Dimensionality Reduction in Python

Thursday, 27 August 2020 10:12 PM

Exploring high dimensional data

Tidy data

Name	Type	HP	Attack	Defense	Speed	Generation
Bulbasaur	Grass	45	49	49	45	1
Ivysaur	Grass	60	62	63	60	1
Venusaur	Grass	80	82	83	80	1
Charmander	Fire	39	52	43	65	1
Charmeleon	Fire	58	64	58	80	1

When I mention dimensionality, I mean the number of columns you have in your dataset,

Tidy data

Name	Type	HP	Attack	Defense	Speed	Generation
Bulbasaur	Grass	45	49	49	45	1
Ivysaur	Grass	60	62	63	60	1
Venusaur	Grass	80	82	83	80	1
Charmander	Fire	39	52	43	65	1
Charmeleon	Fire	58	64	58	80	1

just make sure every column resembles a variable, or in other words

eature, such as the properties of Pokemon like attack and defense in this example.

Tidy data

Name	Type	HP	Attack	Defense	Speed	Generation
Bulbasaur	Grass	45	49	49	45	1
Ivysaur	Grass	60	62	63	60	1
Venusaur	Grass	80	82	83	80	1
Charmander	Fire	39	52	43	65	1
Charmeleon	Fire	58	64	58	80	1

each row holds an observation for each variable.

```
pokemon_df.shape
```

```
(5, 7)
```

When you have many columns in your dataset, say, more than 10, the data is considered high dimensional.

When to use dimensionality reduction?

Name	Type	HP	Attack	Defense	Speed	Generation
Bulbasaur	Grass	45	49	49	45	1
Ivysaur	Grass	60	62	63	60	1
Venusaur	Grass	80	82	83	80	1
Charmander	Fire	39	52	43	65	1
Charmeleon	Fire	58	64	58	80	1

f we're interested in how Pokemon are different, this feature would not be very useful since it has no variance.

The describe method

```
pokemon_df.describe()
```

	HP	Attack	Defense	Speed	Generation
count	5.0	5.0	5.0	5.0	5.0
mean	56.4	61.8	59.2	66.0	1.0
std	15.9	13.0	15.4	14.7	0.0
min	39.0	49.0	43.0	45.0	1.0
25%	45.0	52.0	49.0	60.0	1.0
50%	58.0	62.0	58.0	65.0	1.0
75%	60.0	64.0	63.0	80.0	1.0
max	80.0	82.0	83.0	80.0	1.0

describe() ignores the non-numeric columns in a dataset.

You'll see that for our small dataset the standard deviation of the generation column is zero, and that the maximum and minimum values are the same.

The describe method

```
pokemon_df.describe(exclude='number')
```

	Name	Type
count	5	5
unique	5	2
top	Charmander	Grass
freq	1	3

describe() to do the opposite by passing 'number' to the 'exclude' argument.

Why reduce dimensionality?

Your dataset will:

- be less complex
- require less disk space
- require less computation time
- have lower chance of model overfitting

Feature selection

income	age	favorite color
10000	18	Black
50000	47	Blue
20000	40	Blue
30000	29	Green
20000	22	Purple

→

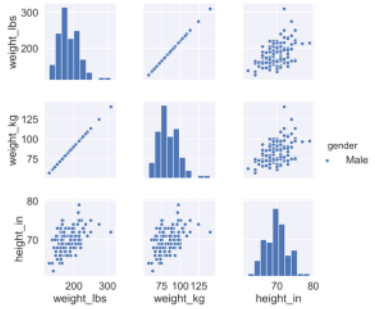
income	age
10000	18
50000	47
20000	40
30000	29
20000	22

```
insurance_df.drop('favorite color', axis=1)
```

Just make sure to pass the axis argument '1', to specify we're dropping a column instead of a row.

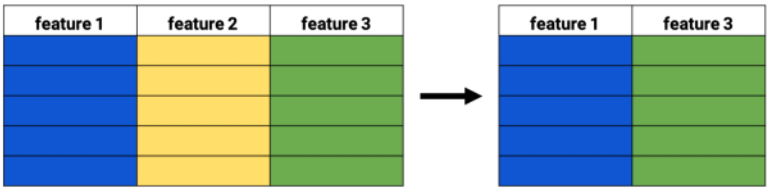
Building a pairplot on ANSUR data

```
sns.pairplot(ansur_df, hue="gender", diag_kind='hist')
```



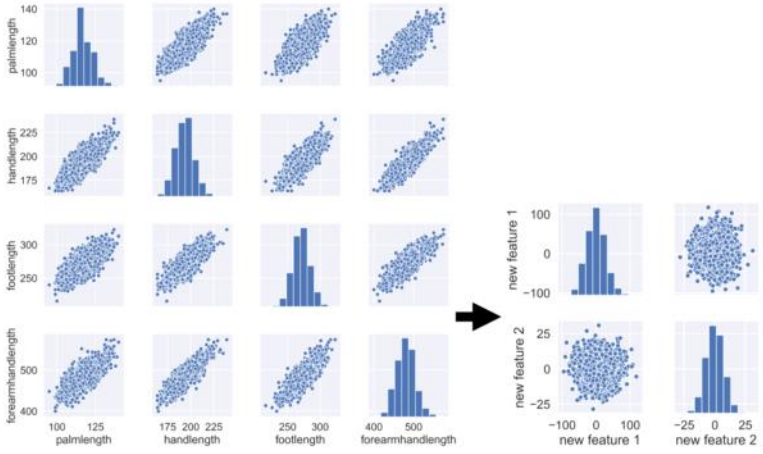
Seaborn's pairplot() is excellent to visually explore small to medium sized datasets.

Feature selection



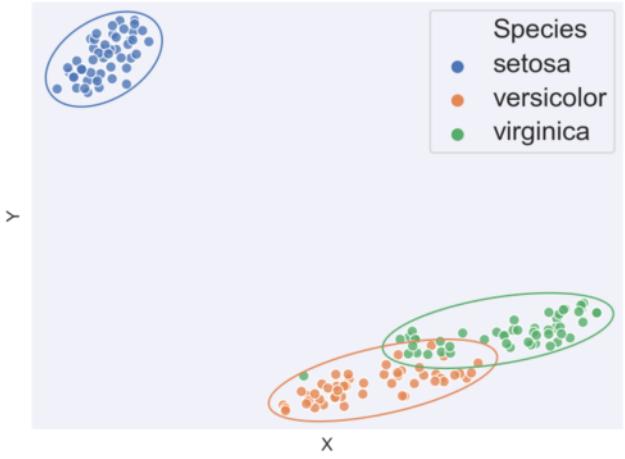
Feature extraction

Feature extraction - Example



In fact, we can reduce the number of dimensions in this ANSUR dataset sample from 4 to 2 with a technique called PCA or Principal Component Analysis and keep 96% of the variance in the data.

t-SNE on IRIS dataset



t-SNE on female ANSUR dataset

```
df.shape
```

```
(1986, 99)
```

body measurements dataset has 99 dimensions.

t-SNE on female ANSUR dataset

```
df.shape
```

```
(1986, 99)
```

```
non_numeric = ['BMI_class', 'Height_class',
               'Gender',  'Component', 'Branch']

df_numeric = df.drop(non_numeric, axis=1)

df_numeric.shape
```

```
(1986, 94)
```

Before we apply t-SNE we're going to remove all non-numeric columns from the dataset by passing a list with the unwanted column names to the pandas dataframe

We could use a trick like one-hot encoding to get around this but we'll be using a different approach here.

Fitting t-SNE

```
from sklearn.manifold import TSNE

m = TSNE(learning_rate=50)
```

High learning rates will cause the algorithm to be more adventurous in the configurations it tries out while low learning rates will cause it to be conservative.

Fitting t-SNE

```
from sklearn.manifold import TSNE

m = TSNE(learning_rate=50)
```

```
tsne_features = m.fit_transform(df_numeric)

tsne_features[1:4,:]
```

```
array([[-37.962185,  15.066088],
       [-21.873512,  26.334448],
       [ 13.97476 ,  22.590828]], dtype=float32)
```

This will project our high-dimensional dataset onto a NumPy array with two dimensions.

Assigning t-SNE features to our dataset

```
tsne_features[1:4,:]
```

```
array([[-37.962185,  15.066088],
       [-21.873512,  26.334448],
       [ 13.97476 ,  22.590828]], dtype=float32)
```

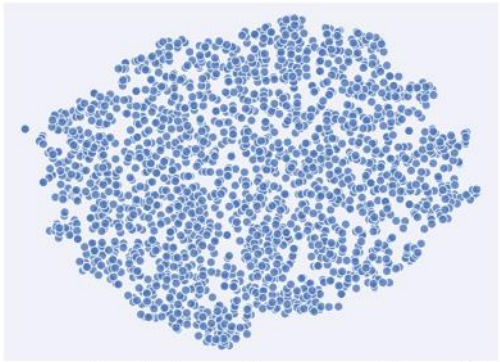
```
df['x'] = tsne_features[:,0]

df['y'] = tsne_features[:,1]
```

We'll assign these two dimensions back to our original dataset naming them 'x' and 'y'.

Plotting t-SNE

```
import seaborn as sns

sns.scatterplot(x="x", y="y", data=df)

plt.show()
```

Plotting t-SNE



The resulting plot shows one big cluster, and in a sense, this could have been expected.

There are no distinct groups of female body shapes with little in between,

instead, there is a more continuous distribution of body shapes, and thus, one big cluster.

Coloring points according to BMI category

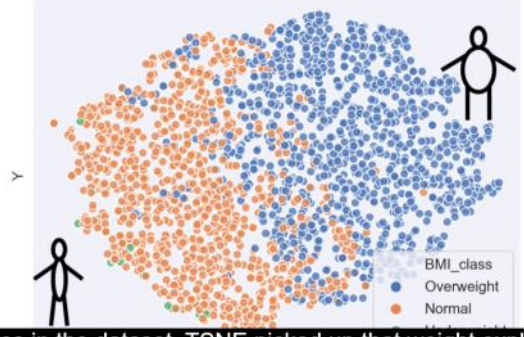
```
import seaborn as sns
import matplotlib.pyplot as plt

sns.scatterplot(x="x", y="y", hue='BMI_class', data=df)

plt.show()
```

The Body Mass Index or BMI is a method to categorize people into weight groups regardless of their height.

Coloring points according to BMI category

we'll be able to see that weight class indeed shows an interesting pattern.

Coloring points according to BMI category



From the 90+ features in the dataset, TSNE picked up that weight explains a lot of variance in the dataset and

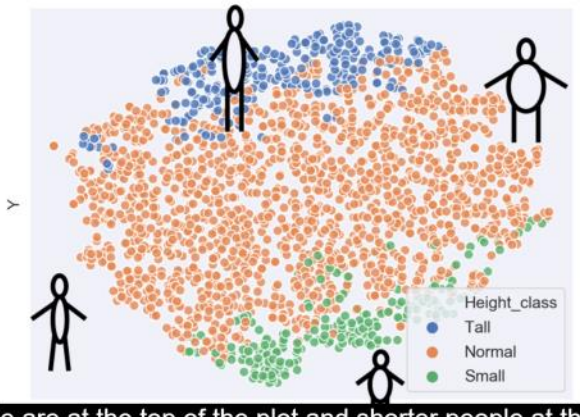
used that to spread out points along the x-axis, with underweight people on the left and overweight people on the right.

Coloring points according to height category

```
import seaborn as sns

import matplotlib.pyplot as plt

sns.scatterplot(x="x", y="y", hue='Height_class', data=df)

plt.show()
```

We've also added a column with height categories to the dataset.

Coloring points according to height category



Tall people are at the top of the plot and shorter people at the bottom.

In conclusion, t-SNE helped us to visually explore our dataset

Feature selection I, selecting for feature information

Models tend to overfit badly on high-dimensional data.

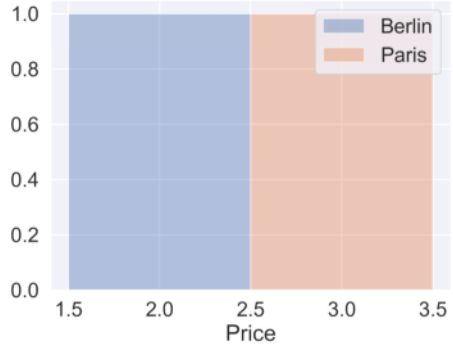
The solution is of course to reduce dimensionality, but which features should you drop?

In this chapter we'll look into detecting low quality features and how to remove them.

But first, we'll build some intuition on why models overfit.

From observation to pattern

City	Price
Berlin	2
Paris	3



It will just memorize the two training examples instead of deriving a general pattern for houses from each city.

If we want the model to generalize we need to give it more observations of house prices for each city.

Building a city classifier - data split

Separate the feature we want to predict from the ones to train the model on.

```
y = house_df['City']

X = house_df.drop('City', axis=1)
```

Perform a 70% train and 30% test data split

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)
```

Building a city classifier - model fit

Create a Support Vector Machine Classifier and fit to training data

```
from sklearn.svm import SVC

svc = SVC()

svc.fit(X_train, y_train)
```

Building a city classifier - predict

```
from sklearn.metrics import accuracy_score

print(accuracy_score(y_test, svc.predict(X_test)))
```

```
0.826
```

```
print(accuracy_score(y_train, svc.predict(X_train)))
```

If this accuracy is much higher than that on the test set we can conclude that the model didn't generalize well but simply memorized all training examples.

Building a city classifier - predict

```
from sklearn.metrics import accuracy_score

print(accuracy_score(y_test, svc.predict(X_test)))
```

```
0.826
```

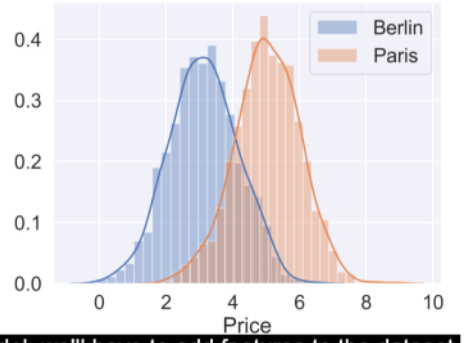
```
print(accuracy_score(y_train, svc.predict(X_train)))
```

```
0.832
```

Fortunately, this is not the case here.

Adding features

City	Price
Berlin	2.0
Berlin	3.1
Berlin	4.3
Paris	3.0
Paris	5.2
...	...



If we want to improve the accuracy of our model, we'll have to add features to the dataset, so

that in cases where the price of the house doesn't allow us to derive the location, something else will.

Adding features

City	Price	n_floors	n_bathroom	surface_m2
Berlin	2.0	1	1	190
Berlin	3.1	2	1	187
Berlin	4.3	2	2	240
Paris	3.0	2	1	170
Paris	5.2	2	2	290
...	...	...	...	...

Features like the number of floors, bathrooms, or surface area in the house all could be helpful.

However, with each feature that we add, we should also increase the number of observations of houses in our dataset.

In fact, to avoid overfitting the number of observations should increase exponentially with the number of features.

datasets this phenomenon is known as the curse of dimensionality.

In this lesson, we'll automate the selection of features that have sufficient variance and not too many missing values.

Creating a feature selector

```
print(ansur_df.shape)
```

```
(6068, 94)
```

```
from sklearn.feature_selection import VarianceThreshold

sel = VarianceThreshold(threshold=1)
```

To remove them, we can use one of Scikit-learn's built-in feature selection tools called VarianceThreshold().

Creating a feature selector

```
print(ansur_df.shape)
```

```
(6068, 94)
```


```
from sklearn.feature_selection import VarianceThreshold

sel = VarianceThreshold(threshold=1)
sel.fit(ansur_df)

mask = sel.get_support()
print(mask)
```

```
array([ True,  True, ..., False,  True])
```

get_support() method will give us a True or False value on whether each feature's variance is above the threshold or not.

We call this type of boolean array a mask,

and we can use this mask to reduce the number of dimensions in our dataset.

Applying a feature selector

```
print(ansur_df.shape)
```

```
(6068, 94)
```

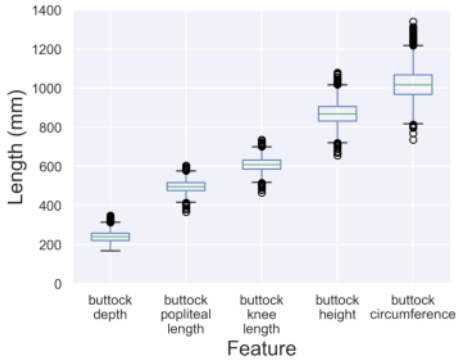
```
reduced_df = ansur_df.loc[:, mask]
print(reduced_df.shape)
```

```
(6068, 93)
```

In this case, our selector has reduced the number of features by just one.

Variance selector caveats

```
buttock_df.boxplot()
```



For this dataset higher values tend to have higher variances and we should therefore normalize the variance before using it for feature selection.

Normalizing the variance

```
from sklearn.feature_selection import VarianceThreshold

sel = VarianceThreshold(threshold=0.005)

sel.fit(ansur_df / ansur_df.mean())
```

```
mask = sel.get_support()
reduced_df = ansur_df.loc[:, mask]
print(reduced_df.shape)
```

```
(6068, 45)
```

Missing value selector

Name	Type 1	Type 2	Total	HP	Attack	Defense
Bulbasaur	Grass	Poison	318	45	49	49
Ivysaur	Grass	Poison	405	60	62	63
Venusaur	Grass	Poison	525	80	82	83
Charmander	Fire	NaN	309	39	52	43
Charmeleon	Fire	NaN	405	58	64	58

Another reason you might want to drop a feature is that it contains a lot of missing values.

Identifying missing values

```
pokemon_df.isna()
```

Name	Type 1	Type 2	Total	HP	Attack	Defense
False	False	False	False	False	False	False
False	False	False	False	False	False	False
False	False	False	False	False	False	False
False	False	True	False	False	False	False
False	False	True	False	False	False	False

isna() method we can identify them with a boolean value.

Counting missing values

```
pokemon_df.isna().sum()
```

```
Name          0
Type 1        0
Type 2      386
Total         0
HP            0
Attack        0
Defense       0
dtype: int64
```

Counting missing values

```
pokemon_df.isna().sum() / len(pokemon_df)
```

```
Name        0.00
Type 1      0.00
Type 2      0.48
Total       0.00
HP          0.00
Attack      0.00
Defense     0.00
dtype: float64
```

the dataframe we get a ratio of missing values between zero and one.

In this example it turns out that almost half the Pokemon don't have a value for the Type 2 feature.

Applying a missing value threshold

```
# Fewer than 30% missing values = True value
mask = pokemon_df.isna().sum() / len(pokemon_df) < 0.3
print(mask)
```

```
Name        True
Type 1      True
Type 2     False
Total       True
HP          True
Attack      True
Defense     True
```

Based on this ratio we can create a mask for features that have fewer missing values than a certain threshold.

Applying a missing value threshold

```
reduced_df = pokemon_df.loc[:, mask]

reduced_df.head()
```

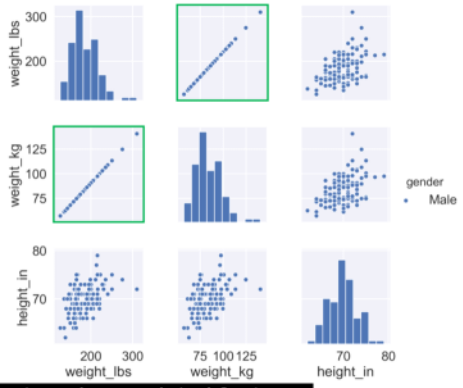
Name	Type 1	Total	HP	Attack	Defense
Bulbasaur	Grass	318	45	49	49
Ivysaur	Grass	405	60	62	63
Venusaur	Grass	525	80	82	83
Charmander	Fire	309	39	52	43
Charmeleon	Fire	405	58	64	58

When features have some missing values, but not too much, we could apply imputation to fill in the blanks.

individual properties, such as the variance they show or the proportion of missing values they have.

Pairwise correlation

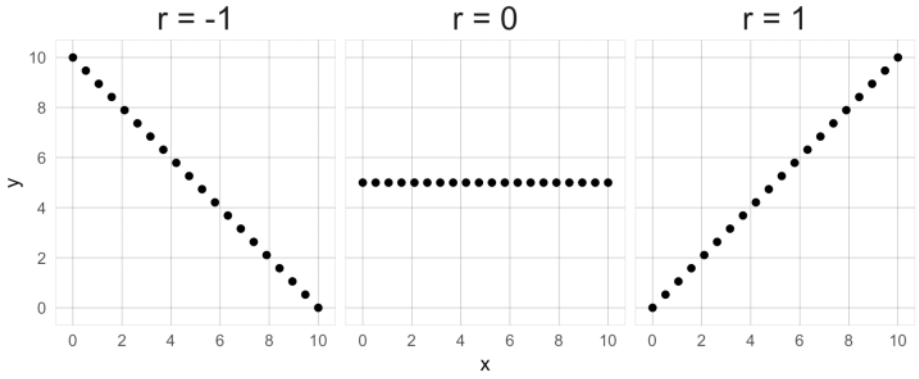
```
sns.pairplot(ansur, hue="gender")
```



It allowed us to visually identify strongly correlated features.

However, if we want to quantify the correlation between features, this method would fall short.

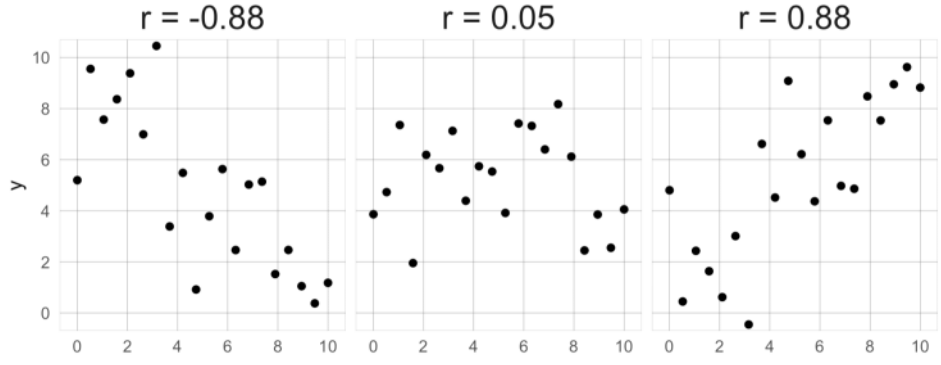
Correlation coefficient



The value of r always lies between minus one and plus one.

Minus one describes a perfectly negative correlation, zero describes no correlation at all and plus one stands for a perfect positive correlation.

Correlation coefficient



When the relation between two features shows more variance, as is usually the case in real-world data, the correlation coefficients will be a bit closer to zero.

Correlation matrix

```
weights_df.corr()
```

	weight_lbs	weight_kg	height_in
weight_lbs	1.00	1.00	0.47
weight_kg	1.00	1.00	0.47
height_in	0.47	0.47	1.00

If we call it on the dataset the pairplot was built on, we'd get a so-called correlation matrix.

Correlation matrix

```
weights_df.corr()
```

	weight_lbs	weight_kg	height_in
weight_lbs	1.00	1.00	0.47
weight_kg	1.00	1.00	0.47
height_in	0.47	0.47	1.00

In fact, it even shows every pairwise correlation

coefficient twice, since the correlation of A to B equals that of B to A.

	weight_lbs	weight_kg	height_in
weight_lbs	1.00	1.00	0.47
weight_kg	1.00	1.00	0.47
height_in	0.47	0.47	1.00

Perfectly correlated features, such as weight in kilograms and

weight in pounds marked in red here, get a correlation coefficient of one.

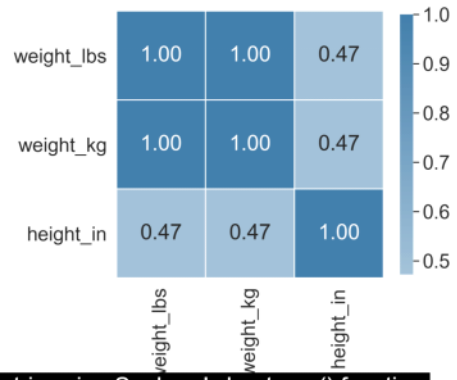
Meaning that if you know one feature, you can perfectly predict the other for this dataset.

	weight_lbs	weight_kg	height_in
weight_lbs	1.00	1.00	0.47
weight_kg	1.00	1.00	0.47
height_in	0.47	0.47	1.00

By definition, the diagonal in our correlation matrix shows a series of ones, telling us that, not surprisingly, each feature is perfectly correlated to itself.

Visualizing the correlation matrix

```
cmap = sns.diverging_palette(h_neg=10,
                             h_pos=240,
                             as_cmap=True)

sns.heatmap(weights_df.corr(), center=0,
            cmap=cmap, linewidths=1,
            annot=True, fmt=".2f")
```



We can visualize this simple correlation matrix using Seaborn's heatmap() function.

We can improve this plot further by removing duplicate and unnecessary information like the correlation coefficients of one on the diagonal.

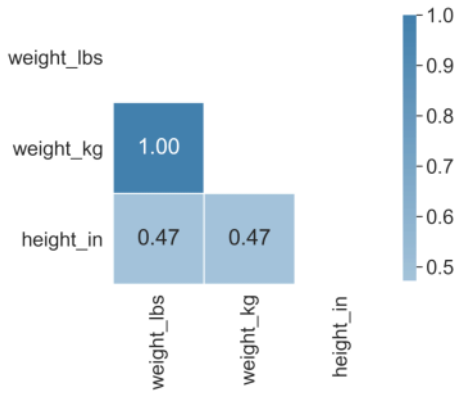
Visualizing the correlation matrix

```
corr = weights_df.corr()

mask = np.triu(np.ones_like(corr, dtype=bool))
```

We use NumPy's ones_like() function to create a matrix filled with True values with the same dimensions as our correlation matrix and then pass this to NumPy's triu(), for triangle upper, function to set all non-upper triangle values to False.

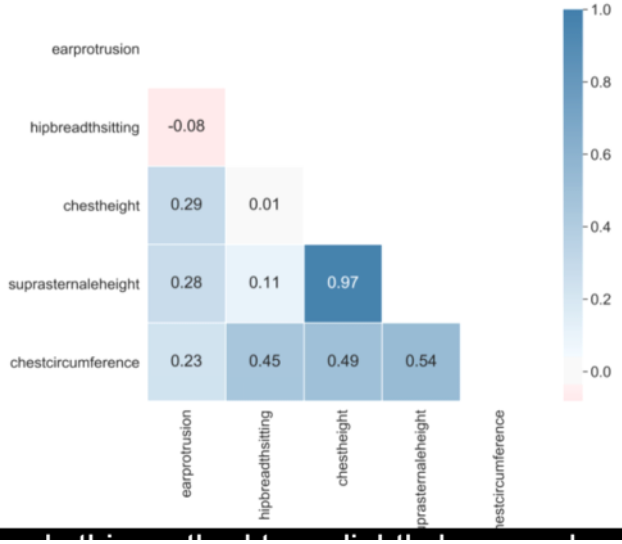
Visualizing the correlation matrix

```
sns.heatmap(weights_df.corr(), mask=mask,
            center=0, cmap=cmap, linewidths=1,
            annot=True, fmt=".2f")
```



When we pass this mask to the heatmap() function it will ignore the

Visualising the correlation matrix



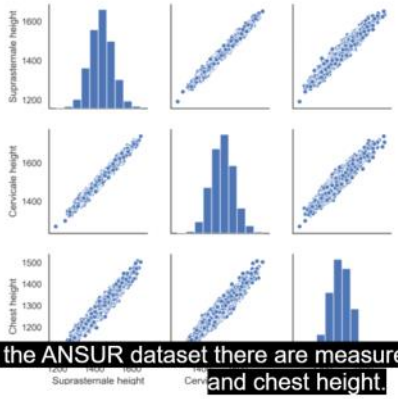
instantly spot that chest height is not correlated to the hip breadth while sitting, and that the suprasternale height is very strongly correlated to the chest height.

Features that are perfectly correlated to each other, with a correlation coefficient of one or minus one, bring no new information to a dataset but do add to the complexity.

So naturally, we would want to drop one of the two features that hold the same information.

In addition to this we might want to drop features that have correlation coefficients close to one or minus one if they are measurements of the same or similar things.

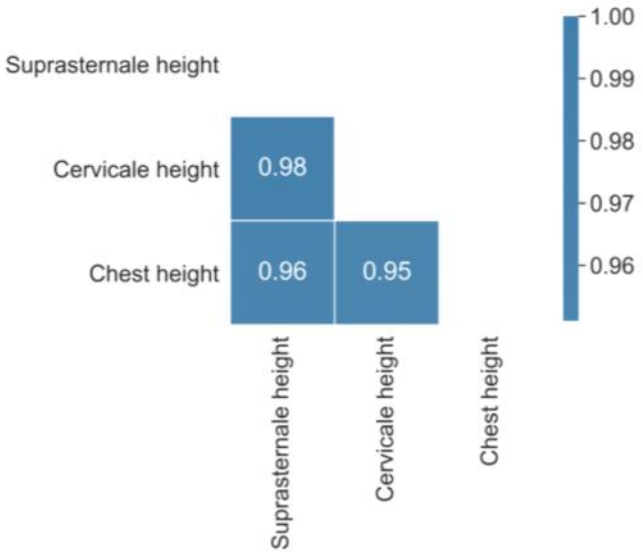
Highly correlated data



For example, in the ANSUR dataset there are measurements for suprasternale, cervicale and chest height.

The suprasternale and cervicale are two bones in the chest region so these three measurements always have very similar values.

Highly correlated features



We get correlation coefficients as high as 98%.

So for these features it too makes sense to keep only one.

Not just for simplicity's sake but also to avoid models to overfit on the small, probably meaningless, differences between these values.

If you are confident that dropping highly correlated features will not cause you to lose too much information, you can filter them out using a threshold value.

Removing highly correlated features

```
# Create positive correlation matrix
corr_df = chest_df.corr().abs()
# Create and apply mask
mask = np.triu(np.ones_like(corr_df, dtype=bool))
tri_df = corr_matrix.mask(mask)

tri_df
```

	Suprasternale height	Cervicale height	Chest height
Suprasternale height	NaN	NaN	NaN
Cervicale height	0.983033	NaN	NaN
Chest height	0.956111	0.951101	NaN

Removing highly correlated features

```
# Find columns that meet treshold
to_drop = [c for c in tri_df.columns if any(tri_df[c] > 0.95)]

print(to_drop)
```

```
['Suprasternale height', 'Cervicale height']
```

The reason we used the mask to set half of the matrix to NA values is

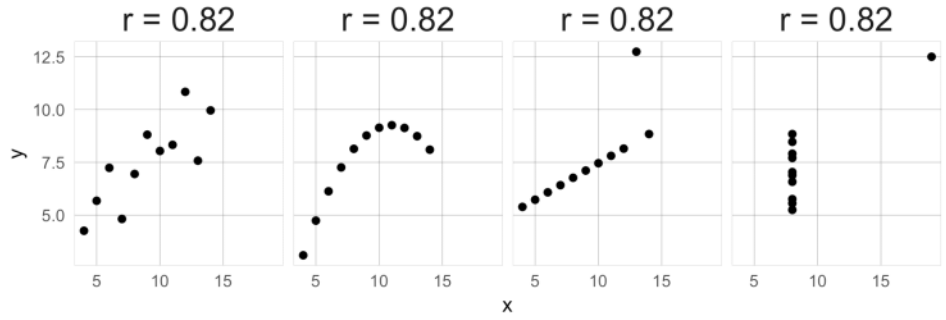
that we want to avoid removing both features when they have a strong correlation.

```
# Drop those columns
reduced_df = chest_df.drop(to_drop, axis=1)
```

Finally we drop the selected features from the dataframe with the .

drop() method.

Correlation caveats - Anscombe's quartet

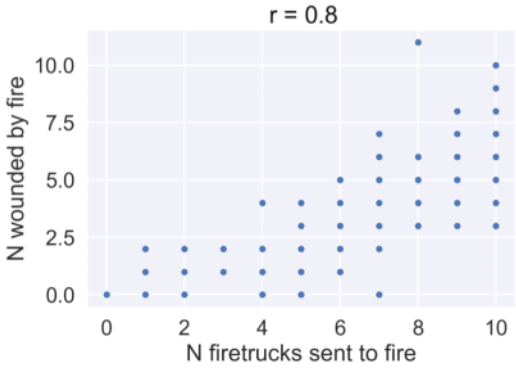


For example, the four datasets displayed here, known as

Anscombe's quartet, all have the same correlation coefficient of 0. 82.

Correlation caveats - causation

```
sns.scatterplot(x="N firetrucks sent to fire",
                y="N wounded by fire",data=fire_df)
```



A final thing to know about strong correlations is that they do not imply causation.

Feature selection II, selecting for model accuracy

Ansur dataset sample

Gender	chestdepth	handlength	neckcircumference	shoulderlength	earlength
Female	243	176	326	136	62
Female	219	177	325	135	58
Male	259	193	400	145	71
Male	253	195	380	141	62

Pre-processing the data

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

X_train_std = scaler.fit_transform(X_train)
```

Creating a logistic regression model

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

lr = LogisticRegression()
lr.fit(X_train_std, y_train)

X_test_std = scaler.transform(X_test)

y_pred = lr.predict(X_test_std)
print(accuracy_score(y_test, y_pred))
```

```
0.99
```

Inspecting the feature coefficients

```
print(lr.coef_)
```

```
array([[-3.  ,  0.14,  7.46,  1.22,  0.87]])
```

```
print(dict(zip(X.columns, abs(lr.coef_[0]))))
```

```
{'chestdepth': 3.0,
 'handlength': 0.14,
 'neckcircumference': 7.46,
 'shoulderlength': 1.22,
 'earlength': 0.87}
```

We can use the zip function to transform the output into a dictionary that shows which feature has which coefficient.

If we want to remove a feature from the initial dataset with as little effect on the predictions as possible, we should pick the one with the lowest coefficient, "handlength" in this case.

The fact that we standardized the data first makes sure that we can compare the coefficients to one another.

Features that contribute little to a model

```
X.drop('handlength', axis=1, inplace=True)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3)

lr.fit(scaler.fit_transform(X_train), y_train)

print(accuracy_score(y_test, lr.predict(scaler.transform(X_test))))
```

```
0.99
```

When we remove the "handlength" feature at the start of the process, our model accuracy remains unchanged at 99% while we did reduce our dataset's complexity.

We could repeat this process until we have the desired number of features remaining, but it turns out, there's a Scikit-learn function that does just that.

Recursive Feature Elimination

```
from sklearn.feature_selection import RFE

rfe = RFE(estimator=LogisticRegression(), n_features_to_select=2, verbose=1)
rfe.fit(X_train_std, y_train)
```

```
Fitting estimator with 5 features.
Fitting estimator with 4 features.
Fitting estimator with 3 features.
```

Dropping a feature will affect other feature's coefficients

RFE for "Recursive Feature Elimination" is a feature selection algorithm that can be wrapped around any model that produces feature coefficients or feature importance values.

Inspecting the RFE results

```
X.columns[rfe.support_]
```

```
Index(['chestdepth', 'neckcircumference'], dtype='object')
```

```
print(dict(zip(X.columns, rfe.ranking_)))
```

```
{'chestdepth': 1,
 'handlength': 4,
 'neckcircumference': 1,
 'shoulderlength': 2,
 'earlength': 3}
```

```
print(accuracy_score(y_test, rfe.predict(X_test_std)))
```

```
0.99
```

Once RFE is done we can check the support_ attribute that contains True/False values to see which features were kept in the dataset.

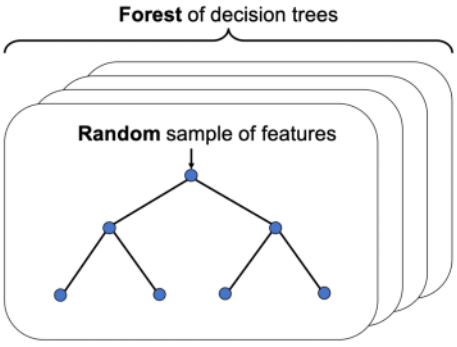
Using the zip function once more, we can also check out rfe's

ranking_ attribute to see in which iteration a feature was dropped.

Values of 1 mean that the feature was

kept in the dataset until the end while high values mean the feature was dropped early on.

Random forest classifier



It's an ensemble model that will pass different, random, subsets of features to a number of decision trees.

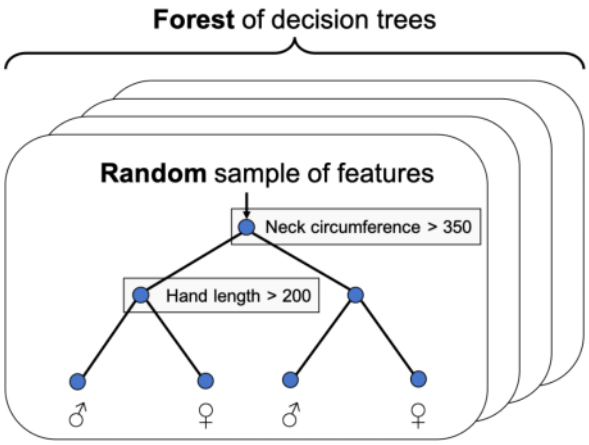
To make a prediction it will aggregate over the predictions of the individual trees.

Random forest classifier

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

rf = RandomForestClassifier()

rf.fit(X_train, y_train)

print(accuracy_score(y_test, rf.predict(X_test)))
```

```
0.99
```

Random forest classifier



Feature importance values

```
rf = RandomForestClassifier()

rf.fit(X_train, y_train)

print(rf.feature_importances_)
```

```
array([0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.04, 0.  , 0.01, 0.01,
       0.  , 0.  , 0.  , 0.  , 0.01, 0.01, 0.  , 0.  , 0.  , 0.  , 0.05,
       ...
       0.  , 0.14, 0.  , 0.  , 0.  , 0.06, 0.  , 0.  , 0.  , 0.  , 0.  ,
       0.  , 0.07, 0.  , 0.  , 0.01, 0.  ])
```

```
print(sum(rf.feature_importances_))
```

```
1.0
```

the random forest algorithm manages to calculate feature importance values.

These values can be extracted from a trained model using the feature_importances_ attribute.

Just like the coefficients produced by the logistic regressor, these feature importance values

can be used to perform feature selection, since for unimportant features they will be close to zero.

An advantage of these feature importance values over coefficients is that

they are comparable between features by default, since they always sum up to one.

Feature importance as a feature selector

```
mask = rf.feature_importances_ > 0.1

print(mask)
```

```
array([False, False, ..., True, False])
```

```
X_reduced = X.loc[:, mask]

print(X_reduced.columns)
```

```
Index(['chestheight', 'neckcircumference', 'neckcircumferencebase',
       'shouldercircumference'], dtype='object')
```

Remember dropping one weak feature can make other features relatively more or less important.

If you want to play safe and minimize the risk of dropping the wrong features,

you should not drop all least important features at once but rather one by one.

RFE with random forests

```
from sklearn.feature_selection import RFE

rfe = RFE(estimator=RandomForestClassifier(),
          n_features_to_select=6, verbose=1)

rfe.fit(X_train,y_train)
```

```
Fitting estimator with 94 features.
Fitting estimator with 93 features
...
Fitting estimator with 8 features.
Fitting estimator with 7 features.
```

```
print(accuracy_score(y_test, rfe.predict(X_test))
```

```
0.99
```

However, training the model once for each feature we want to drop can result in too much computational overhead.

0.99

To speed up the process we can pass the "step" parameter to RFE().

RFE with random forests

```
from sklearn.feature_selection import RFE

rfe = RFE(estimator=RandomForestClassifier(),
          n_features_to_select=6, step=10, verbose=1)

rfe.fit(X_train,y_train)
```

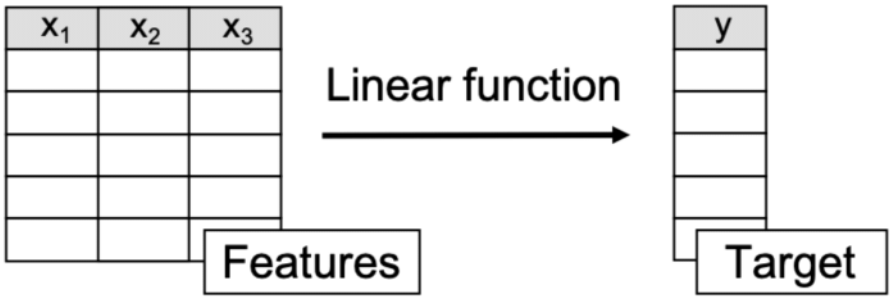
```
Fitting estimator with 94 features.
Fitting estimator with 84 features.
...
Fitting estimator with 24 features.
Fitting estimator with 14 features.
```

Here, we've set it to 10 so that on each iteration the 10 least important features are dropped.

```
print(X.columns[rfe.support_])
```

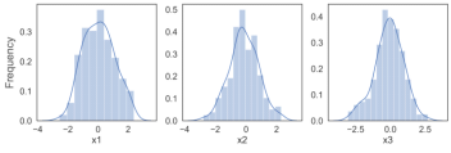
```
Index(['biacromialbreadth', 'handbreadth', 'handcircumference',
       'neckcircumference', 'neckcircumferencebase', 'shouldercircumference'], dtype='object')
```

Linear model concept

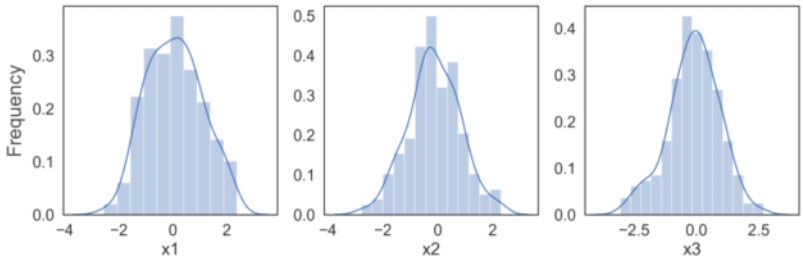


Creating our own dataset

x1	x2	x3
1.76	-0.37	-0.60
0.40	-0.24	-1.12
0.98	1.10	0.77
...	...	...



Creating our own dataset



Creating our own target feature:

$$y = 20 + 5x_1 + 2x_2 + 0x_3 + error$$

The 20 at the start is called the intercept; 5, 2 and 0 are the coefficients of our features, they determine how big an effect each has on the target.

The third feature has a coefficient of zero and will therefore have no effect on the target whatsoever.

It would be best to remove it from the dataset, since it could confuse a model and make it overfit.

Linear regression in Python

```
from sklearn.linear_model import LinearRegression

lr = LinearRegression()
lr.fit(X_train, y_train)

# Actual coefficients = [5 2 0]
print(lr.coef_)
```

```
[ 4.95  1.83 -0.05]
```

These are the three values we just set to 5, 2, and 0 and the model was able to estimate them pretty accurately.

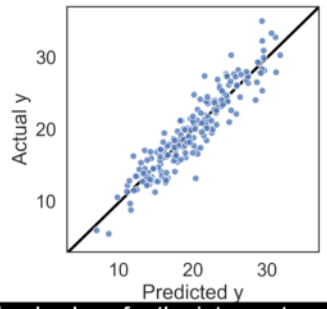
```
# Actual intercept = 20
print(lr.intercept_)
```

```
19.8
```

Same goes for the intercept.

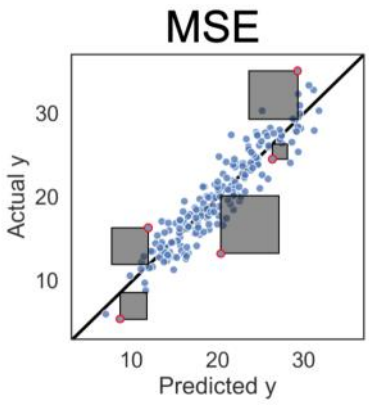
However, the third feature, which had no effect whatsoever, was estimated to have a small effect of -0.

Loss function: Mean Squared Error



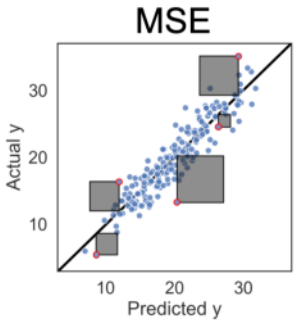
The model will try to find optimal values for the intercept and coefficients by minimizing a loss function.

Loss function: Mean Squared Error



This function contains the mean sum of the squared differences between actual and predicted values, the gray squares in the plot. Minimizing this MSE makes the model as accurate a possible. However, we don't want our model to be super accurate on the training set if that means it no longer generalizes to new data.

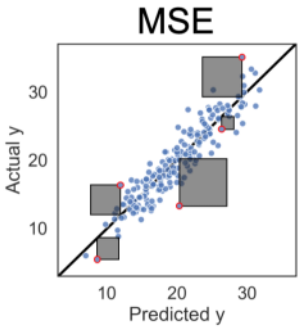
Adding regularization



$$+ \overbrace{\alpha(|\beta_1|+|\beta_2|+|\beta_3|)}^{\text{Regularization term}}$$

To avoid this we can introduce regularization.

Adding regularization



$$+ \overbrace{\alpha(|\beta_1|+|\beta_2|+|\beta_3|)}^{\text{Regularization term}}$$

Tries to make model simple

The strength of regularization can be tweaked with

[1] alpha, when it's too low the model might overfit, when it's too high the model might become too simple and inaccurate. One linear model that includes this type of regularization is called Lasso, for least absolute shrinkage and selection

Lasso regressor

```
from sklearn.linear_model import Lasso

la = Lasso()
la.fit(X_train, y_train)

# Actual coefficients = [5 2 0]
print(la.coef_)
```

```
[4.07 0.59 0.  ]
```

```
print(la.score(X_test, y_test))
```

```
0.861
```

Lasso regressor

```
from sklearn.linear_model import Lasso

la = Lasso(alpha=0.05)
la.fit(X_train, y_train)

# Actual coefficients = [5 2 0]
print(la.coef_)
```

```
[ 4.91  1.76 0.  ]
```

```
print(la.score(X_test, y_test))
```

```
0.974
```

LassoCV regressor

```
from sklearn.linear_model import LassoCV

lcv = LassoCV()

lcv.fit(X_train, y_train)

print(lcv.alpha_)
```

The LassoCV() class will use cross validation to try out different alpha settings and select the best one.

```
0.09
```

When we fit this model to our training data it will get an alpha_ attribute with the optimal value.

LassoCV regressor

```
mask = lcv.coef_ != 0

print(mask)
```

```
[ True True False ]
```

coefficient, we once again create a mask with True values for all non-zero coefficients.

```
reduced_X = X.loc[:, mask]
```

We can then apply it to our feature dataset X with the loc method.

Taking a step back

- Random forest is combination of decision trees.
- We can use combination of models for feature selection too.

Feature selection with LassoCV

```
from sklearn.linear_model import LassoCV

lcv = LassoCV()
lcv.fit(X_train, y_train)

lcv.score(X_test, y_test)
```

```
0.99
```

```
lcv_mask = lcv.coef_ != 0
sum(lcv_mask)
```

```
66
```

feature has a coefficient different from 0 we can see that this is the case for 66 out of 91 features.

Feature selection with random forest

```
from sklearn.feature_selection import RFE
from sklearn.ensemble import RandomForestRegressor

rfe_rf = RFE(estimator=RandomForestRegressor(),
             n_features_to_select=66, step=5, verbose=1)

rfe_rf.fit(X_train, y_train)
```

have it select the same number of features as the LassoCV() regressor did.

```
rf_mask = rfe_rf.support_
```

We can then use the support_ attribute of the fitted model to create rf_mask.

Feature selection with gradient boosting

```
from sklearn.feature_selection import RFE
from sklearn.ensemble import GradientBoostingRegressor

rfe_gb = RFE(estimator=GradientBoostingRegressor(),
             n_features_to_select=66, step=5, verbose=1)

rfe_gb.fit(X_train, y_train)
gb_mask = rfe_gb.support_
```

Combining the feature selectors

```
import numpy as np

votes = np.sum([lcv_mask, rf_mask, gb_mask], axis=0)

print(votes)
```

```
array([3, 2, 2, ..., 3, 0, 1])
```

If we want to make sure we don't lose any information, we could select all features with at least one vote.

```
mask = votes >= 2
```

In this example we chose to have at least two models voting for a feature in order to keep it.

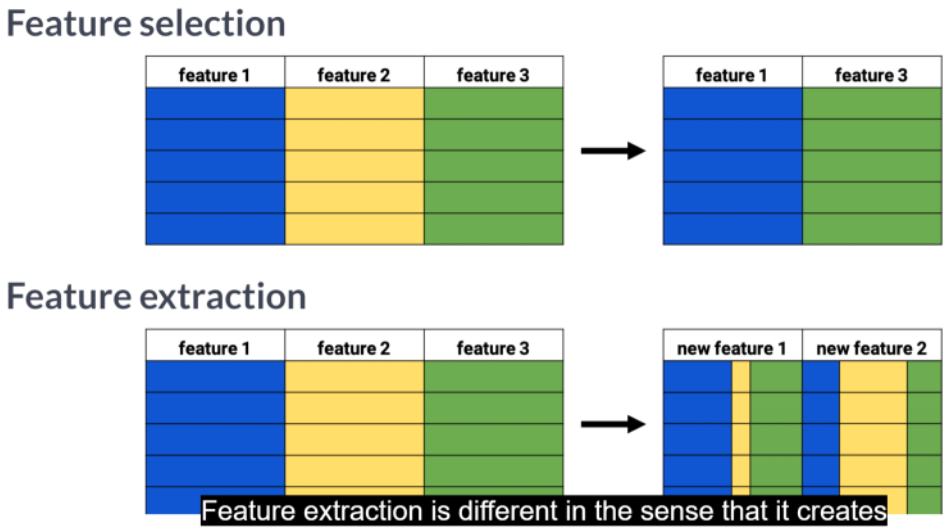
All that is left now is to actually implement the dimensionality reduction.

```
reduced_X = X.loc[:, mask]
```

We do that with the loc method of our feature dataframe X.

Feature extraction

This comes down to calculating new features based on the existing ones while trying to lose as little information as possible.



Feature extraction is different in the sense that it creates new features, which are in fact combinations of the original ones.

There are powerful algorithms that will calculate the new features in a way that as much information as possible is preserved, but before we get into those, let's look at more simple feature extraction.

Feature generation - BMI

```
df_body['BMI'] = df_body['Weight kg'] /  df_body['Height m'] ** 2
```

Weight kg	Height m	BMI
81.5	1.776	25.84
72.6	1.702	25.06
92.9	1.735	30.86

Feature generation - BMI

```
df_body.drop(['Weight kg', 'Height m'], axis=1)
```

BMI
25.84
25.06
30.86

Feature generation - averages

left leg mm	right leg mm
882	885
870	869
901	900

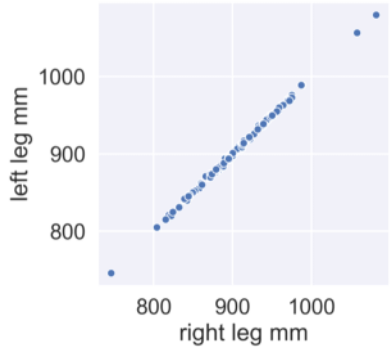
```
leg_df['leg mm'] = leg_df[['right leg mm', 'left leg mm']].mean(axis=1)
```

Feature generation - averages

```
leg_df.drop(['right leg mm', 'left leg mm'], axis=1)
```

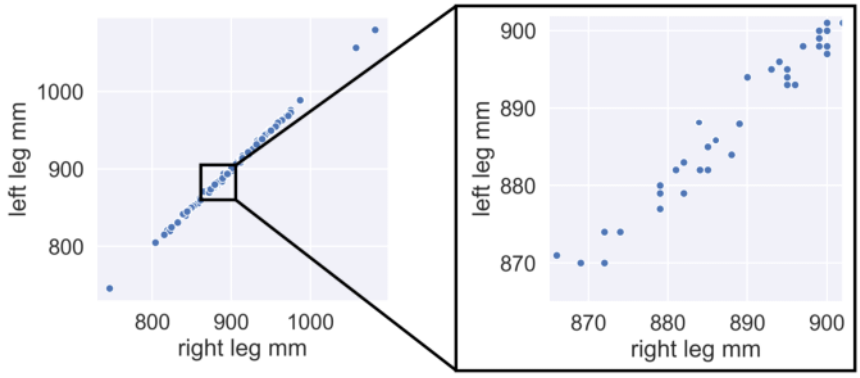
leg mm
883.5
869.5
900.5

Cost of taking the average



Taking the average of two features comes with the cost of losing some information.

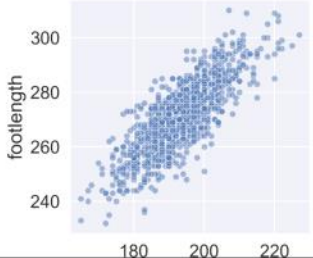
Cost of taking the average



We can now see the differences between both features more clearly.

Intro to PCA

```
sns.scatterplot(data=df, x='handlength', y='footlength')
```



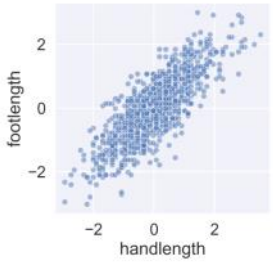
Now let's take a step back and look at a different data sample with hand lengths versus feet lengths.

Instead of taking the mean of both features we'll explore an alternative technique.

Intro to PCA

```
scaler = StandardScaler()
df_std = pd.DataFrame(scaler.fit_transform(df), columns = df.columns)
```

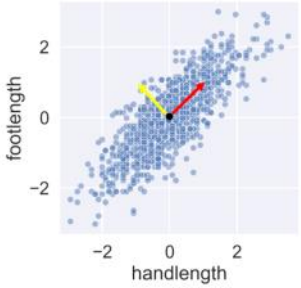
For this technique it's important to scale the features first, so that their values are easier to compare.



The strongest pattern in this dataset is that people with big feet also tend to have big hands.

People with a positive value for this vector have relatively long hands and feet, and people with a negative value have relatively short ones.

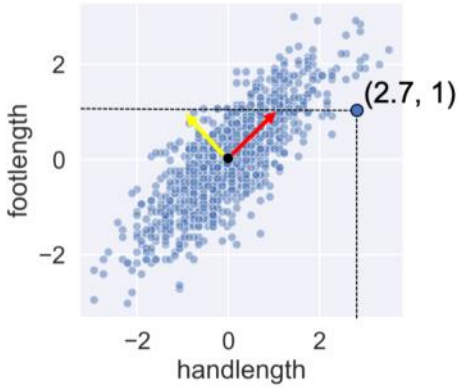
We could add a second vector perpendicular to the first one to account for the rest of the variance in this dataset.



People with a positive value for this second vector have relatively long feet compared to their hand length and people with a negative value have relatively big hands.

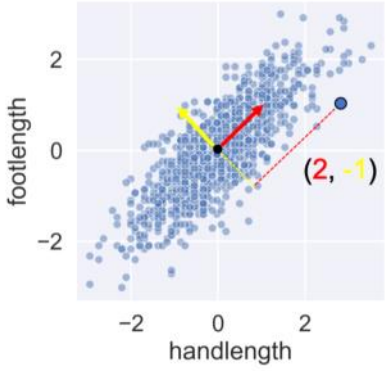
Every point in this dataset could be described by multiplying and then summing these two perpendicular vectors.

PCA concept



7 and 1 in the original hand length versus foot length reference system.

PCA concept



In this case it would be 2 times the red vector and minus 1 times the yellow vector.

red one is most important as it is aligned with the biggest source of variance in the data.

But before we do this, we have to scale the values with the StandardScaler().

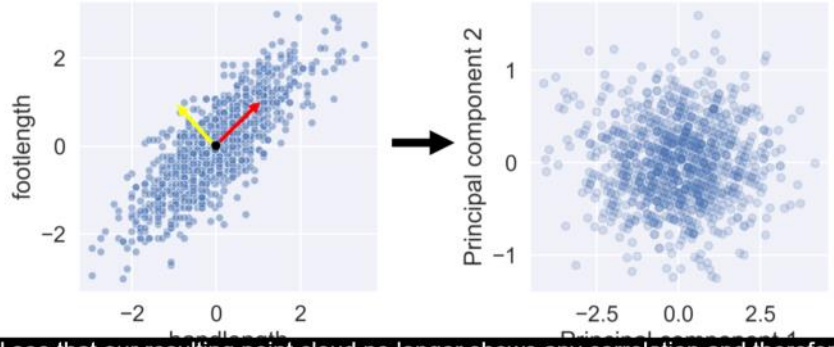
PCA can really underperform if you don't do this.

Calculating the principal components

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
std_df = scaler.fit_transform(df)
from sklearn.decomposition import PCA

pca = PCA()
print(pca.fit_transform(std_df))
```

```
[[-0.08320426 -0.12242952]
 [ 0.31478004  0.57048158]
 ...
 [-0.5609523   0.13713944]
 [-0.0448304  -0.37898246]]
```

PCA removes correlation



you'll see that our resulting point cloud no longer shows any correlation and therefore, no more duplicate information.

dataset we would also have to add a third principal component if we don't want to lose any information.

You could describe a 100 feature dataset with 100 principal components.

The components are much harder to understand than the original features.

The answer lies in the fact that the components share no

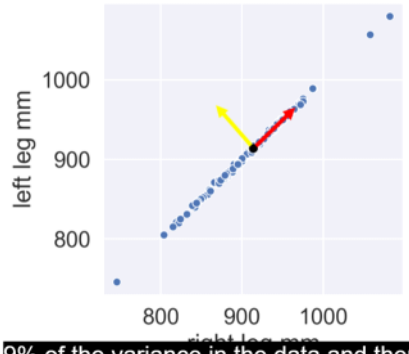
duplicate information and that they are ranked from most to least important.

Principal component explained variance ratio

```
from sklearn.decomposition import PCA

pca = PCA()

pca.fit(std_df)

print(pca.explained_variance_ratio_)
```

```
array([0.90, 0.10])
```

PCA for dimensionality reduction




```
print(pca.explained_variance_ratio_)
```

```
array([0.9997, 0.0003])
```

9% of the variance in the data and therefore it would clearly make sense to drop the second component.

PCA for dimensionality reduction

```
pca = PCA()

pca.fit(ansur_std_df)

print(pca.explained_variance_ratio_)
```

```
array([0.44, 0.18, 0.04, 0.03, 0.02, 0.02, 0.02, 0.01, 0.01, 0.01, 0.01,
       0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01,
       0.01, 0.01, 0.01, 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
       ...
       0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
       0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  , 0.  ,
       0.  , 0.  , 0.  , 0.  , 0.  , 0.  ])
```

When we fit pca to this data we'll find that the 2 first components explain 44 and 18% of the variance in the data.

PCA for dimensionality reduction

```
pca = PCA()

pca.fit(ansur_std_df)

print(pca.explained_variance_ratio_.cumsum())
```

```
array([0.44, 0.62, 0.66, 0.69, 0.72, 0.74, 0.76, 0.77, 0.79, 0.8 , 0.81,
       0.82, 0.83, 0.84, 0.85, 0.86, 0.87, 0.87, 0.88, 0.89, 0.89, 0.9 ,
       0.9 , 0.91, 0.92, 0.92, 0.92, 0.93, 0.93, 0.94, 0.94, 0.94, 0.95,
       ...
       0.99, 0.99, 0.99, 0.99, 0.99, 1.  , 1.  , 1.  , 1.  , 1.  , 1.  ,
       1.  , 1.  , 1.  , 1.  , 1.  , 1.  , 1.  , 1.  , 1.  , 1.  , 1.  ,
       1.  , 1.  , 1.  , 1.  , 1.  , 1.  , 1.  ])
```

Using just the first two components would still allow us to keep 62% of the variance in the

data whereas we would have to use 10 components if we would want to keep 80% of the variance.

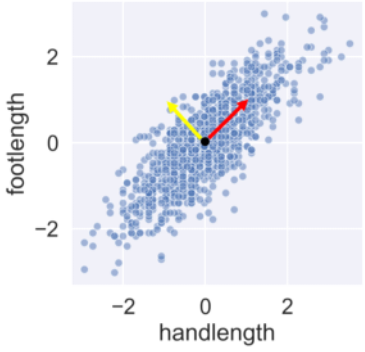
Understanding the components

```
print(pca.components_)
```

```
array([[  0.71, 0.71],
       [ -0.71, 0.71]])
```

PC 1 = 0.71 x Hand length + 0.71 x Foot length

PC 2 = -0.71 x Hand length + 0.71 x Foot length



The features that have the biggest positive or negative effects on a component can then be used to add a meaning to that component.

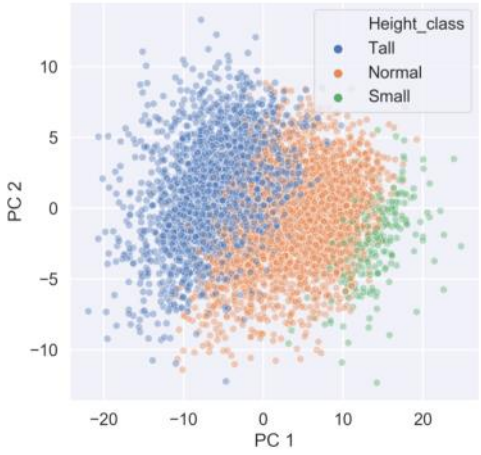
In the example shown here the effects of the features on the first components are positive and equally strong at 0.

So the first component is affected just as much by hand as foot length.

However, the second component is negatively affected by hand length.

So people who score high for the second component have short hands compared to their feet.

PCA for data exploration



PCA in a pipeline

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline

pipe = Pipeline([
        ('scaler', StandardScaler()),
        ('reducer', PCA())])
pc = pipe.fit_transform(ansur_df)

print(pc[:,:2])
```

```
array([[-3.46114925,  1.5785215 ],
       [ 0.90860615,  2.02379935],
       ...,
       [10.7569818 , -1.40222755],
       [ 7.64802025,  1.07406209]])
```

Checking the effect of categorical features

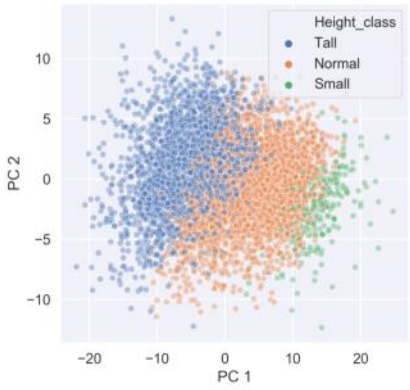
```
print(ansur_categories.head())
```

Branch	Component	Gender	BMI_class	Height_class
Combat Arms	Regular Army	Male	Overweight	Tall
Combat Support	Regular Army	Male	Overweight	Normal
Combat Support	Regular Army	Male	Overweight	Normal
Combat Service Support	Regular Army	Male	Overweight	Normal
Combat Service Support	Regular Army	Male	Overweight	Tall

PCA is not the preferred algorithm to reduce the dimensionality of categorical datasets

Checking the effect of categorical features

```
ansur_categories['PC 1'] = pc[:,0]
ansur_categories['PC 2'] = pc[:,1]
sns.scatterplot(data=ansur_categories,
                x='PC 1', y='PC 2',
                hue='Height_class', alpha=0.4)
```

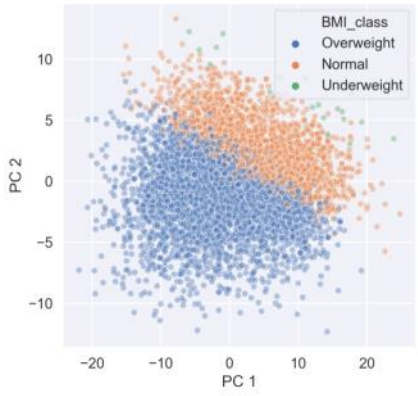


Checking the effect of categorical features

```
sns.scatterplot(data=ansur_categories,
                x='PC 1', y='PC 2',
                hue='Gender', alpha=0.4)
```

Checking the effect of categorical features

```
sns.scatterplot(data=ansur_categories,
                x='PC 1', y='PC 2',
                hue='BMI_class', alpha=0.4)
```



PCA in a model pipeline

```
pipe = Pipeline([
        ('scaler', StandardScaler()),
        ('reducer', PCA(n_components=3)),
        ('classifier', RandomForestClassifier())])
pipe.fit(X_train, y_train)
print(pipe.steps[1])
```

```
('reducer',
 PCA(copy=True, iterated_power='auto', n_components=3, random_state=None,
   svd_solver='auto', tol=0.0, whiten=False))
```

PCA in a model pipeline

```
pipe.steps[1][1].explained_variance_ratio_.cumsum()
```

```
array([0.56, 0.69, 0.74])
```

```
print(pipe.score(X_test, y_test))
```

```
0.986
```

Setting an explained variance threshold

```
pipe = Pipeline([
        ('scaler', StandardScaler()),
        ('reducer', PCA(n_components=0.9))])
```

9 it will make sure to select enough components to explain 90% of the variance.

```
# Fit the pipe to the data
pipe.fit(poke_df)

print(len(pipe.steps[1][1].components_))
```

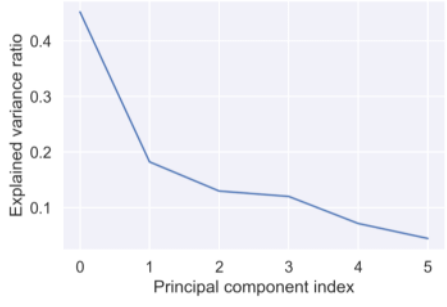
```
5
```

Fact is, there is not a single right answer to the question "how many components should I keep?"

There is, however, a trick that can help you find a good balance.

An optimal number of components

```
pipe.fit(poke_df)

var = pipe.steps[1][1].explained_variance_ratio_

plt.plot(var)

plt.xlabel('Principal component index')
plt.ylabel('Explained variance ratio')
plt.show()
```

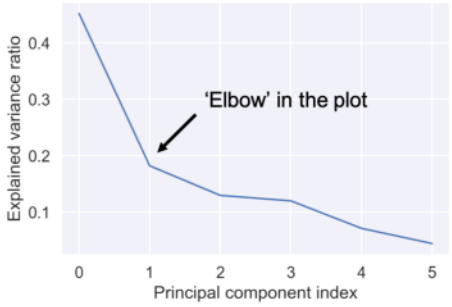


fitted PCA instance, you'll get to see that most of the explained variance is concentrated in the first few components.

the explained variance ratio per component starts to level out quite abruptly.

An optimal number of components

```
pipe.fit(poke_df)

var = pipe.steps[1][1].explained_variance_ratio_

plt.plot(var)

plt.xlabel('Principal component index')
plt.ylabel('Explained variance ratio')
plt.show()
```



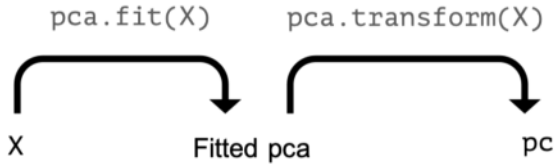
the 'elbow' in the plot.

And it typically gives you a good starting point for the number of components to keep.

Do note that the x-axis shows you the index of the components and not the total number.

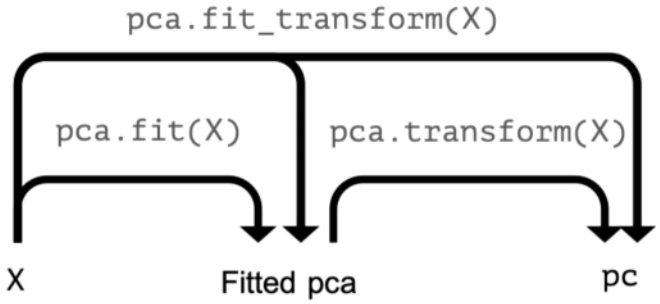
So since the elbow is at the component with index 1 here, we'd select 2 components.

PCA operations



Up until now, we've seen how you can use PCA to go from an input feature dataset X to a NumPy array of principal components pc, either by first fitting pca to the data and then transforming that data in two operations

PCA operations



or in one go with the .

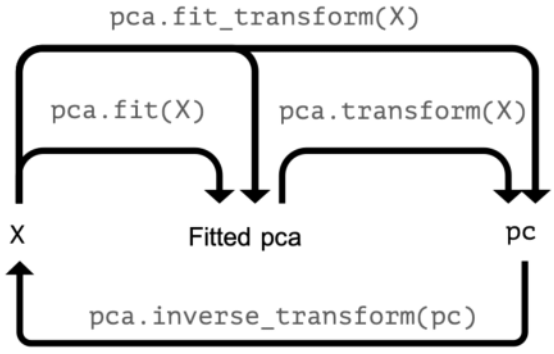
fit_transform() method.

A final trick that I'd like to teach you, is how to go

back from the principal components to the original feature space.

inverse_transform() method on the principal component array.

PCA operations



Because you typically lose information going from left to right in this overview, you'll

see that the result from going back to the original feature space will have changed somewhat.

Compressing images

An application where this is relevant is image compression.

Compressing images

```
print(X_test.shape)
```

```
(15, 2914)
```

62 x 47 pixels = 2914 grayscale values

```
print(X_train.shape)
```

```
(1333, 2914)
```

Compressing images

```
pipe = Pipeline([
        ('scaler', StandardScaler()),
        ('reducer', PCA(n_components=290))])
pipe.fit(X_train)
pc = pipe.fit_transform(X_test)

print(pc.shape)
```

Compressing images

```
pipe = Pipeline([
        ('scaler', StandardScaler()),
        ('reducer', PCA(n_components=290))])
pipe.fit(X_train)
pc = pipe.fit_transform(X_test)

print(pc.shape)
```

```
(15, 290)
```

Rebuilding images

```
pc = pipe.transform(X_test)

print(pc.shape)
```

```
(15, 290)
```

```
X_rebuilt = pipe.inverse_transform(pc)

print(X_rebuilt.shape)
```

```
(15, 2914)
```

```
img_plotter(X_rebuilt)
```

Rebuilding images

What you've learned

- Why dimensionality reduction is important & when to use it
- Feature selection vs extraction
- High dimensional data exploration with t-SNE & PCA
- Use models to find important features
- Remove unimportant ones

learn on this topic, for instance, we've been focusing on numerical data in this course and

this has come at the cost of giving categorical features and text data a bit of a cold treatment.

I encourage you to have a look at the specialized techniques for those data types.