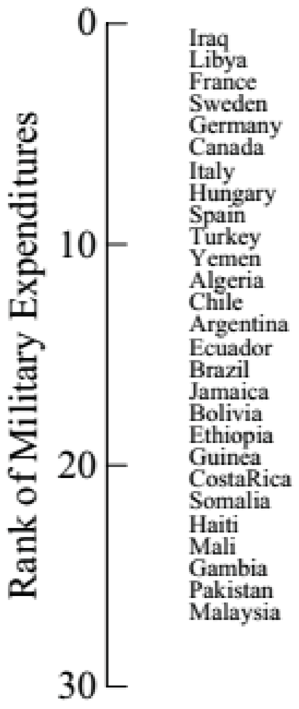
VISUALISATION

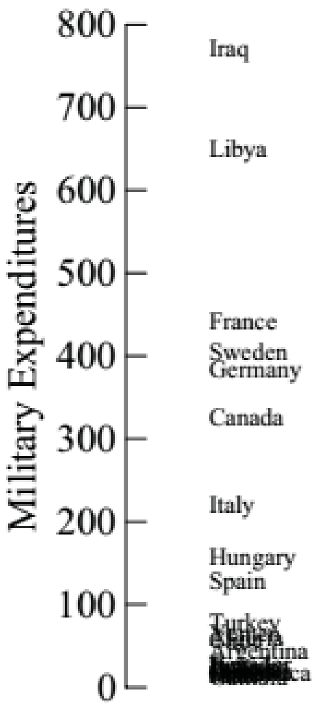
Lecture 7: dimension reduction

Damon Wischik

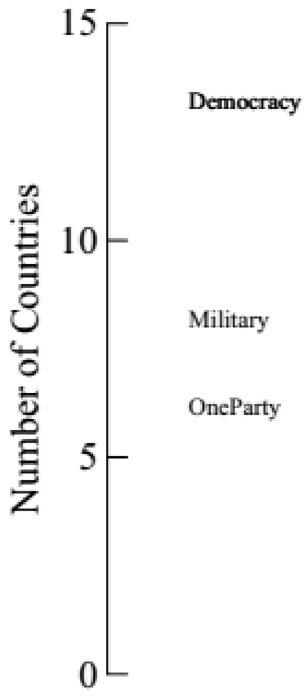
Ordinal: we can ask which is greater, but not measure how much



Interval: we can subtract one value from another



Ratio: we can divide one value by another

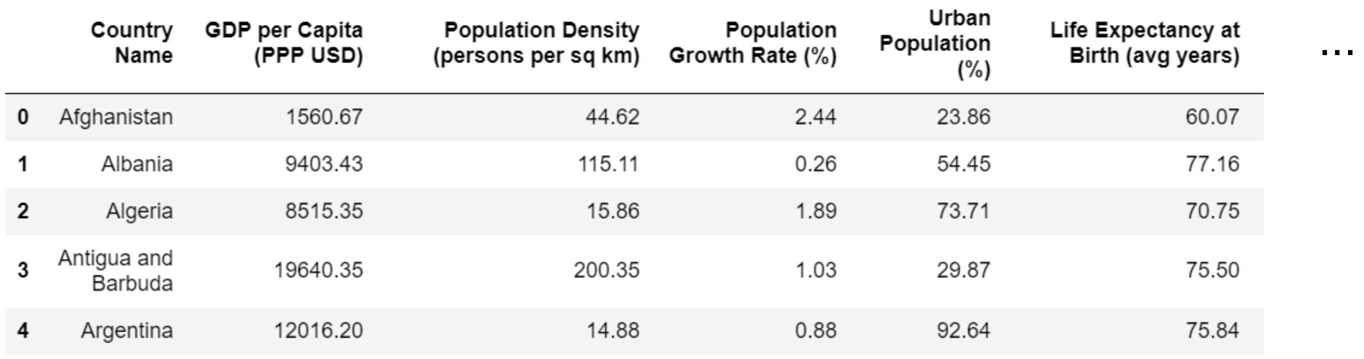


Ordinal scales show us proximity of datapoints.
Interval and ratio scales show us clustering.

How do we decide which scale to show?
What do we do when there are too many scales to choose from?

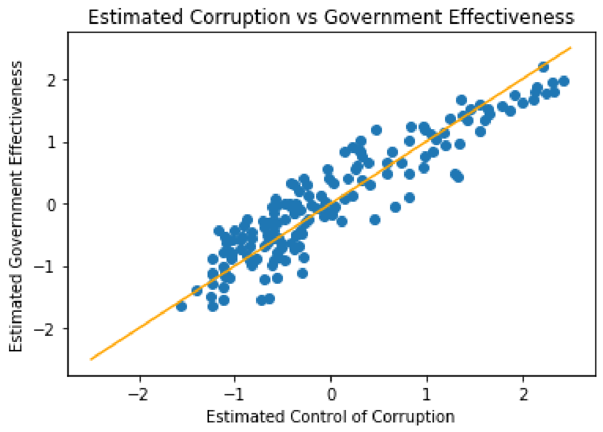
Principal Components Analysis

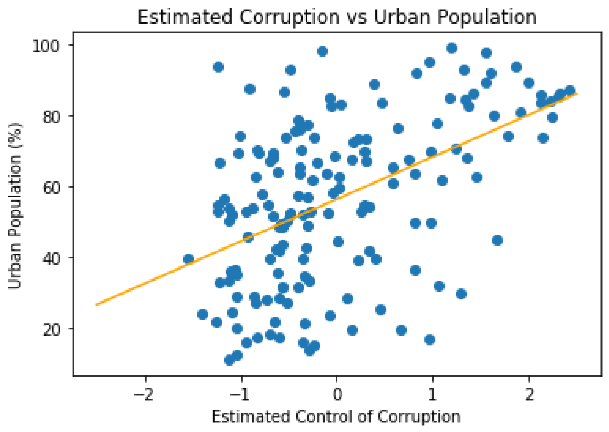
DATASET: world bank demographic indicators for 161 countries

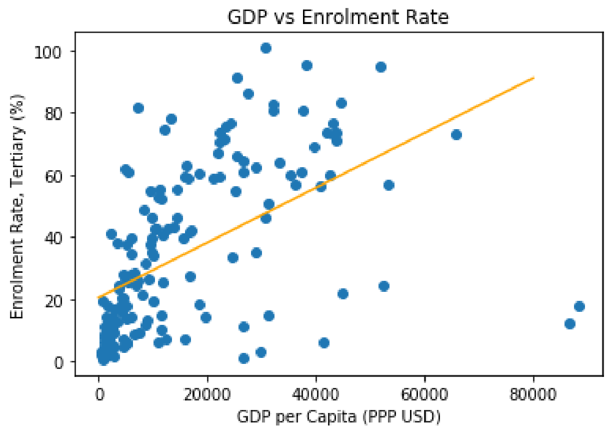


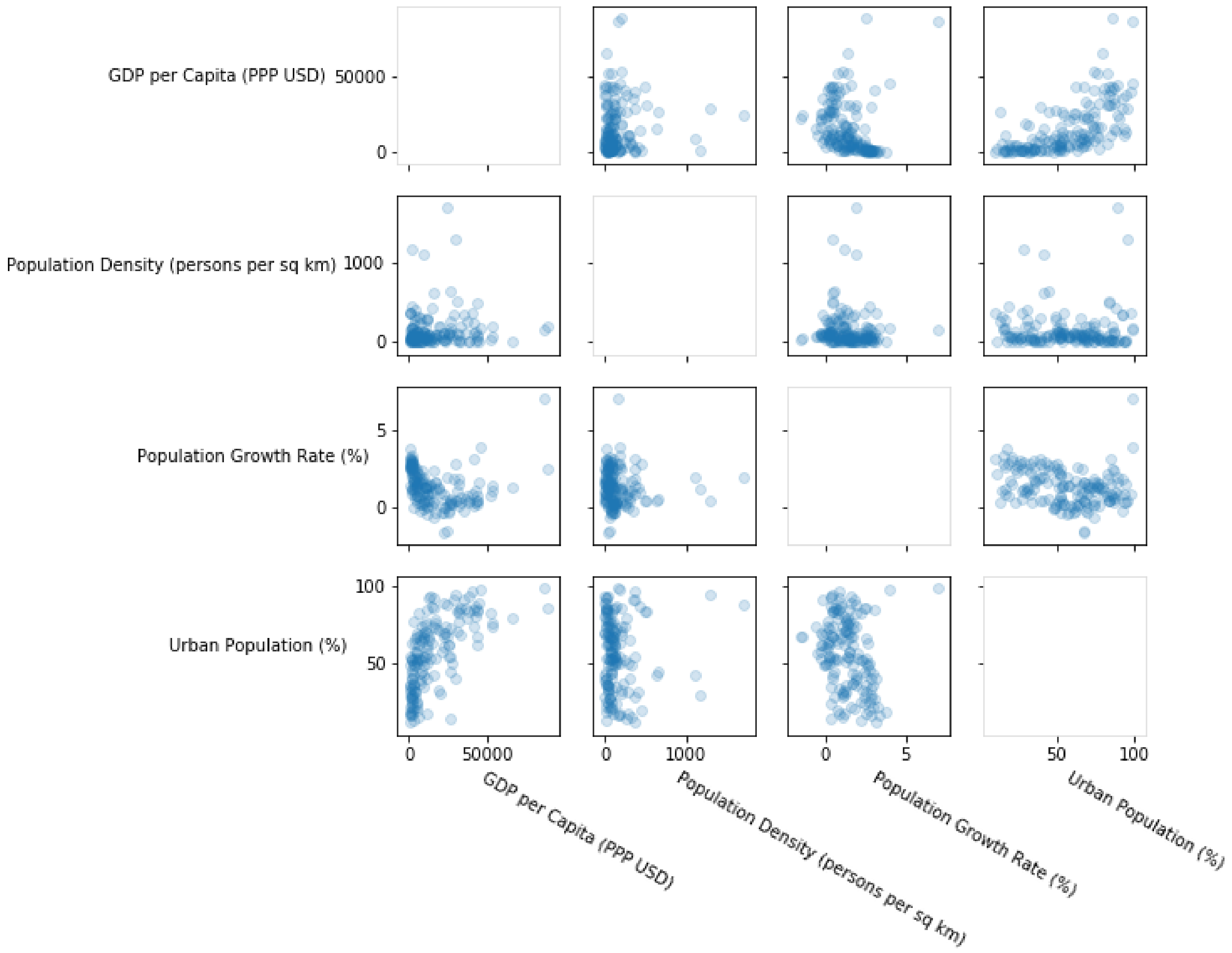
	Country Name	GDP per Capita (PPP USD)	Population Density (persons per sq km)	Population Growth Rate (%)	Urban Population (%)	Life Expectancy at Birth (avg years)	...
0	Afghanistan	1560.67	44.62	2.44	23.86	60.07	
1	Albania	9403.43	115.11	0.26	54.45	77.16	
2	Algeria	8515.35	15.86	1.89	73.71	70.75	
3	Antigua and Barbuda	19640.35	200.35	1.03	29.87	75.50	
4	Argentina	12016.20	14.88	0.88	92.64	75.84	

This dataset has 12 columns, in addition to Country Name.
In Lecture 2 we ran linear regression on some of them.
Who decides which variable is the response and which is the predictor?

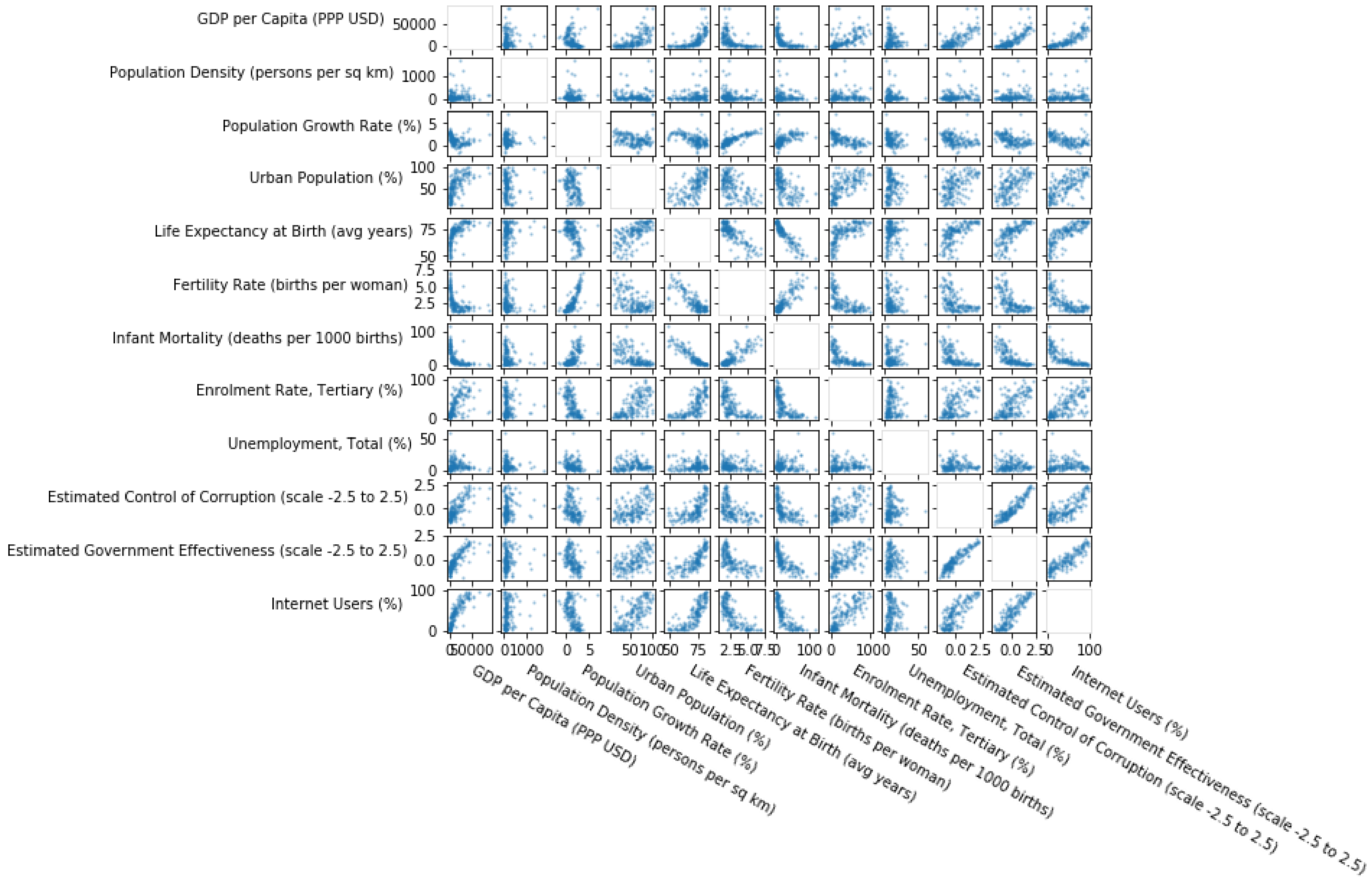




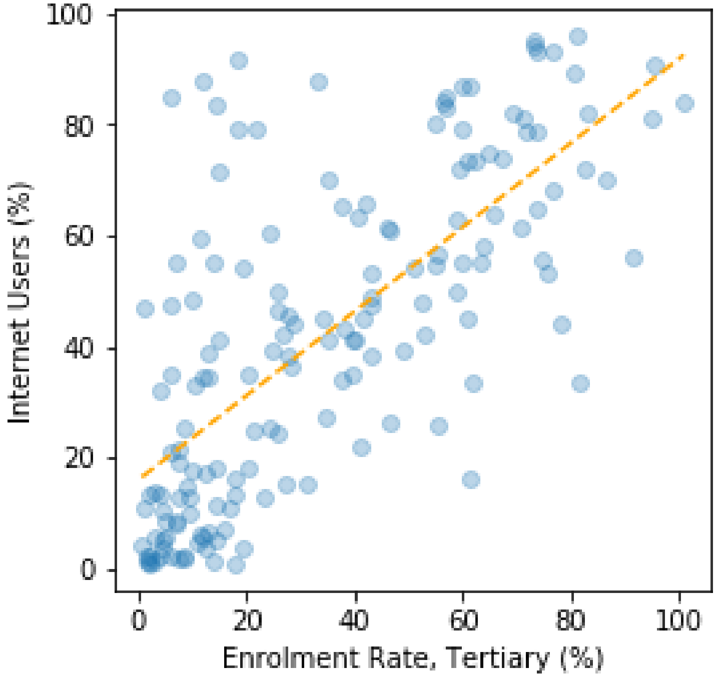




Why just those two variables? (This chart is called a *splom*, scatter plot matrix).



A splom is unwieldy for 12 variables. And what if we had 1200 or 12000?
Is there a way to condense them? Perhaps it would help us see natural clusters.

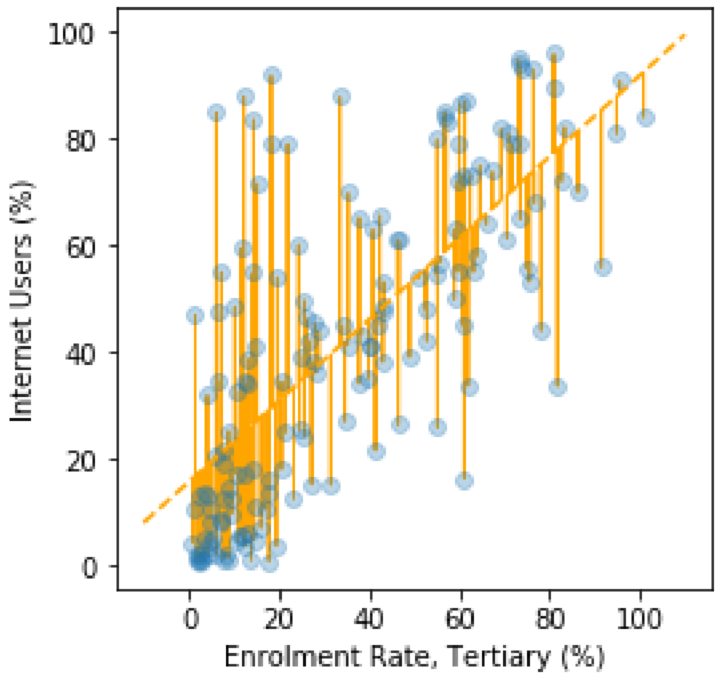


Linear regression: model the data by

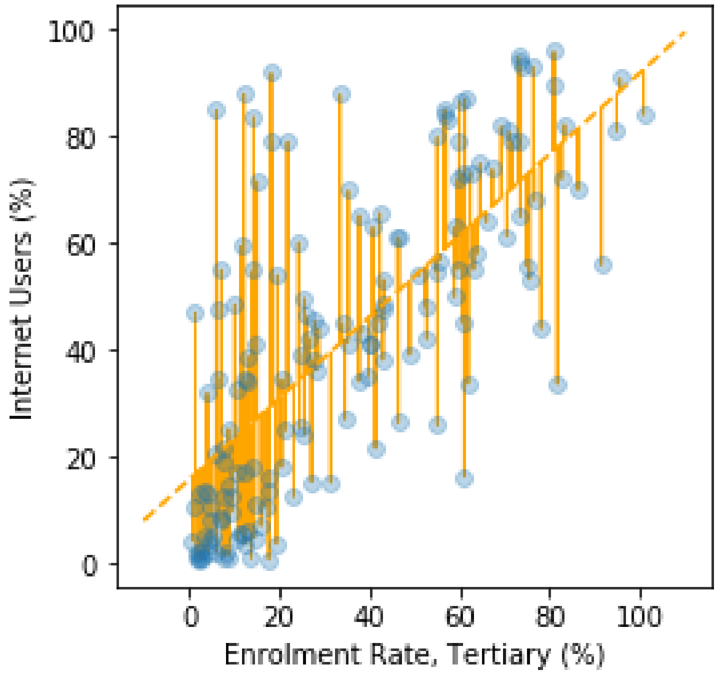
$$y_i = \alpha\, x_i + \beta$$

and choose the parameters α and β to minimize

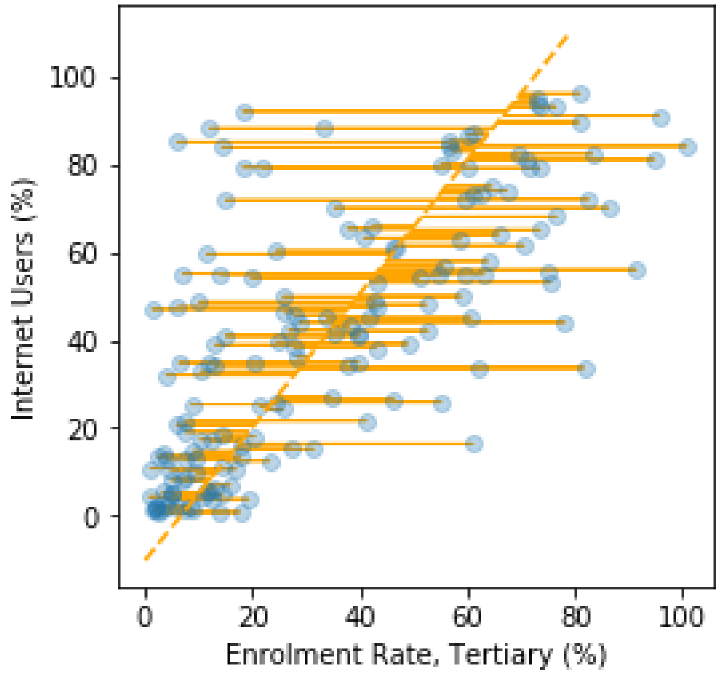
$$L(\alpha, \beta) = \frac{1}{2} \sum_{i=1}^{n} (\underset{\substack{\uparrow \\ \text{predicted}}}{\alpha\, x_i + \beta} - \underset{\substack{\uparrow \\ \text{observed}}}{y_i})^2$$



$$y = 15.81 + 0.7619x$$



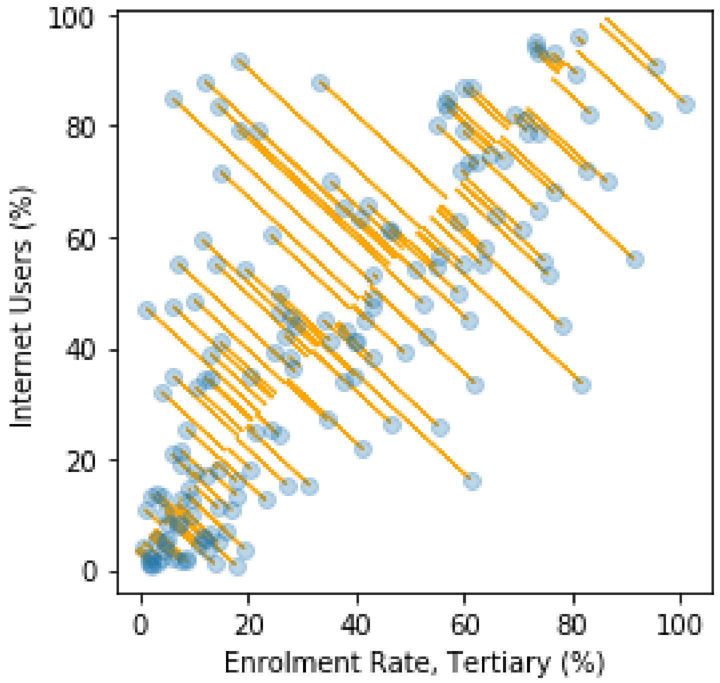
$$x = 6.728 + 0.6583y$$
$$\Rightarrow \quad y = -10.22 + 1.519x$$



If we regress y against x, should we get the same answer as regressing x against y?

If not, is there a way to express the relationship between x and y that doesn't require an arbitrary choice of response versus predictor?

$y = 3.89 + 1.11x$



Optimal projection: model the data by

$$y = \alpha\, x + \beta$$

and choose the parameters α and β to minimize

$$L(\alpha,\beta) = \frac{1}{2}\sum_{i=1}^{n} \left\| \begin{pmatrix} x_i \\ y_i \end{pmatrix} - \underset{\uparrow}{\mathrm{proj}_{\alpha,\beta}} \begin{pmatrix} x_i \\ y_i \end{pmatrix} \right\|^2$$

project $\begin{pmatrix} x_i \\ y_i \end{pmatrix}$ onto the linear subspace $y = \alpha x + \beta$

A better way to write this subspace is

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} \mu_x \\ \mu_y \end{pmatrix} + \lambda_i \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} + \begin{pmatrix} \varepsilon_{x,i} \\ \varepsilon_{y,i} \end{pmatrix}$$

Each 2d record (x_i, y_i) in the dataset is summarised by a 1d summary λ_i.

We can generalize the method, to reduce an arbitrary number of data dimensions down to one dimension.

Optimal 1d projection: model the data by projecting it onto a linear subspace,

$$\vec{x}_i = \vec{\mu} + \lambda_i \vec{\delta} + \vec{\varepsilon}_i$$

and choose the subspace to minimize

$$L = \frac{1}{2} \sum_{i=1}^{n} \|\vec{\varepsilon}_i\|^2$$

```python
fx,fy = 'Enrolment Rate, Tertiary (%)', 'Internet Users (%)'
X = countries[[fx,fy]].values

pca = sklearn.decomposition.PCA()
pca_result = pca.fit_transform(X)

# Report the slope
δx,δy = pca.components_[0]

# Report the means
μx,μy = pca.mean_

# Report the predictions
λ = pca_result[:,0]
predx,predy = μx+λ*δx, μy+λ*δy
```

We can generalize the method, to reduce an arbitrary number of data dimensions down to an arbitrary number of dimensions.

PCA projection: model the data by representing the entire space using an optimal basis

$$\vec{x}_i = \vec{\mu} + \sum_{k=1}^{K} \lambda_{k,i} \vec{\delta}_k$$

where K is the number of features for each datapoint, and the basis vectors $\vec{\delta}_k$ are chosen so that partial projections onto $L < K$ dimensions

$$\vec{x}_i \approx \vec{\mu} + \sum_{k=1}^{L} \lambda_{k,i} \vec{\delta}_k$$

are as accurate as possible, in the mean square error sense.

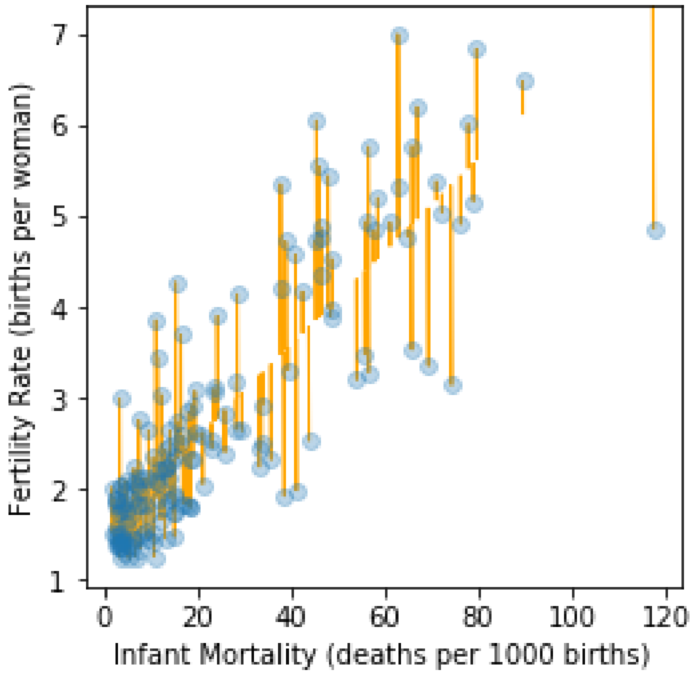
```python
X = countries[features].values
pca = sklearn.decomposition.PCA()
pca_result = pca.fit_transform(X)

μ = pca.mean_
pred = μ + np.zeros_like(pca_result)
for k in range(L):        # L = number of PCA components to use
    λk = pca_result[:,k]
    δk = pca.components_[k]
    pred = pred + λk.reshape((-1,1)) * δk.reshape((1,-1))
```

Optimal 1d projection: model the data by

$$\begin{pmatrix} x_i \\ y_i \end{pmatrix} = \begin{pmatrix} \mu_x \\ \mu_y \end{pmatrix} + \lambda_i \begin{pmatrix} \delta_x \\ \delta_y \end{pmatrix} + \begin{pmatrix} \varepsilon_{x,i} \\ \varepsilon_{y,i} \end{pmatrix}$$

and choose the parameters μ and δ to minimize

$$L(\mu, \delta) = \frac{1}{2} \sum_{i=1}^{n} \left\| \begin{pmatrix} x_i \\ y_i \end{pmatrix} - \mathrm{proj}_{\mu,\delta} \begin{pmatrix} x_i \\ y_i \end{pmatrix} \right\|^2$$

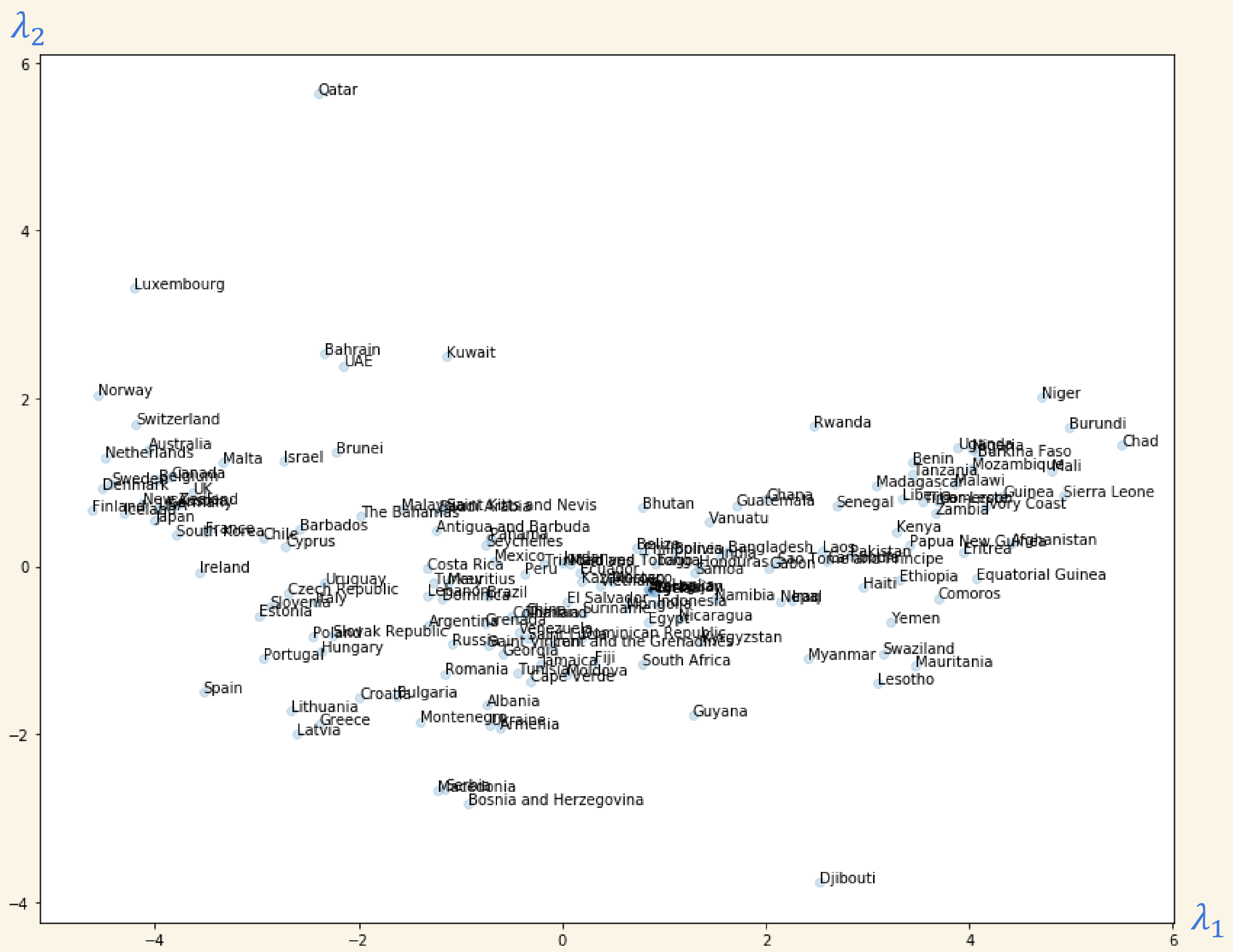


There is a gotcha.

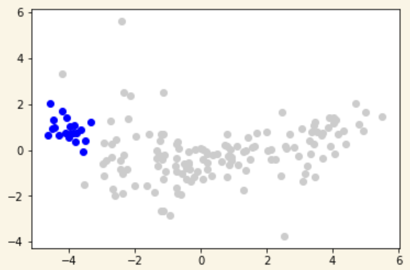
Why do these two variables come out looking so different?

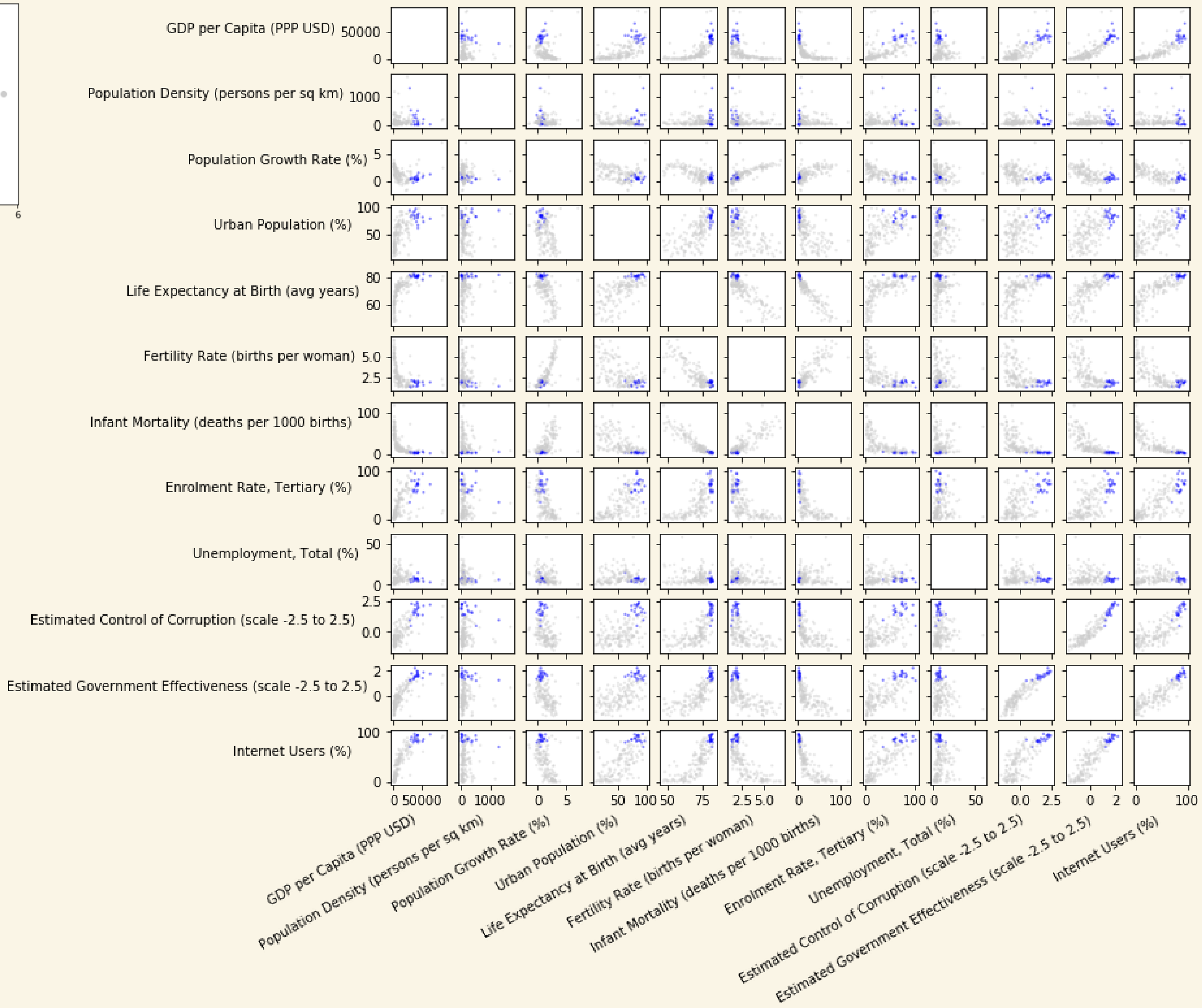
Solution: scale the x and y columns so they have the same standard deviation, before doing the fit.

Run PCA, then project onto the first two principal components, to get a nice visualization.

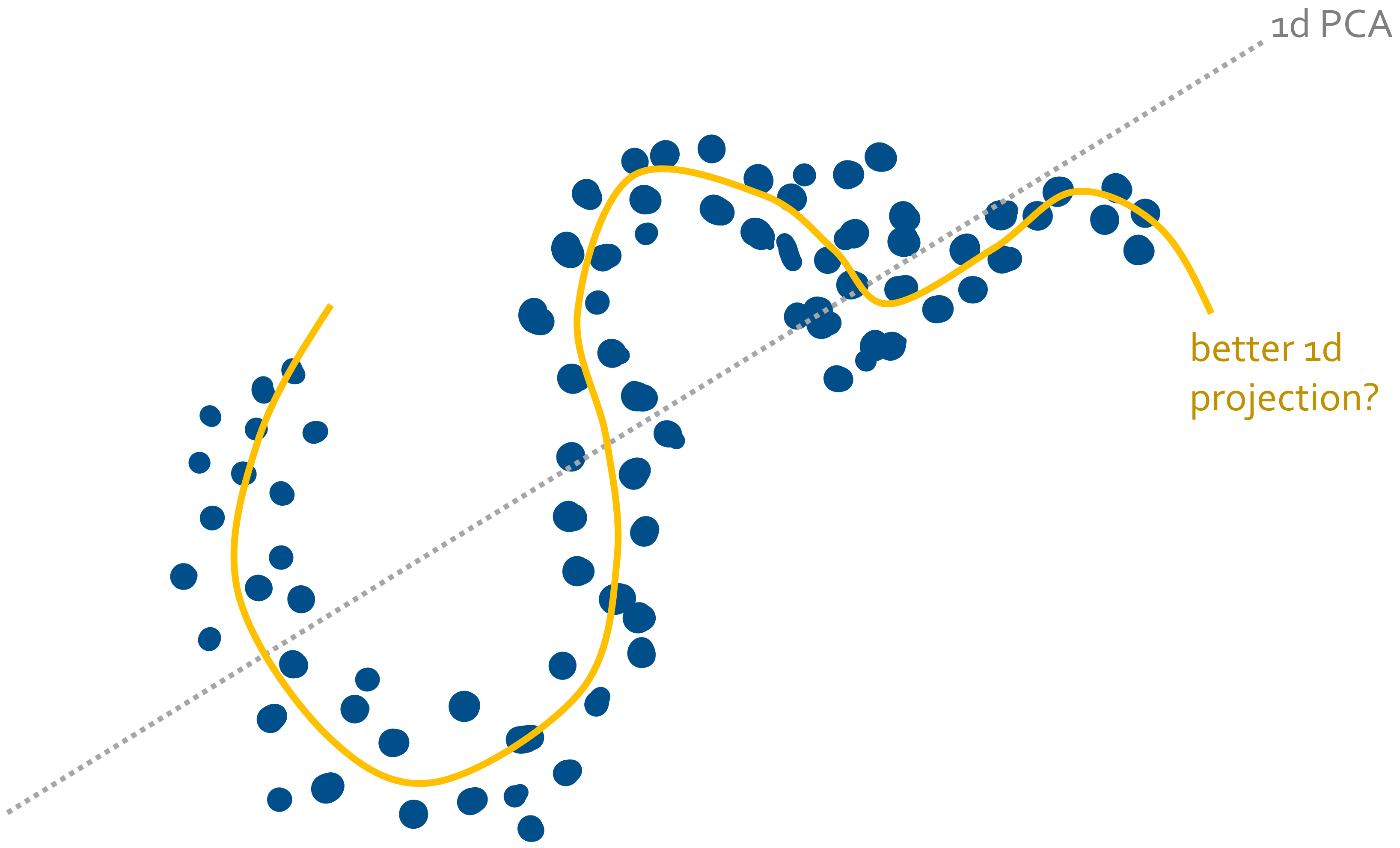


Run PCA, then project, and you may spot clusters.





But what if the data isn't linear?

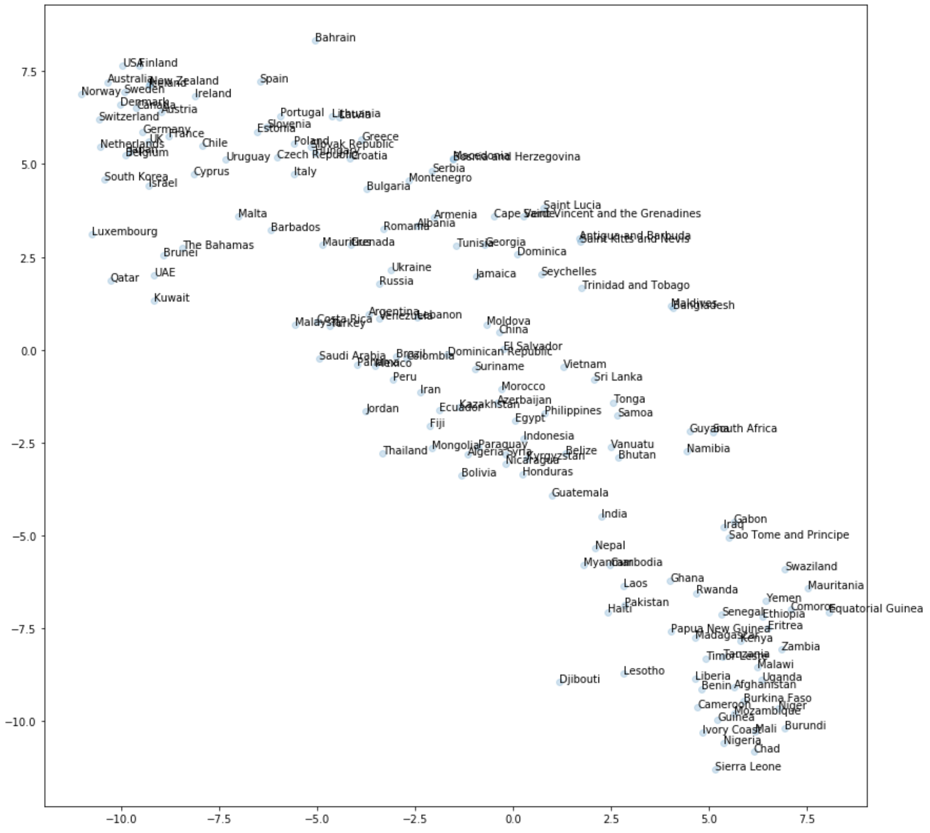


t-SNE

(stochastic neighbourhood embedding with the *t* distribution)

t-SNE is another method for dimension reduction

```python
X = countries[features].values
# scale the columns, so they have the same variance
for k in range(len(features)):
    X[:,k] = X[:,k] / np.std(X[:,k])

# you have to declare up front how many dimensions you want to reduce to
tsne = sklearn.manifold.TSNE(n_components=2)
tsne_results = tsne.fit_transform(X)

p1,p2 = tsne_results[:,0], tsne_results[:,1]
plt.scatter(p1, p2, alpha=.2)
```



The idea behind t-SNE

1. Compute the $n \times n$ distance matrix between all n datapoints in the original dataset
$$d_{i,j} = \left\|\vec{x}_i - \vec{x}_j\right\|$$

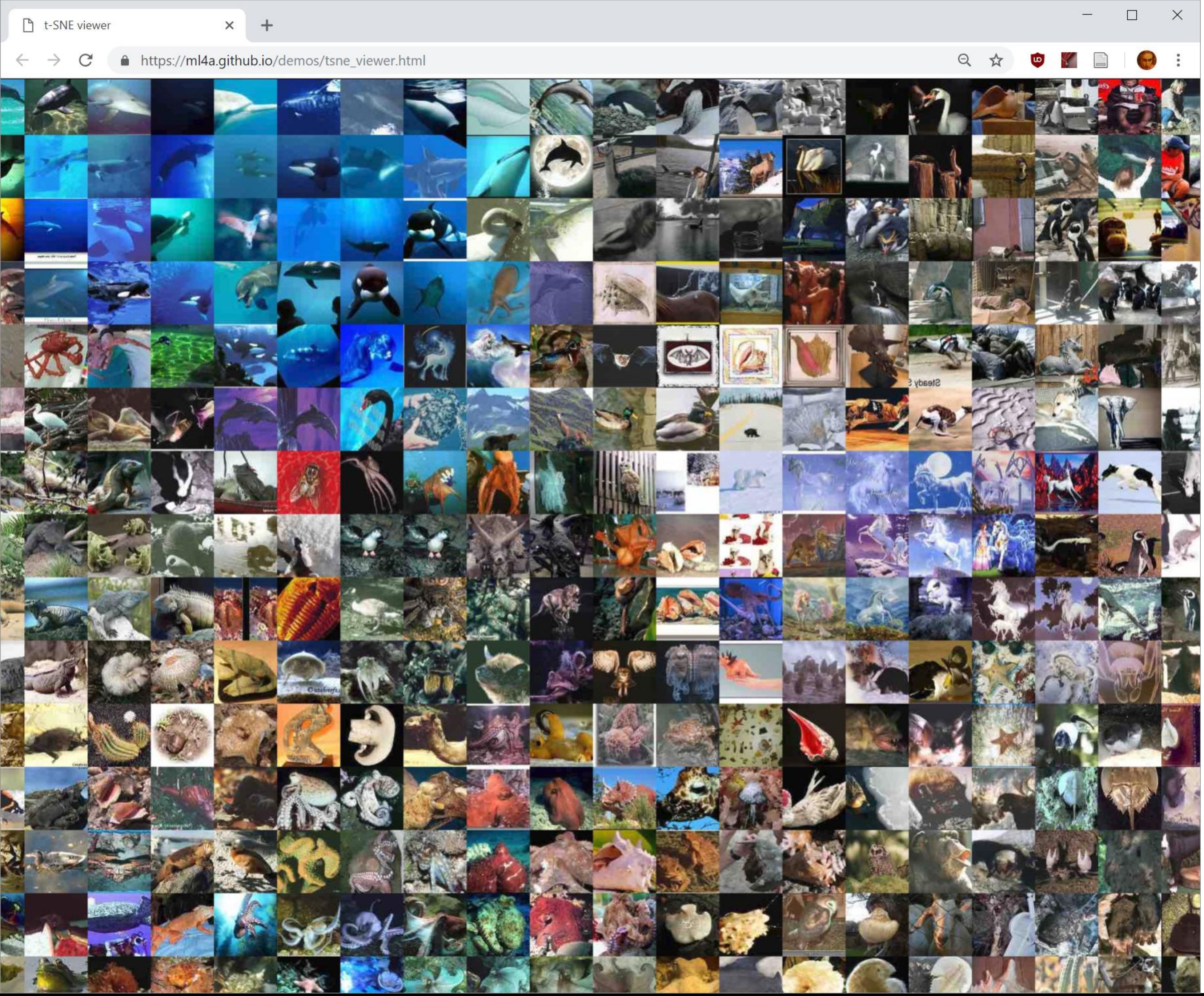
Most implementations insist on Euclidean distance. Make sure it's appropriate.

2. Let $\vec{z}_i \in \mathbb{R}^2$ be an arbitrary *embedding* of $\vec{x}_i$ into 2 dimensions, and compute the $n \times n$ distance matrix in this space
$$e_{i,j} = \left\|\vec{z}_i - \vec{z}_j\right\|$$

3. Define an appropriate loss function that measures the difference between these two distance matrices
$$L(d, e(z)) = \cdots$$

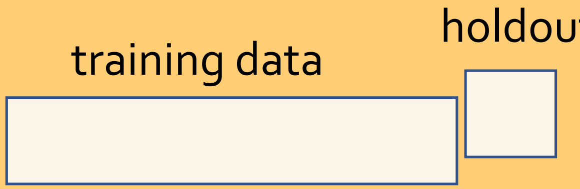
it involves probability & the t-distribution, hence t-SNE

4. Pick the embedding to minimize this loss
$$\min L\big(d, e(z)\big) \quad \text{over} \quad z \in \mathbb{R}^{n\times 2}$$

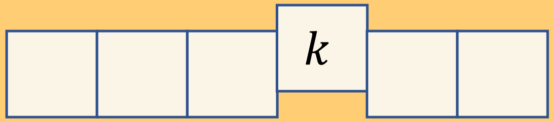


You can also force the points onto a grid, using the Jonker-Volgenant algorithm https://blog.sourced.tech/post/lapjv/

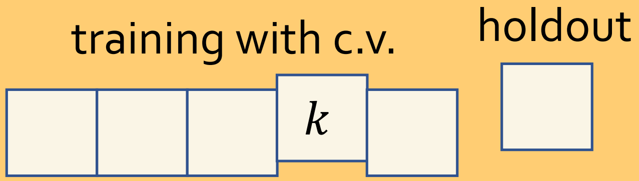
Cross-validation, generative models, and perplexity



simple validation: set aside some holdout data, train the model on the rest, and evaluate your model on the holdout set



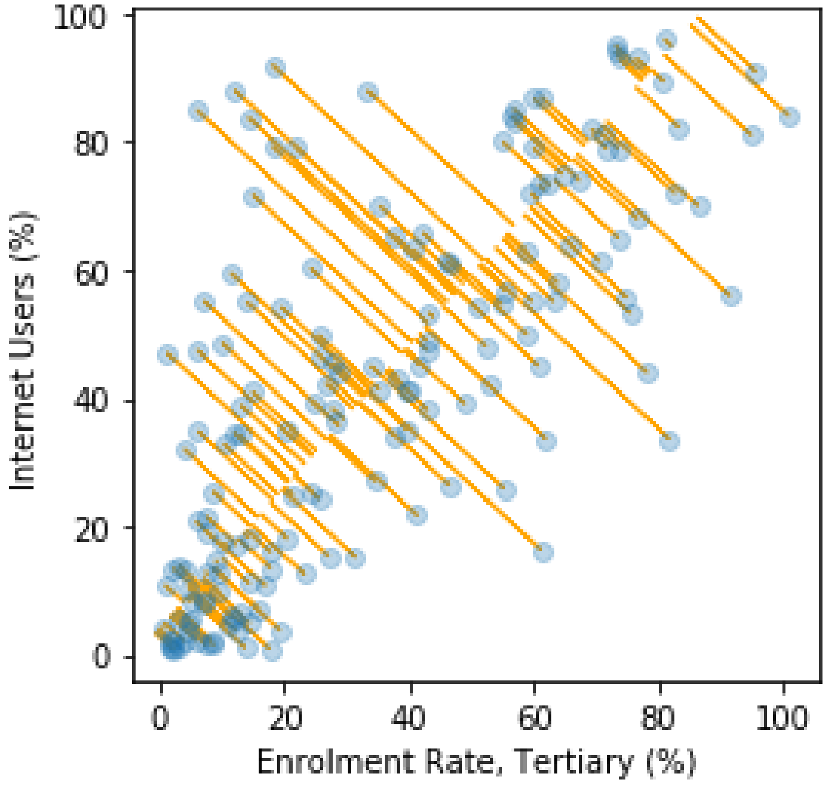
cross validation: for each block k, train your model on the rest, and evaluate on block k, then average the results



cross validation + model tuning: use cross validation to tune parameters etc., then evaluate your tuned model on a holdout set

Parameter tuning: how many dimensions to use?
Model tuning: which features to use?
Method choice: PCA or t-SNE?

How should we evaluate the performance of a model on the holdout set?

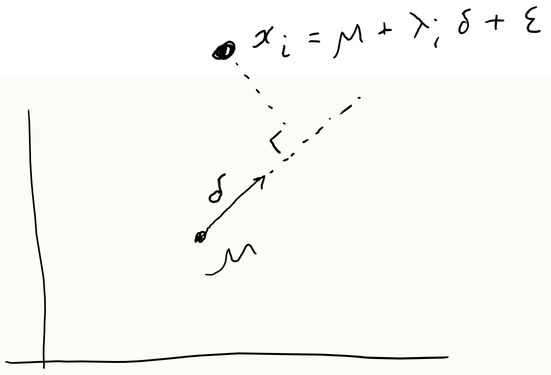


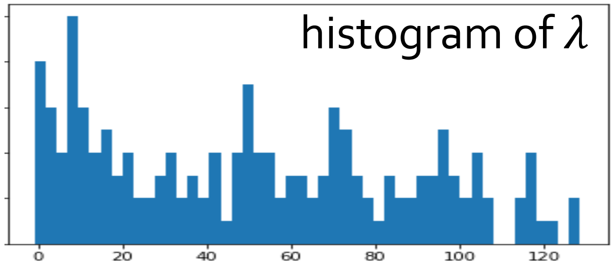
Is there a way to do this for t-SNE?

Probabilistic generative models

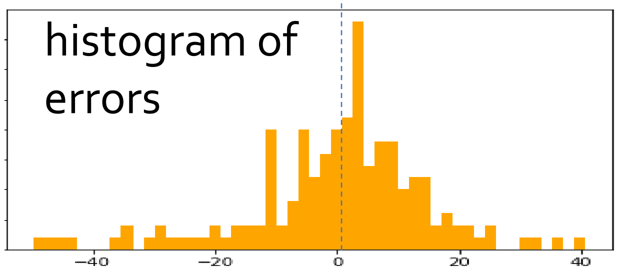
From a reduced-dimension model, we can synthesize new random values like those in the dataset.

```
fit a PCA model
get μ, δ
get a histogram of values of λ and of ε

def random_datapoint():
    pick a random newλ from the (smoothed) distribution found for λ
    pick a random newε from the (smoothed) distribution found for ε
    return μ + newλ*δ + newε
```

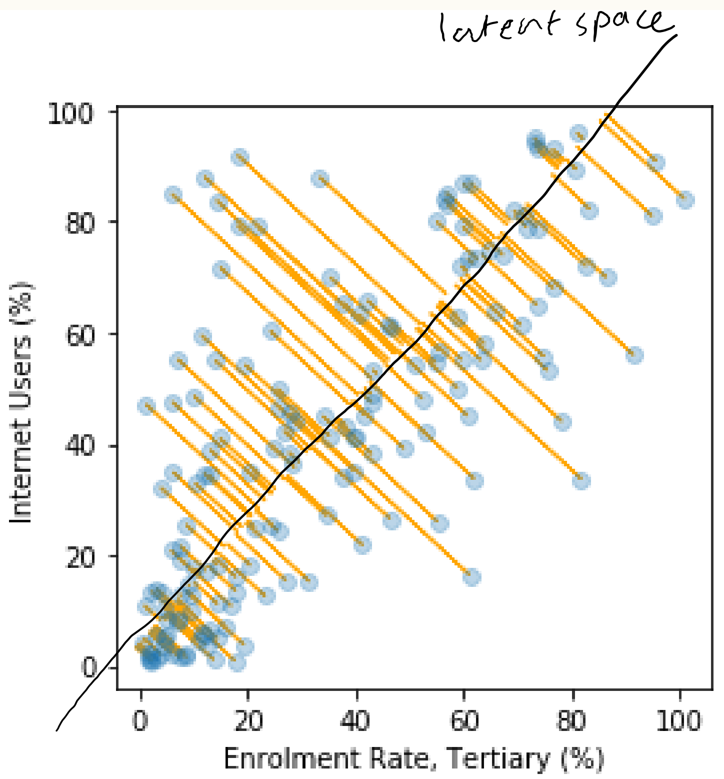




+



=



Perplexity

A generative model defines a probability density function $P(\vec{x})$, the probability of seeing datapoint $\vec{x}$.

If it's a good model, $P(\vec{x})$ will be high for the holdout datapoints. We can evaluate the model on the m holdout datapoints by

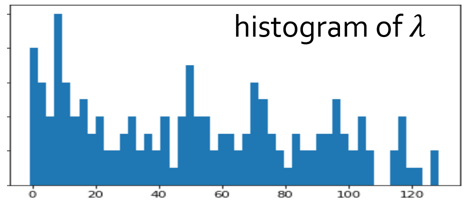
$$\text{loss} = -\frac{1}{m}\sum_{i=1}^{m} \log_2 P(\vec{x}_i)$$

$$\text{perplexity} = 2^{\text{loss}}$$

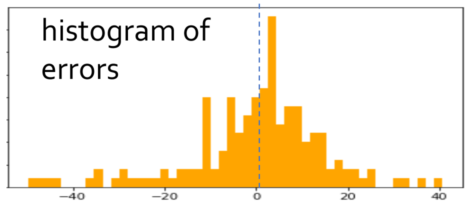
high P is good so low loss is good

"perplexity": how surprised the model is to see a new datapoint

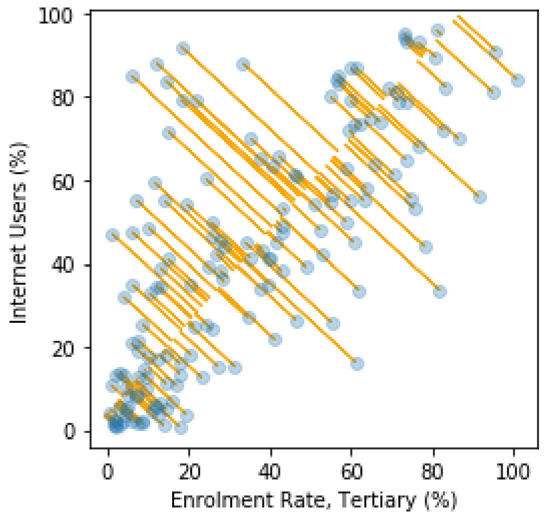
loss = average number of bits needed to describe a new datapoint



+



=

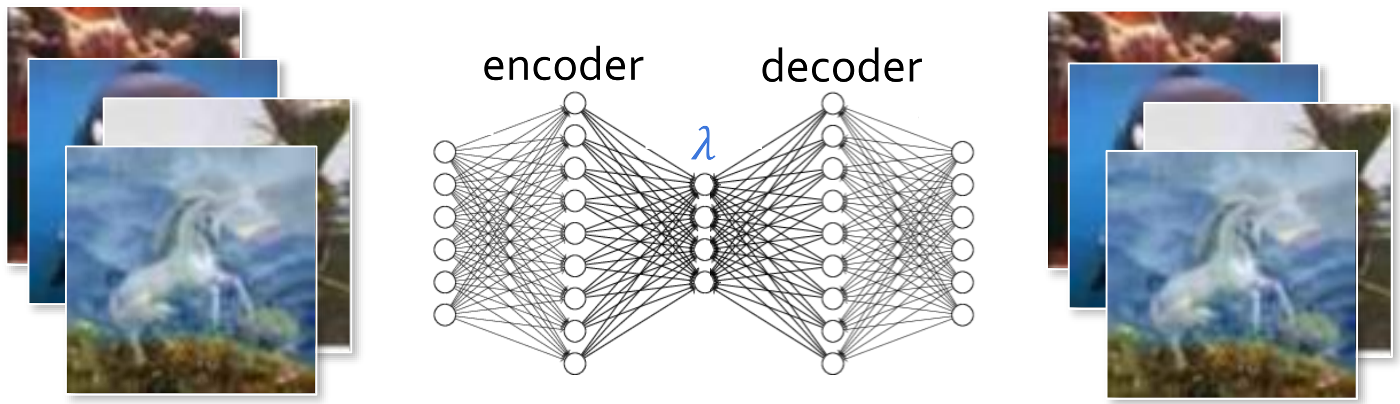


Variational autoencoders

PCA is more interpretable: it gives a decomposition into latent variable + noise, which allows cross validation.

t-SNE is less rigid: it's not limited to linear projections.

Can we get the best of both?



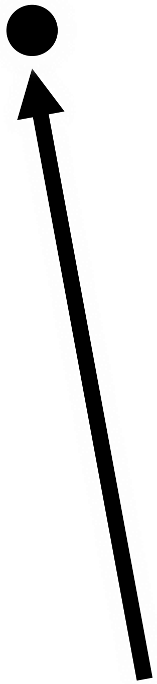
1. Train a neural network to reproduce its input.
   This automatically gives a decomposition, datapoint = f (latent variable) + noise
2. Have just a few nodes in the middle layer,
   so that it achieves dimension-reduction
3. Include in the loss function "The distribution of λ should be independent Normal(0,1)",
   similar to the t-SNE loss function

Latent space interpolation

https://hackernoon.com/latent-space-visualization-deep-learning-bits-2-bd09a46920df

let enc(x_1)=λ_1

let enc(x_2)=λ_2



let enc(x_3)=λ_3

let $\lambda_{\text{new}} = 0.4\lambda_1 + 0.4\lambda_2 + 0.2\lambda_3$
what is dec(λ_{new})?