Data Mining 2

Topic 02 : Feature Engineering

Lecture 04 : Feature Selection

Dr Kieran Murphy

Department of Computing and Mathematics, Waterford Institute of Technology.
(Kieran.Murphy@setu.ie)

Spring Semester, 2025

Outline

- Curse of Dimensionality
- Feature selection

Feature Selection

Feature Selection

to efficiently find a minimum set of features that contain all the substantial information needed for predicting the target value.

- Initial feature vectors are typically not optimal since they may contain many redundant or irrelevant features.
- Hence their further processing is needed.
- Feature selection does not change existing features — it aims at selecting a subset of the most relevant features in order to both reduce the dimension of the feature vectors and remove unuseful 'noisy' initial features.
- Effect of feature selection is a more compact feature set:
  - improved model interpretability,
  - shorter training times,
  - enhanced prediction performance
    - lower dimension of feature vectors can reduce the risk of overfitting.
    - some learning methods do not work well if the feature set contains highly dependent features (e.g., Naïve Bayes learner, or SVM).

Curse of Dimensionality

> *The curse of dimensionality refers to various phenomena that arise when analysing and organising data high-dimensional spaces (often with hundreds or thousands of dimensions) that do not occur in low-dimensional settings.*
>
> —*wikipedia*

In other words:

> *The more columns you have, it creates a space that is more and more empty. And the more dimensions you have, the more likely data points sit on the edge of that space.*

Data sparsity

- When the dimensionality increases, the volume of the space increases so fast that the available data become sparse. This sparsity is problematic for any method that requires statistical significance.

$\Rightarrow$ In order to obtain a statistically sound and reliable result the space needs to be adequately sampled — the amount of data needed often grows exponentially with the dimensionality.

Dissimilarity of data points

- Organising and summarising data often relies on detecting areas where objects form groups with similar properties; in high dimensional data however all objects appear to be sparse and dissimilar in many ways which prevents common data organisation strategies from being efficient.

Curse of Dimensionality — Need for many observations I

High dimensional data is difficult to work because there are not enough observations to get good/reliable statistical estimates

Consider the following example:

- Generate n random bit strings of length d with the probability of a each bit being 1 is p.
- How big should n be to 'observe' most of the possible values?

This experiment described more formally is:

- Sample space consists of points in d dimensional spaces, where each dimension is the result of a Bernoulli experiment*.
- What — as a function of d and p — is the expected number of unique points seen relative to the size of the sample space (2^d) when generating a sample of n observations?

... could get out our probability notes ... but python is easier ...

*A Bernoulli distributions has two outcomes 0 and 1 with probability $1 - p$ and p respectively.

Curse of Dimensionality — Need for many observations II

```python
import numpy as np

def generate_observations(d, p, n):
    "Generate n random d-bit strings with prob p of bit set."
    observations = [
        "".join(np.random.choice(["0","1"], size=d, p=[1-p,p], replace=True))
        for k in range(n)
    ]
    return observations

def summary_coverage(d, p, n):
    "Report on coverage of d-bit string space using n observations."
    observations = generate_observations(d, p, n)
    nunique = len(set(observations))

    print(f"Input:"
        f"\n\tp(x=1) = {p}\n\tdimensions = {d}\n\tobservations = {n}")
    print(f"Output:"
        f"\n\tNumber of possible different values: {2**d}"
        f"\n\tNumber of observed different values: {nunique}")
```

Curse of Dimensionality — Need for many observations III

Output from function `generate_observations` ...

2
```
# demo generation
generate_observations(3, 0.5, 5)
```

```
['101', '010', '010', '011', '000']
```

Behaviour when $p = 0.5$ ...

3
```
summary_coverage(7, 0.5, 10**3)
```

```
Input:
    p(x=1) = 0.5
    dimensions = 7
    observations = 1000
Output:
    Number of possible different values: 128
    Number of observed different values: 128
```

- All $2^d = 128$ points in sample space are equally likely.
- n approximately the size of the sample space is sufficient.

Need exponentially more data as dimension grows.

Behaviour when p small ...

4
```
summary_coverage(7, 0.1, 1000)
```

```
Input:
    p(x=1) = 0.1
    dimensions = 7
    observations = 1000
Output:
    Number of possible different values: 128
    Number of observed different values: 52
```

5
```
summary_coverage(7, 0.1, 10**6)
```

```
Input:
    p(x=1) = 0.1
    dimensions = 7
    observations = 1000000
Output:
    Number of possible different values: 128
    Number of observed different values: 125
```

- Probability of observing individual points varies from $(1-p)^d = 0.4782969$ down to $p^d = 0.0000001$.

$\Rightarrow$ exponentially large n needed.

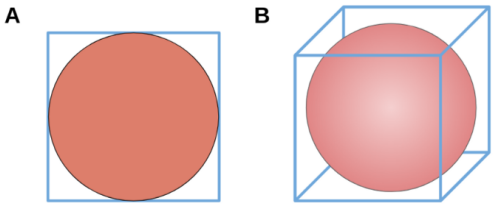
Curse of Dimensionality — Data Sparsity I

High-dimensional data is difficult to work not only because there are not enough observations to get good estimates . . . but also because data distributed in a high dimensional space necessarily tends to be very sparse.

Data Sparsity
implies long distances between randomly distributed points

Consider the following example:

- Generate uniformly distributed random points in a unit n-dimensional hypercube.
- What will be their average/expected distance from the origin?



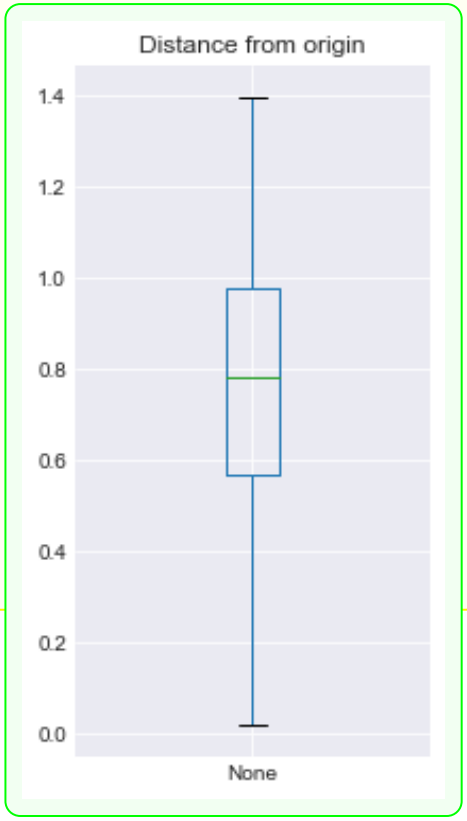
. . . could get our probability and linear algebra notes . . . but python is easier . . .

Curse of Dimensionality — Data Sparsity II

```
6 def dataSparsity(d,n):

    print("dimension = %s" % d )
    print("Number of points = %s" % n )
    print("Maximum possible distance from 0 is: %.2f" % np.sqrt(d))

    # generate n points in d-hypercube
    # a single call followed by a reshape is much faster
    # each row is a random point in d-dimensional space
    points = np.random.uniform(size=d*n).reshape((d,n))

    # axis=0 computes norm (L2 distance) for each row
    distances = np.linalg.norm(points, axis=0)

    # 5 number summary
    df = pd.Series(distances)
    print("\nDistribution of distance from origin\n%s:" % df.describe())

    greater_than_1 = 100*sum(distances > 1) / n
    print("%.1f %% of points more than 1 from origin." % (greater_than_1))

    return df
```

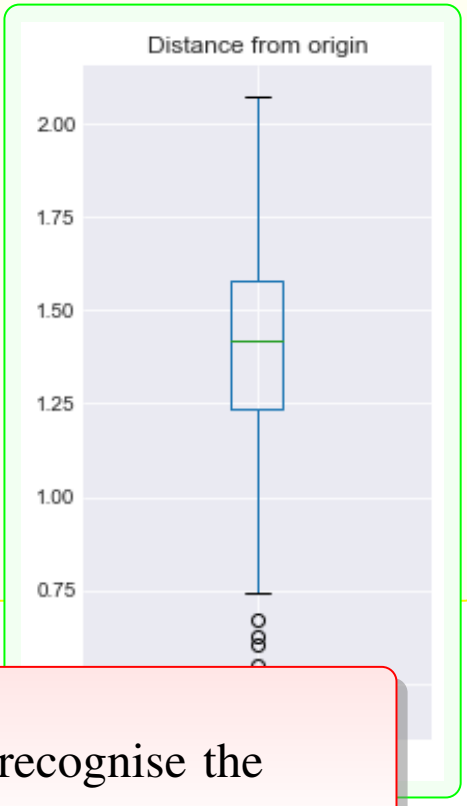
Curse of Dimensionality — Data Sparsity II

```
dimension = 2
Number of points = 1000
Maximum possible distance from 0 is: 1.41

Distribution of distance from origin
count   1000.000000
mean       0.761934
std        0.279541
min        0.018062
25%        0.568450
50%        0.782676
75%        0.976903
max        1.395899
dtype: float64:
20.5 % of points more than 1 from origin.
```



Curse of Dimensionality — Data Sparsity II

```
dimension = 6
Number of points = 1000
Maximum possible distance from 0 is: 2.45

Distribution of distance from origin
count   1000.000000
mean       1.403077
std        0.256048
min        0.429854
25%        1.233491
50%        1.416654
75%        1.579806
max        2.072292
dtype: float64:
93.6 % of points more than 1 from origin.
```

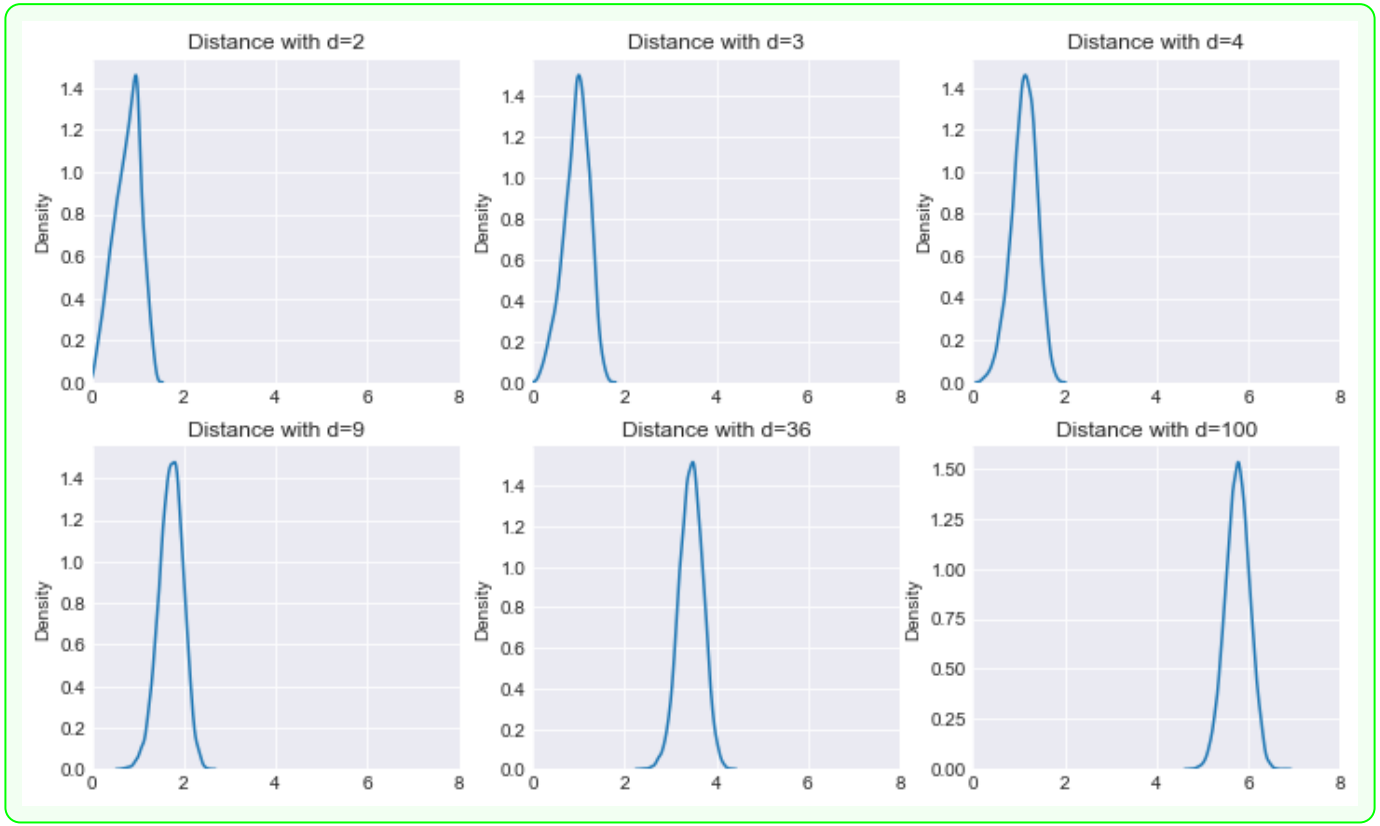


‘Near neighbours’ often do not exist!

Instead, typically you have only many ‘far neighbours’... and you cannot recognise the ‘similar ones’

Curse of Dimensionality — Data Sparsity III

If we plot the distribution of the distances from origin for increasing n, we see that the distances tend to increase with d.

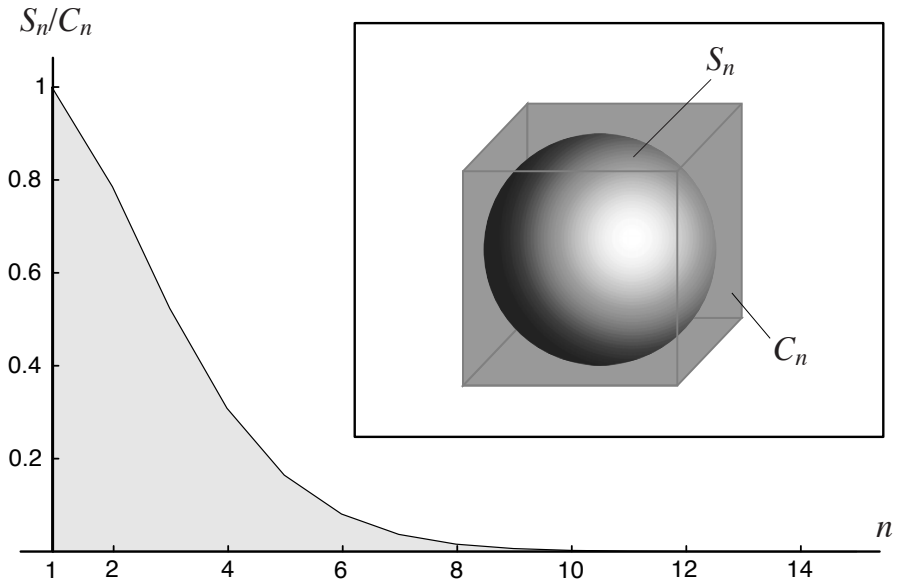


There is noting special about the origin — similar results are obtained if one looks at distances between two random points in d-dimensional space.

Curse of Dimensionality — Geometric Illustration

Another way of thinking about this issues is that as the dimension increases more and more data values will be in the corners of your space. A geometrical[†] illustration of this is possible by considering:

What is the volume of the largest hyper-sphere that placed inside a hyper-cube of side length 2?



n	V_n
1	2.00000
2	3.14159
3	4.18879
4	4.93480
5	5.26379
6	5.16771
7	4.72477
8	4.05871
9	3.29851
10	2.55016

[†]The curse of dimensionality (Mario Köppen)

Curse of Dimensionality — Alternative Viewpoint

Jeremy Howard of fast.ai's course Practical Deep Learning for Coders has a persuasive argument that curse of dimensionality is not a significant issue (see YouTube of lecture).

- Jeremy Howard is a very credible data scientist[‡] and fastai and the above mentioned course are excellent.
- His argument is essentially, that in the past theoretical results have dominated machine learning but now machine learning is driven by empirical performance and in real life datasets higher dimensionality does not appear to cause problems. Reason is that real data is not random and while it sits in some N-dimensional space, it actually occupies a much lower dimensional subspace.
- Full disclosure: He also is dismissive of the "No Free Lunch Theorem", his argument here, I find to be less persuasive.
- Doctors differ, patients die . . . so what should you do?
  Regardless of viewpoint, dimensionality reduction techniques do speed up model training, result in simpler and (sometimes) better performing models and so why not apply them?

[‡] Including his advocacy of wearing face masks as a preventative measure for COVID-19.

Feature selection methods — basic approaches

Feature selection methods can be divided into:

filters

Method selects feature subsets as a pre-processing step, independent of the learning method.

- Since features are selected based on criteria independent of any supervised learner, the performance of filters may not be optimum for a chosen learner.

wrappers

Method use an ML algorithm in conjunction with internal cross validation to score feature subsets by measuring their predictive power.

- use a learner as a black box to evaluate the relative usefulness of a feature subset.
- Wrappers search the best feature subset for a given supervised learner, however, wrappers tend to be computationally expensive.

embedded methods

Method perform feature selection during the process of training.

- Instead of treating a learner as a black box, embedded methods select features using the information obtained from training a learner.

Feature Selection using Filters

Feature selection using filters are heuristic

- We need a function to estimate/evaluate how useful a feature is.
- Frequently/mostly used:
  Information Gain, Gini Index, Chi-square, correlation coefficient, etc.
- Advantages: fast, easily parallelizable.
- Disadvantages: such methods consider only one variable's contribution without other variables' influences.
- However, using them you can easily recognise
  - really useful ones
  - completely unuseful ones
  - highly dependent/correlated ones
- When I use filter methods I tend to error towards including borderline features rather than excluding, because multiple borderline features might have a strong contribution when combined.

Feature Subset Selection using Wrappers

Simple approaches to wrappers — usually greedy procedures

According to the search starting point, we can distinguish three main greedy approaches:

- greedy forward selection
  - start with no variables and add them one by one, at each step adding the one that decreases the error the most, until any further addition does not significantly decrease the error.
- greedy backward elimination
  - start with all the variables and remove them one by one, at each step removing the one that decreases the error the most, until any further removal does not significantly decrease the error.
- a mix of the previous
  - stepwise selection heuristics which start with a (random) subset then adds or removes the features that decreases the error the most, until any further change not significantly decrease the error.

Iterative Feature Subset Selection

A typical feature selection process consists of four basic steps: subset generation, subset evaluation, stopping criterion, and result validation.

The main step is the subset generation. During the subset generation, each candidate subset is evaluated and compared with the previous best one according to a certain evaluation criterion. If the new subset turns out to be better, it replaces the previous best subset. The process of subset generation and evaluation is repeated until a given stopping criterion is satisfied.

General scheme of feature subset selection iterative process
Input: an original feature set

1. Generate a subset
2. Evaluate the subset
3. Test the chosen stopping criterion; if not fulfilled, repeat Step 1
4. Result validation

These slides are intended to be theoretical, but I should mention sklearn's excellent `sklearn.feature_selection.RFECV` that performs recursive feature selection via cross validation, for any learner that supports reporting of feature importance.