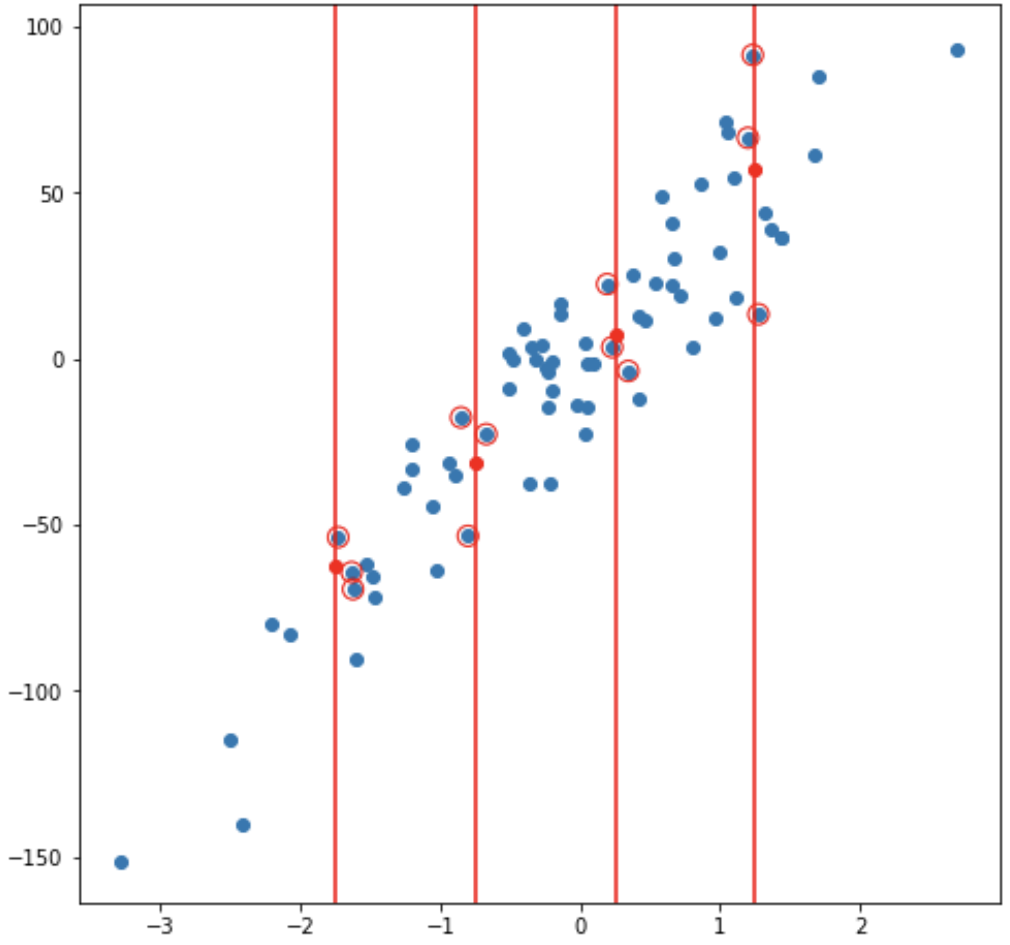
# K-Nearest Neighbors in python

Other parameters

- **Weights** *{'uniform', 'distance'} or callable, default='uniform'*
  - weight function used in prediction.
- **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.
- **leaf_size**, *default=30*
  - Leaf size passed to BallTree or KDTree.
- **metric** *{str or callable}, default='minkowski'*
  - the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric.
- **metric_params**, *default=None*
  - Additional keyword arguments for the metric function.
- **n_jobs,** *default=None*
  - The number of parallel jobs to run for neighbors search.

# Regression with k-NN

- Exactly the same approach
- use the neighbor label value to calculate the value of a new point



- from sklearn.neighbors import KNeighborsRegressor

## PRO of k-NN

- Highly efficient inductive inference method for noisy training data and complex target functions
- Learning is very simple
- k-NN is simple to understand and implement
- k-NN has no assumptions other than the need to standardize features.
- No training step: each new entry is labelled according to these neighbours
- It is possible to enrich the model with run-of-river data.
- No specific work to do to go from a problem with 2 classes, multiclasses or regression
- A very wide variety of distances can be chosen (although we mainly looked at Minkowski)

- **It's an excellent algorithm for replacing missing values...**

## CONS of k-NN

- Need a distance that "matches" the target function, possibly the distance depends on the feature
- k-NN must read the whole dataset for each prediction. Very expensive for large datasets
- **k-NN works well with a small number of features**
  - but the accuracy degrades as the number increases.
- **k-NN works well with a properly balanced dataset**
- **Need to standardize the data to give equal weight to each feature**
- **k-NN doesn't work with missing value**
- **k-NN is very sensitive to outliers** because it simply chooses neighbors based on distance criteria.

- But one of the main problems with k-NN is to choose the optimal number of neighbors to be considered when classifying the new data entry.

## Today lab.

- Part I. K-nearest neighbors for classification
- Part II. K-nearest neighbors for regression

Read and understand the code

- Part III. K-nearest neighbors from scratch

Read the code later

- Part IV. Your work
  1. Impute missing and build knn model
     - Plot confusion matrix
     - Print classification report
     - find the previous values from the confusion matrix (put the formulas in a commented cell)

Put comments on your notebook

Lab part

- Part V. Papers reading
  - Try to understand ANN (Approximate Nearest Neighbors)

Read at least the first 2 papers before the next class