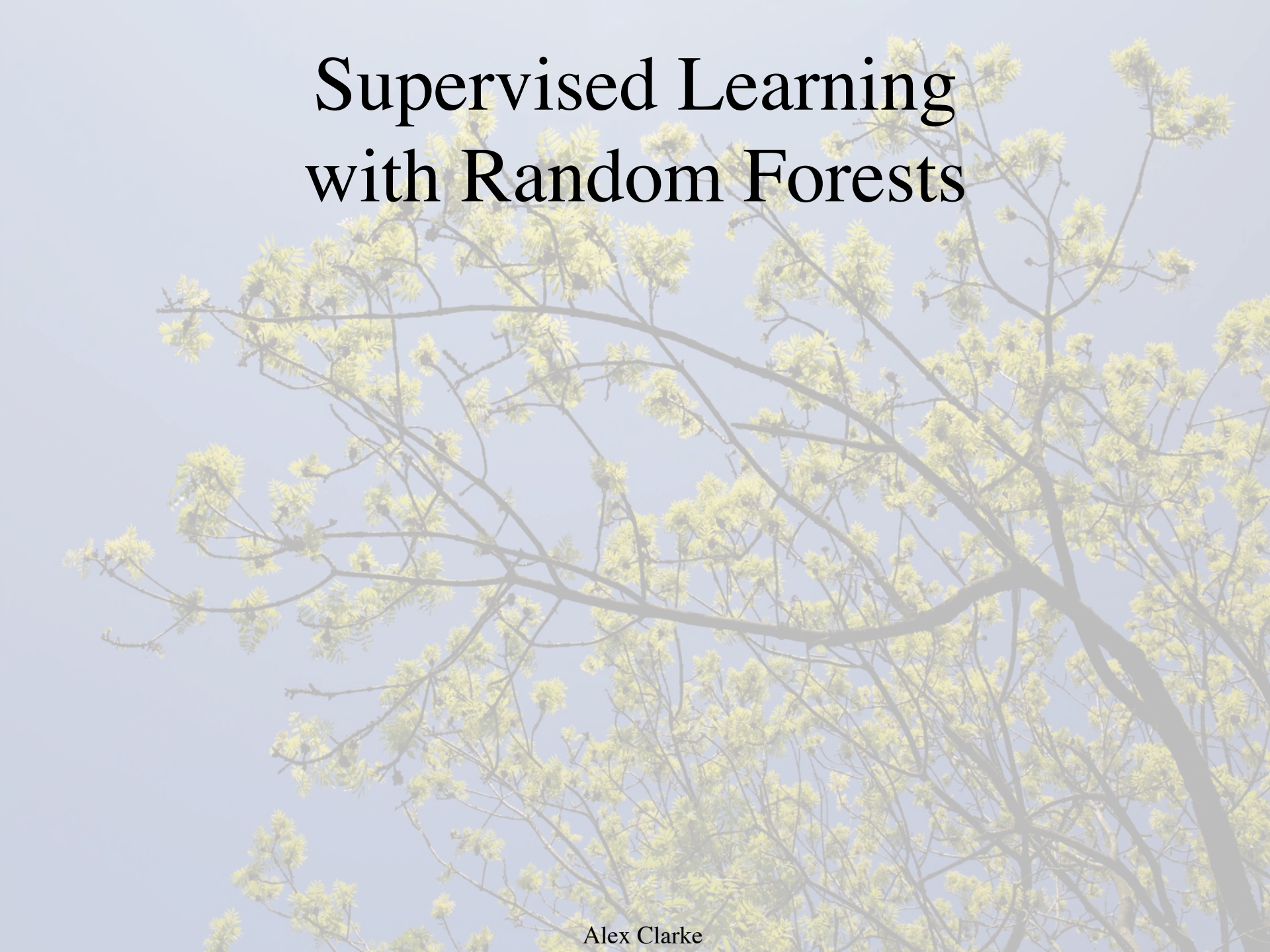
Supervised Learning with Random Forests



Alex Clarke

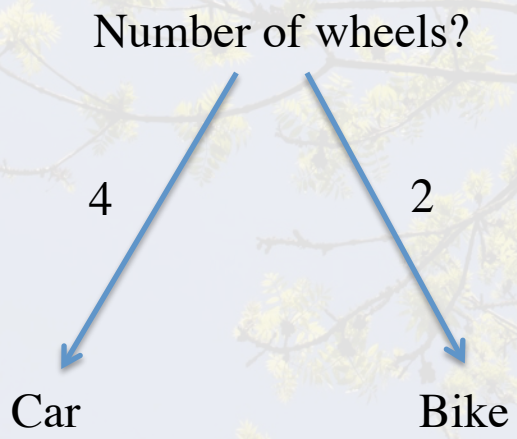
Classification – what is this thing?

Decision trees

Classes: Car or Bike

Features: Number of wheels

Question: How many wheels does it have?



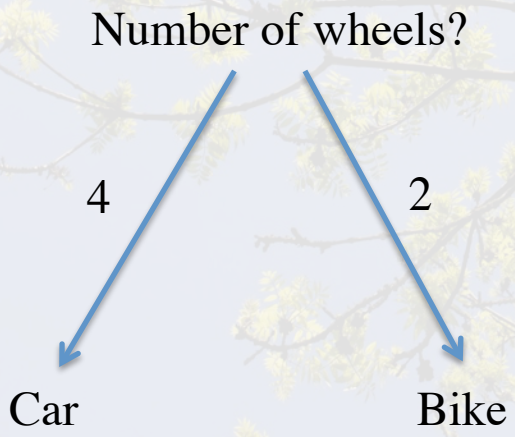
Classification – what is this thing?

Decision trees

Classes: Car or Bike

Features: Number of wheels

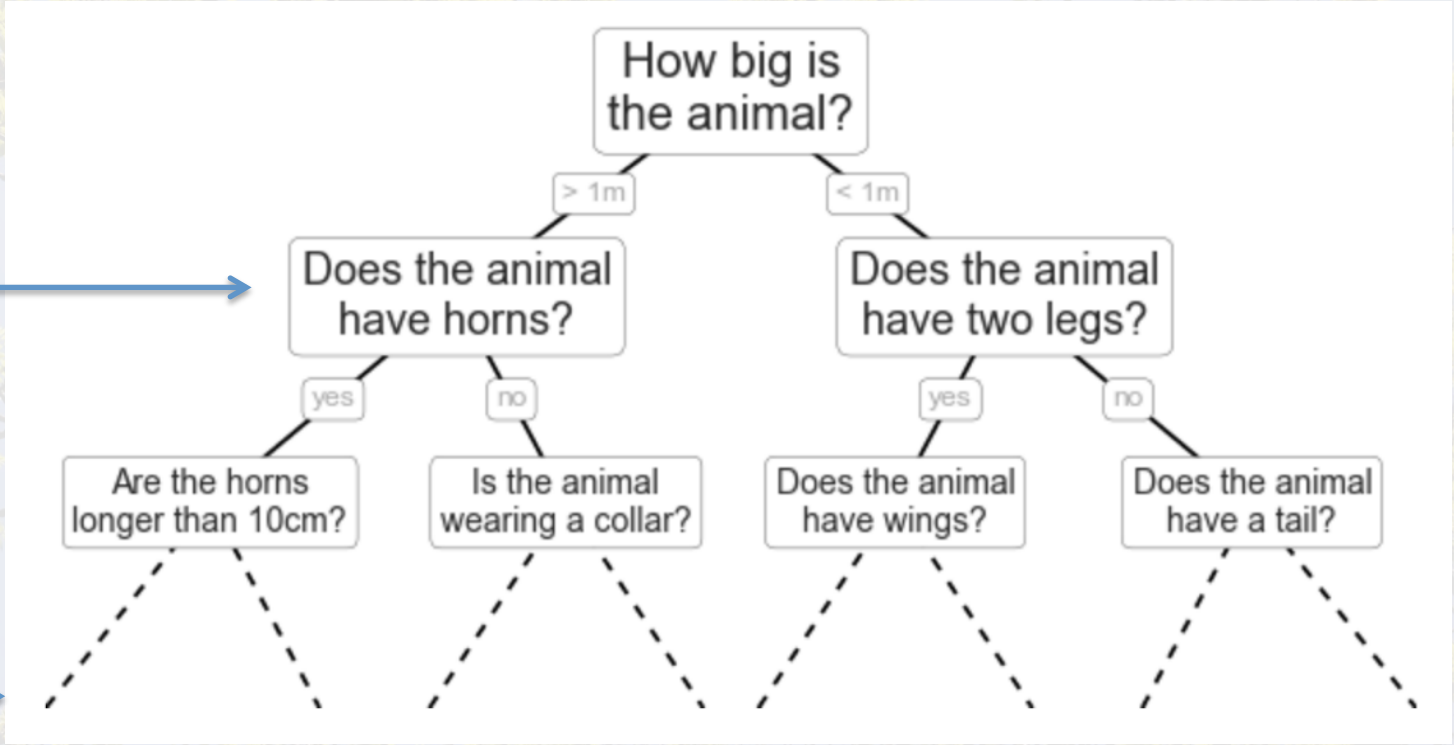
Question: How many wheels does it have?



Classification – what is this thing?

Decision trees

features



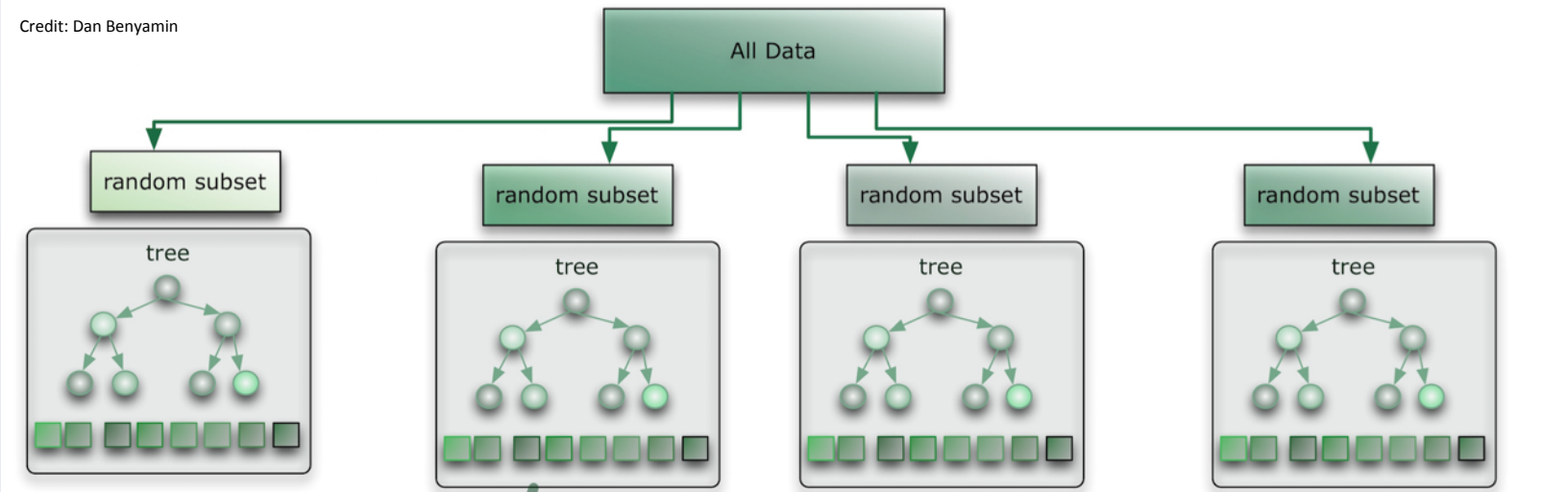
Classes are discrete

Requires training data such that the algorithm can map decisions to known output classes

Random Forests

Combine many decision trees into a single model

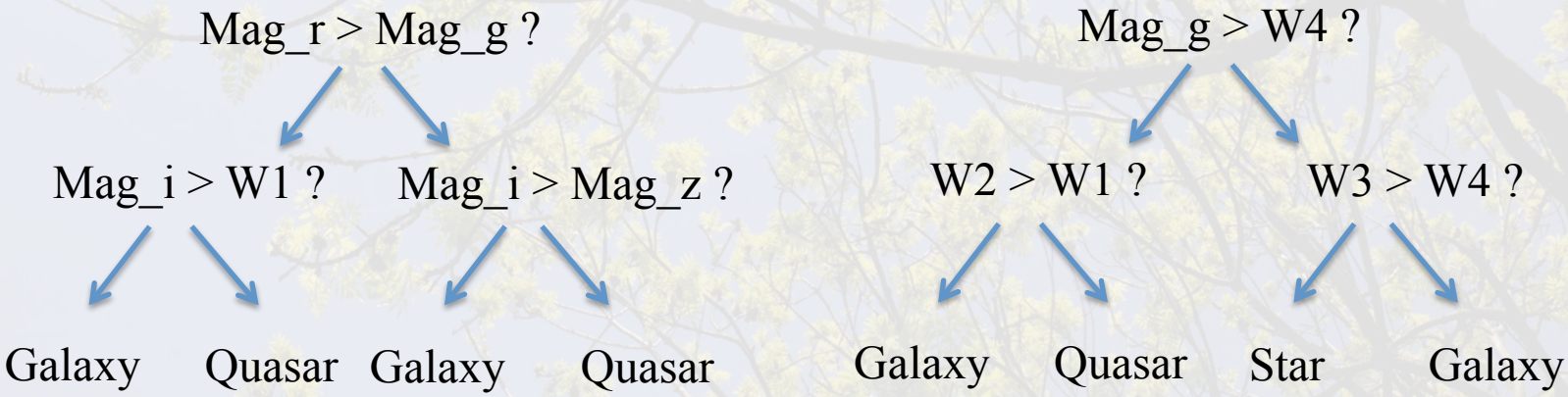
- Each decision tree in the forest considers a random subset of features and data (so is a biased estimator)
- They each capture different trends in the data
- At the end take majority vote from all decision trees for the predicted class



2.5 million sources with photometry points (optical/IR/radio)

Class	Mag_u	Mag_g	Mag_r	Mag_i	Mag_z	w1	w2	w3	w4
Star	21.2	19.8	19.1	18.7	18.3	14.3	14.1	10.7	8.9
Galaxy	21	17	24	23.2	18.5	15.1	15.2	9	8
Quasar	20.9	19.4	21.4	17.1	21.1	16.4	15.7	10.1	9.9
…	…	…	…	…	…	…	…	…	…

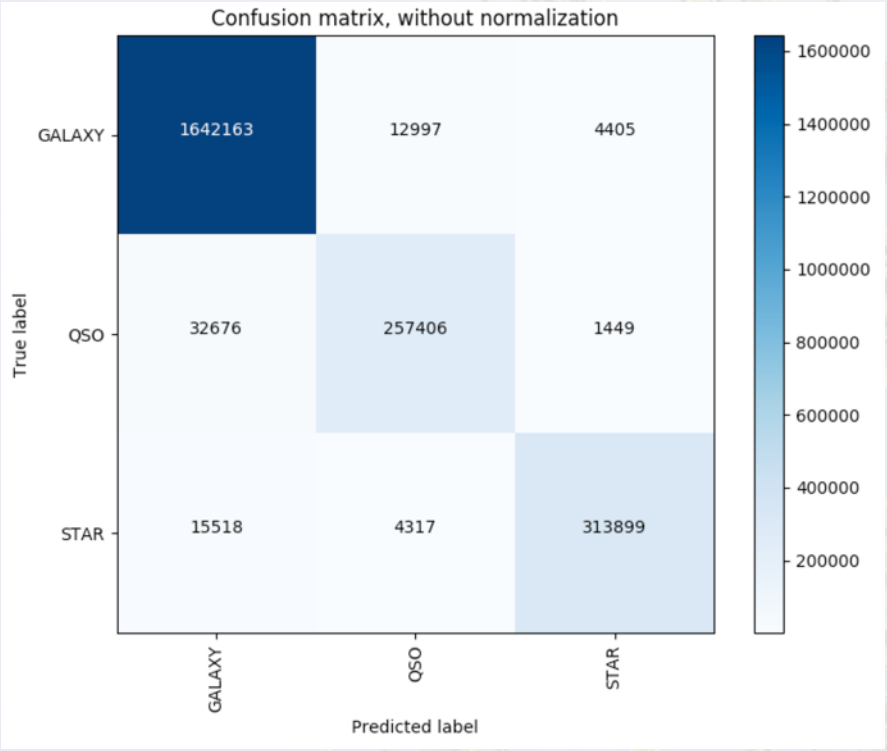
Lots of trees working on random subsets of features and samples

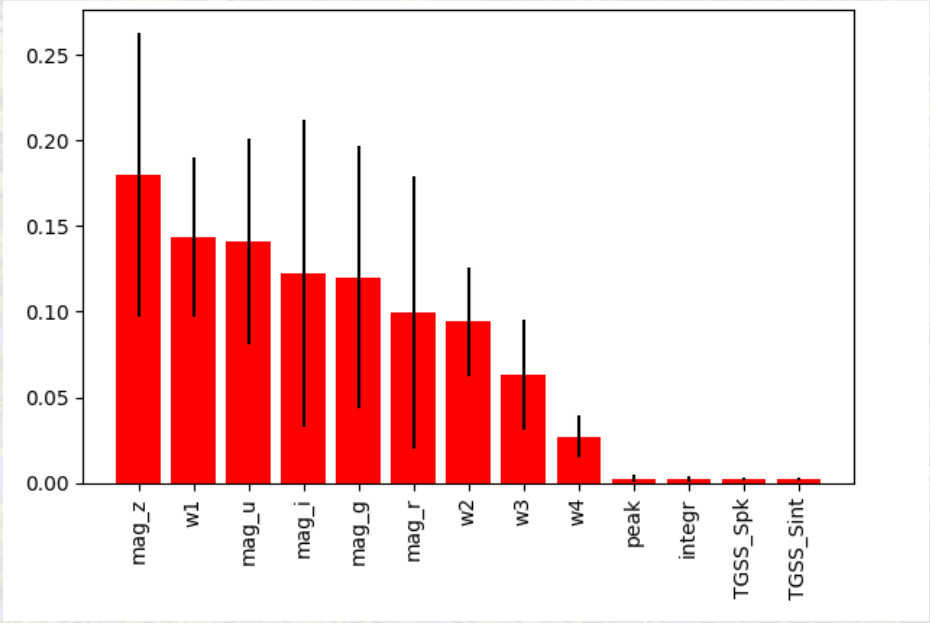


Ensemble of trees decides the most likely class at the end

2.5 million sources with 13 photometry points (optical/IR/radio)
Trained on 250k samples, classifies remaining 2.25 million with 97% accuracy*

**Also 0.97 precision and recall*





More info at: github.com/informationcake/AENEAS

Python implementation

Load in libraries from sklearn:

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.metrics import accuracy_score
```

Load data in:

```
>>> all_features = [[mag_u, mag_g, …], [mag_u, mag_g, …], …]  #photometry data
>>> all_classes = [Galaxy,Star, …] #class labels
```

split your data set up into two parts, one for training, one for testing:

```
>>> features_train, features_test, classes_train, classes_test = train_test_split(all_features,
all_classes, test_size=0.5)
```

Initialise classifier (n_estimators is number of trees):

```
>>> forest = RandomForestClassifier(n_estimators=100)
```

fit to the train data:

```
>>> forest = forest.fit(features_train, classes_train)
```

Predict classes from your test data

```
>>> classes_predicted= forest.predict(features_test)
```

Check accuracy:

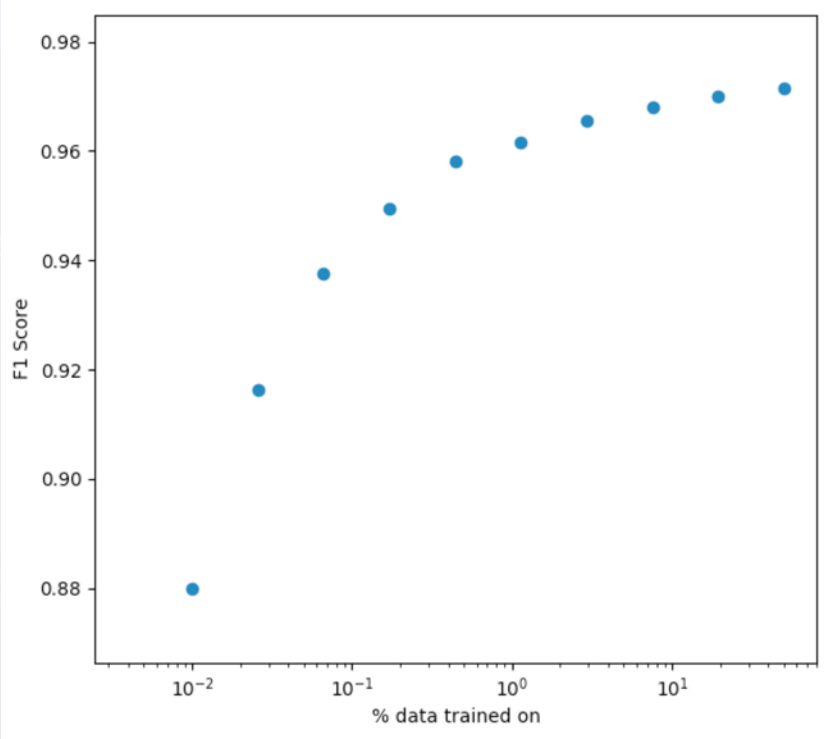
```
>>> accuracy  = accuracy_score(classes_test, classes_predicted)
```

Tuning hyper parameters

2.5 million sources with 13 photometry points (optical/IR/radio)

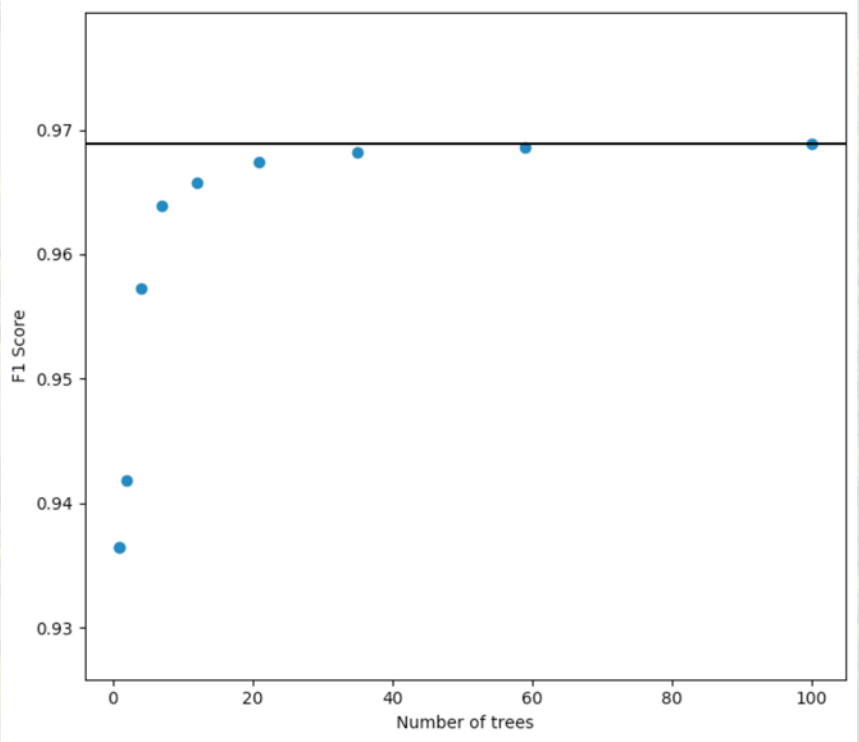
How much data needed to train on?

Depends on the complexity of the data and classifications



Number of trees (estimators)?

The more features you have the more you'll need



Questions about Random Forests?

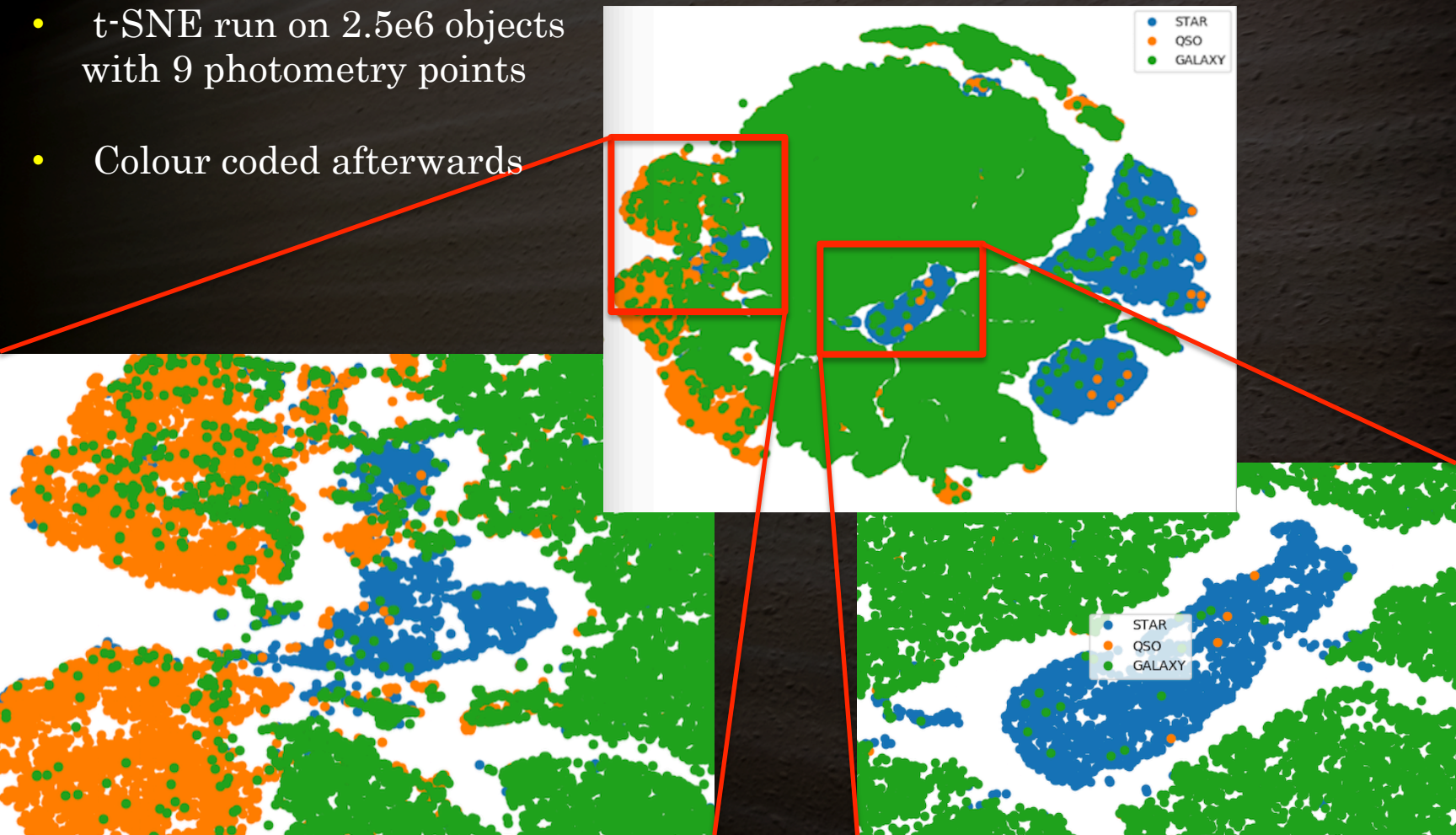
t-distributed Stochastic Neighbour Embedding

t-distributed Stochastic Neighbour Embedding

- t-SNE is great for embedding high-dimensional data in 2/3D for visualisation, and discovering clustering in data sets
- It's non linear
- Can be slow on large data sets because it's an iterative algorithm
- Has a cost function that is not convex - different initializations give different results.
- Requires a level of interaction to interpret results

t-distributed Stochastic Neighbour Embedding

- t-SNE run on 2.5e6 objects with 9 photometry points
- Colour coded afterwards



t-distributed Stochastic Neighbour Embedding

Wikipedia says…

Stage 1:

Construct a probability distribution over pairs of high-dimensional objects in such a way that similar objects have a high probability of being picked, whilst dissimilar points have an extremely small probability of being picked.

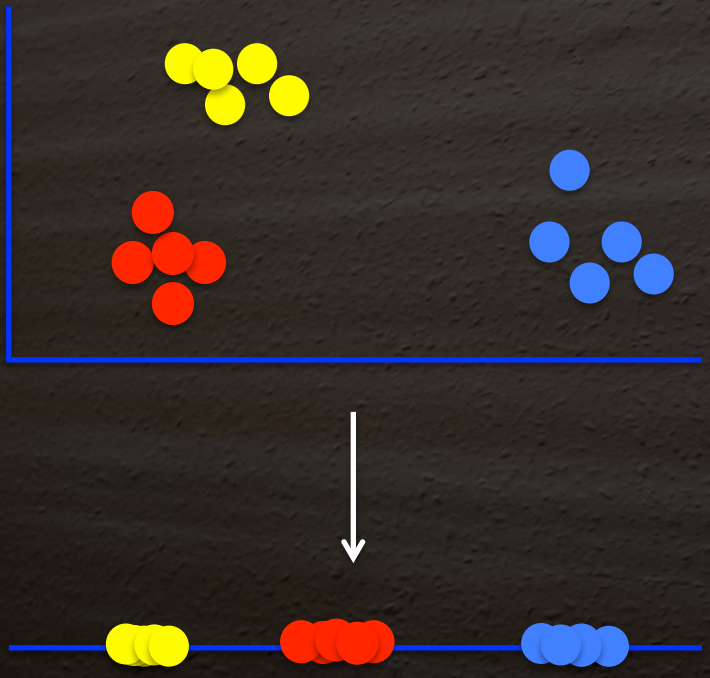
Stage 2:

Define a similar probability distribution over the points in the 2D map, and minimizes the Kullback–Leibler divergence between the two distributions with respect to the locations of the points in the map.

Breaking this down…

t-distributed Stochastic Neighbour Embedding

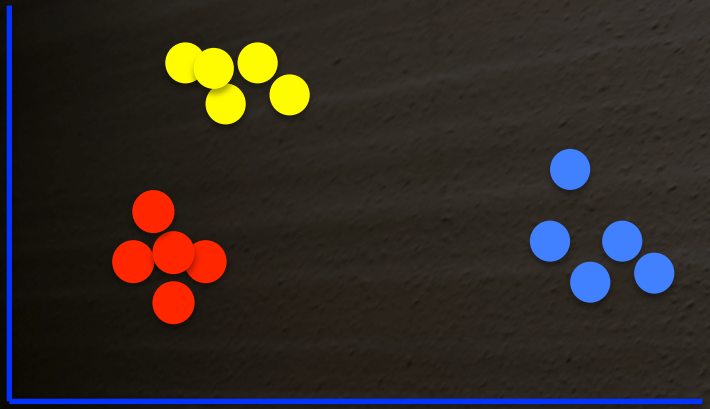
Simple 2D to 1D example



Note that this can't be achieved by projecting onto one axis

t-distributed Stochastic Neighbour Embedding

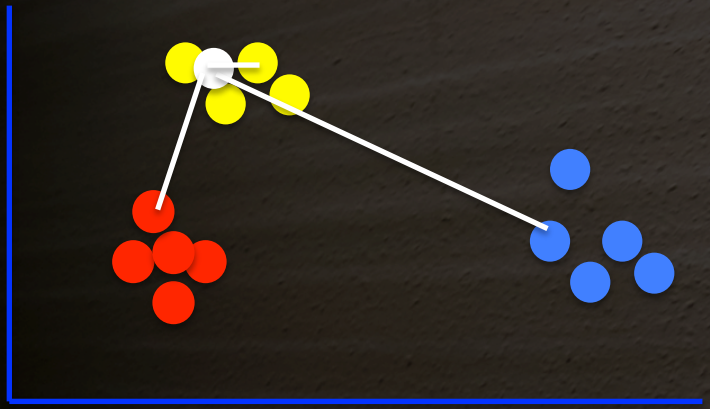
Simple 2D to 1D example

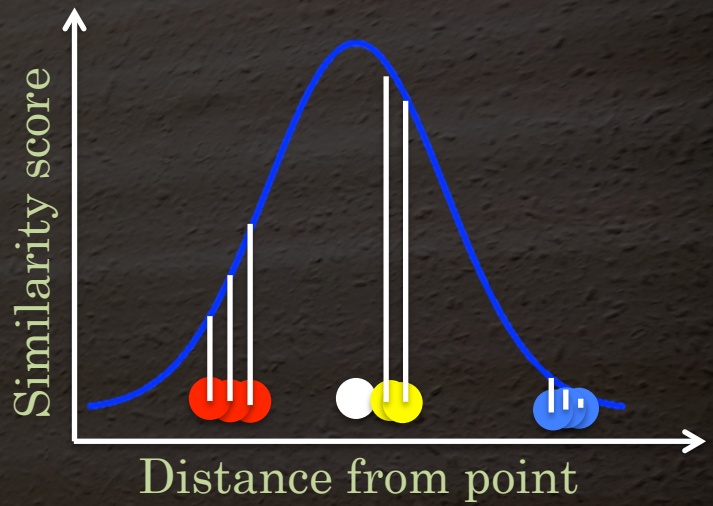


First find out how similar
the points are in 2D

t-distributed Stochastic Neighbour Embedding

Simple 2D to 1D example

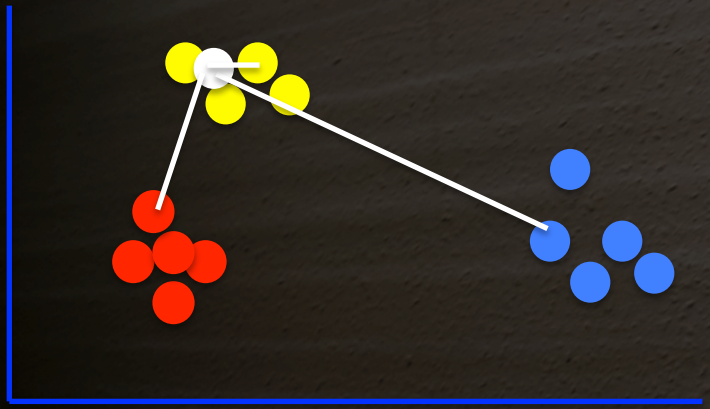


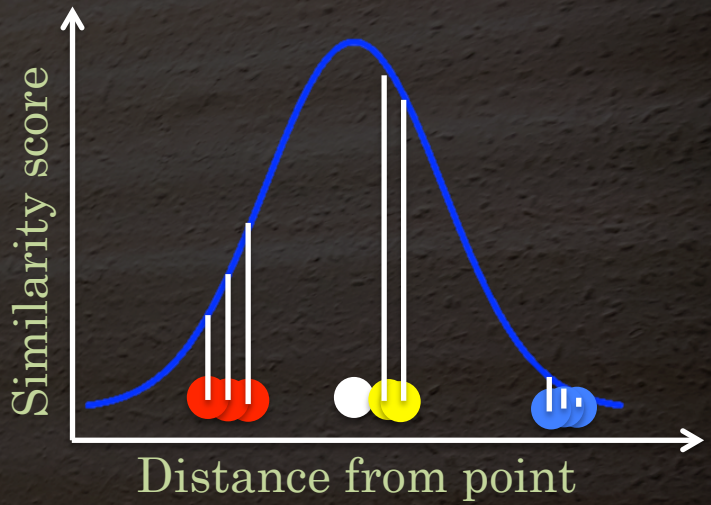


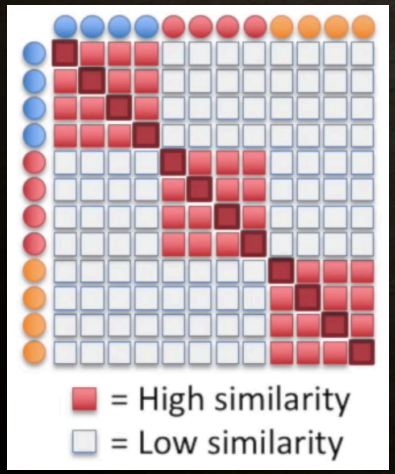
- Centre a Gaussian distribution on each point to obtain similarity scores for each pair of points

t-distributed Stochastic Neighbour Embedding

Simple 2D to 1D example







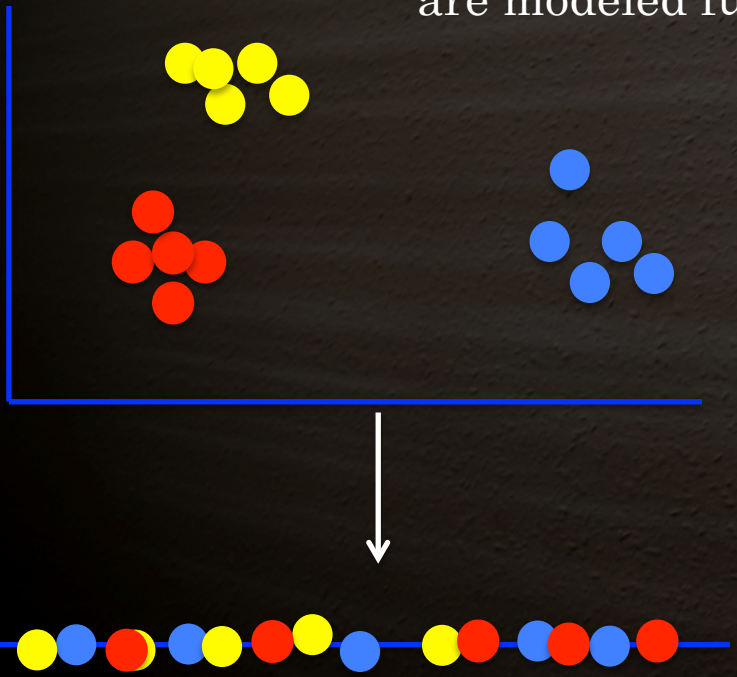
- Centre a Gaussian distribution on each point to obtain similarity scores for each pair of points
- Obtain a matrix of scores for all points
- Note that a hyper-parameter called perplexity can tune the scaling of dense or sparse clusters

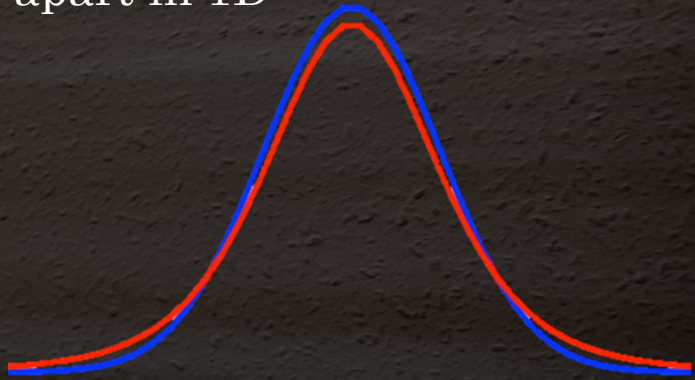
t-distributed Stochastic Neighbour Embedding

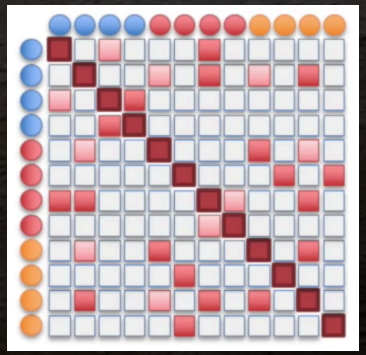
Simple 2D to 1D example

Randomly project data from 2D into 1D, and calculate similarity scores in 1D

But this time use a t-distribution, not a Gaussian, so that dissimilar objects are modeled further apart in 1D



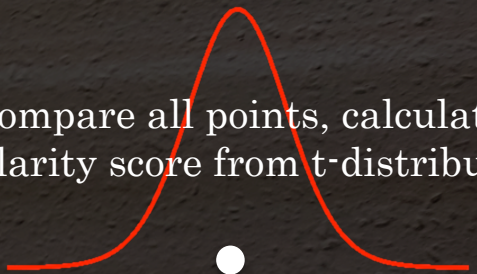


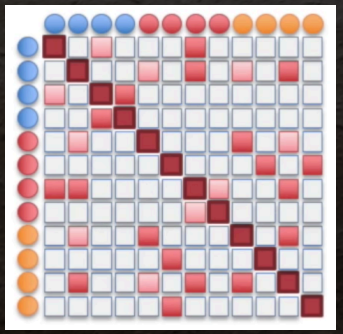


t-distributed Stochastic Neighbour Embedding

Simple 2D to 1D example

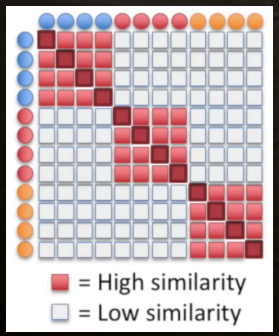
Compare all points, calculate similarity score from t-distribution





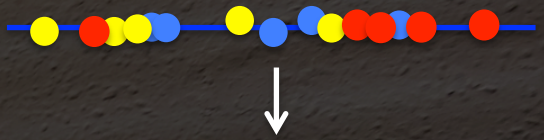
Compare to original distribution in 2D

Points are nudged around slowly in 1D space – attracted to similar points, repelled by dissimilar points

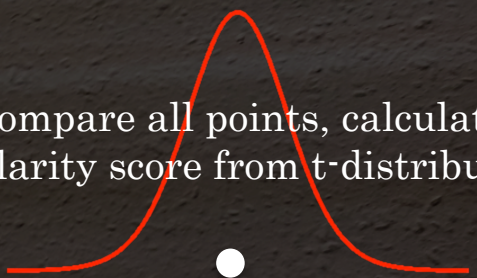


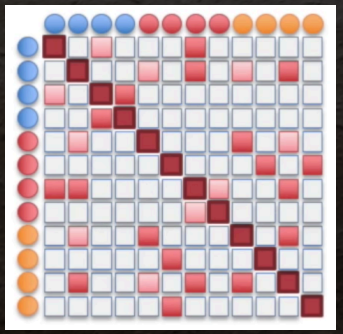
t-distributed Stochastic Neighbour Embedding

Simple 2D to 1D example

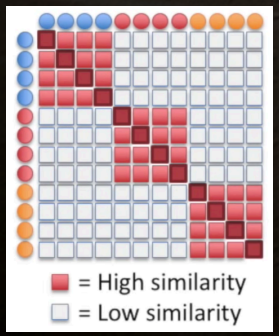


Compare all points, calculate similarity score from t-distribution





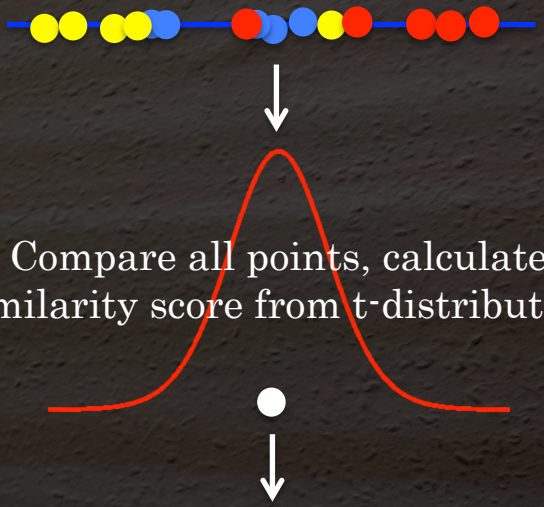
Compare to original distribution in 2D



Points are nudged around slowly in 1D space – attracted to similar points, repelled by dissimilar points

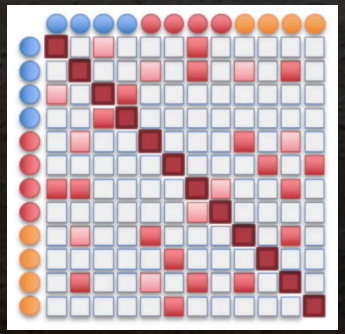
t-distributed Stochastic Neighbour Embedding

Simple 2D to 1D example

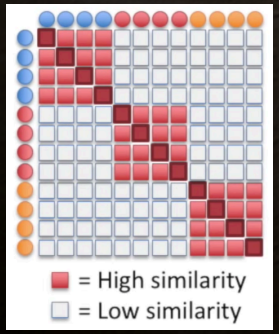


Compare all points, calculate similarity score from t-distribution

Points are nudged around slowly in 1D space – attracted to similar points, repelled by dissimilar points

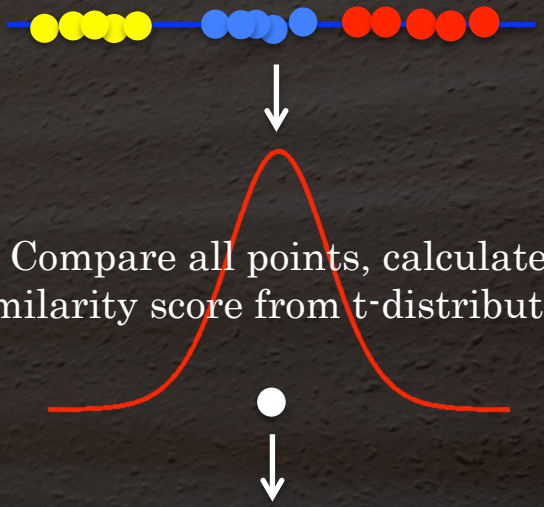


Compare to original distribution in 2D



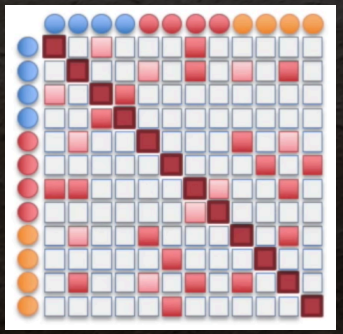
t-distributed Stochastic Neighbour Embedding

Simple 2D to 1D example

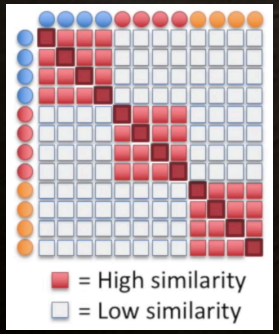


Compare all points, calculate similarity score from t-distribution

Points are nudged around slowly in 1D space – attracted to similar points, repelled by dissimilar points



Compare to original distribution in 2D



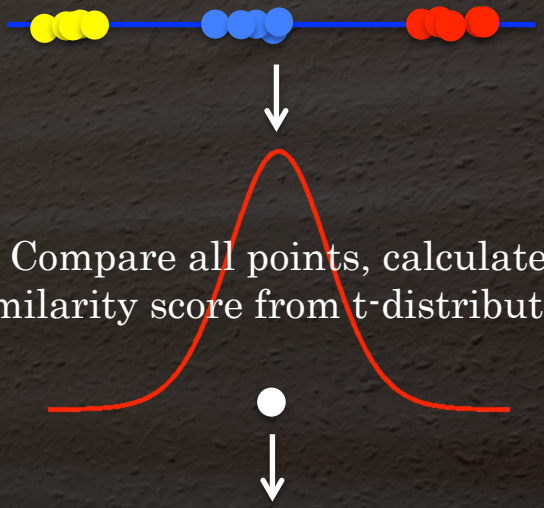
t-distributed Stochastic Neighbour Embedding

Simple 2D to 1D example

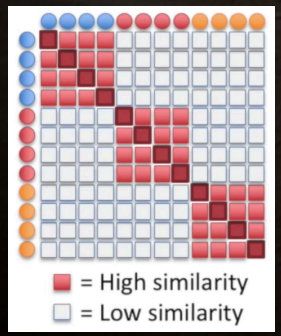
position — gradient decent — 'momentum term'

$$\mathcal{Y}^{(t)} = \mathcal{Y}^{(t-1)} + \eta\frac{\delta C}{\delta \mathcal{Y}} + \alpha(t)\left(\mathcal{Y}^{(t-1)} - \mathcal{Y}^{(t-2)}\right)$$

Points are nudged around slowly in 1D space – attracted to similar points, repelled by dissimilar points

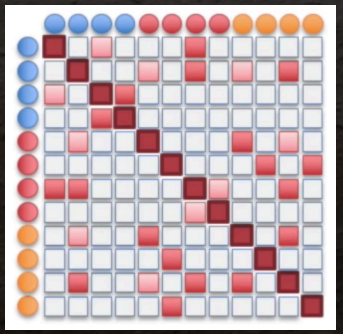


Compare all points, calculate similarity score from t-distribution



Compare to original distribution in 2D

Create some cost function, C



(this plot should have changed every iteration to look similar to the original)

t-distributed Stochastic Neighbour Embedding

Code

```
>>> import numpy as np
>>> from sklearn.manifold import TSNE
>>> X = np.array([[0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 1]])    #some 4x3 array
>>> T = TSNE(n_components=2, perplexity=30, learning_rate=200, n_iter=1000)
>>> X_embedded=T.fit_transform(X)
>>> X_embedded.shape
(4, 2)
```

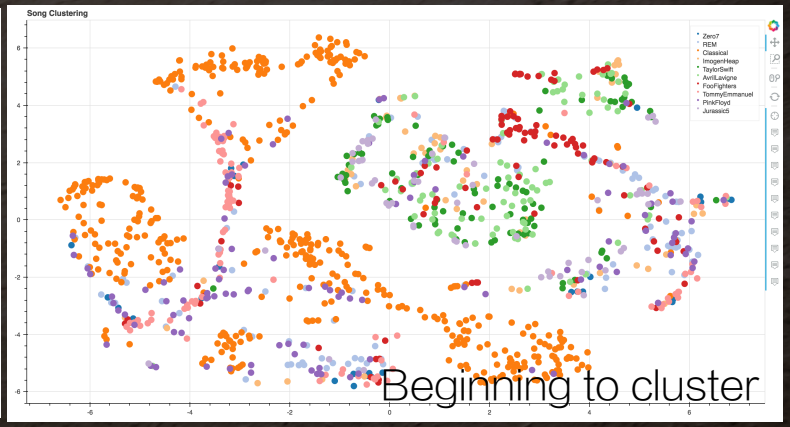
Hyper-parameters

Iterations: as many as required, larger data sets need more

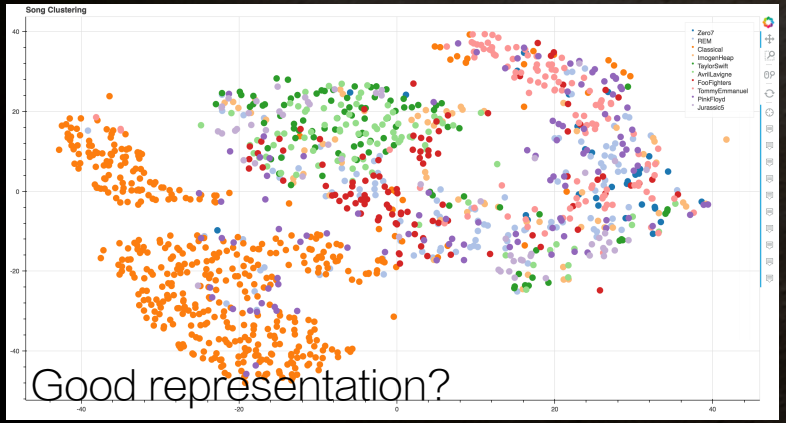
Perplexity: can be interpreted as a smooth measure of the effective number of neighbors per cluster. The performance is fairly robust to changes in the perplexity, and typical values are between 5 and 50

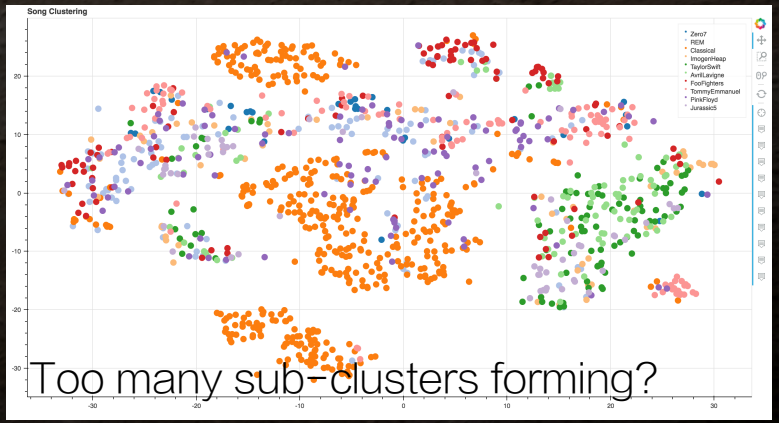
Learning rate: controls how quickly gradient decent takes place. Start low, but can save on iterations if larger. Too large can end up with all data points equally far away from each other ☹

t-distributed Stochastic Neighbour Embedding

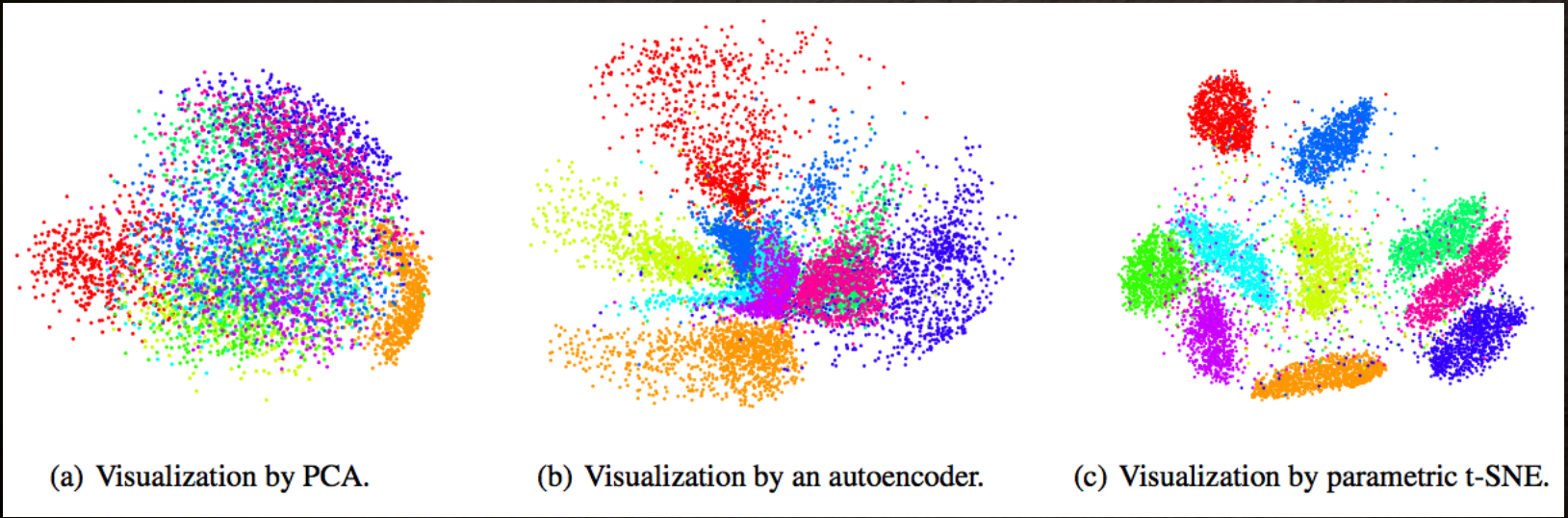


Varying the learning rate





t-distributed Stochastic Neighbour Embedding



(a) Visualization by PCA.

(b) Visualization by an autoencoder.

(c) Visualization by parametric t-SNE.

https://lvdmaaten.github.io/publications/papers/AISTATS_2009.pdf

t-distributed Stochastic Neighbour Embedding

Why use it?

- Extremely effective (non-linear) method in dimensionality reduction and finding clustering patterns

- Visualising the data and probing potential patterns helps give you insight and guide you with further analysis

t-distributed Stochastic Neighbour Embedding

References

t-SNE overview paper:
http://www.cs.toronto.edu/~hinton/absps/tsne.pdf

A nice explanatory video:
https://www.youtube.com/watch?v=NEaUSP4YerM

Parametric embedding (obtaining a model from t-SNE):
https://lvdmaaten.github.io/publications/papers/AISTATS_2009.pdf

Multicore t-SNE:
https://github.com/DmitryUlyanov/Multicore-TSNE