

# Dimensionality Reduction in Python

How feature extraction techniques can reduce dimensionality



Black Raven (James Ng)

Jun 27 · 29 min read



This is a tutorial to share what I have learnt in Dimensionality Reduction in Python, capturing the learning objectives as well as my personal notes. The course is taught by Jerone Boeye from DataCamp (pdf notes here), and it includes 4 chapters:

Chapter 1. Exploring high dimensional data

Chapter 2. Feature selection I, selecting for feature information

Chapter 3. Feature selection II, selecting for model accuracy

Chapter 4. Feature extraction





Photo by Aditya Chinchure on Unsplash

High-dimensional datasets can be overwhelming and leave you not knowing where to start. Typically, you'd visually explore a new dataset first, but when you have too many dimensions the classical approaches will seem insufficient. Fortunately, there are visualization techniques designed specifically for high dimensional data and you'll be introduced to these in this course.

After exploring the data, you'll often find that many features hold little information because they don't show any variance or because they are duplicates of other features. You'll learn how to detect these features and drop them from the dataset so that you can focus on the informative ones. In a next step, you might want to build a model on these features, and it may turn out that some don't have any effect on the thing you're trying to predict. You'll learn how to detect and drop these irrelevant features too, in order to reduce dimensionality and thus complexity.

Finally, you'll learn how feature extraction techniques can reduce dimensionality for you through the calculation of uncorrelated principal components.

• • •

## Chapter 1. Exploring high dimensional data

You'll be introduced to the concept of dimensionality reduction and will learn when and why this is important. You'll learn the difference between feature selection and feature extraction and will apply both techniques for data exploration. The chapter ends with a lesson on t-SNE, a powerful feature extraction technique that will allow you to visualize a high-dimensional dataset.

### Introduction

Dataset with more than 10 columns are considered high dimensional data.

### Finding the number of dimensions in a dataset

A larger sample of the Pokemon dataset has been loaded for you as the Pandas dataframe `pokemon_df`.

How many dimensions, or columns are in this dataset?

Answer: 7 dimensions, each Pokemon is described by 7 features.

```
In [1]: pokemon_df.shape
Out[1]: (160, 7)
```

## Removing features without variance

A sample of the Pokemon dataset has been loaded as `pokemon_df`. To get an idea of which features have little variance, calculate the summary statistics of this sample using `.describe()`. Then adjust the code to create a smaller and useful dataset.

For the number\_cols, Generation column has '1' in all 160 rows.

```
In [1]: pokemon_df.describe()
Out[1]:
          HP      Attack     Defense   Generation
count  160.00000  160.00000  160.000000      160.0
mean   64.61250   74.98125   70.175000      1.0
std    27.92127   29.18009   28.883533      0.0
min    10.00000   5.000000   5.000000      1.0
25%   45.00000   52.00000   50.000000      1.0
50%   60.00000   71.00000   65.000000      1.0
75%   80.00000   95.00000   85.000000      1.0
max   250.00000  155.00000  180.000000      1.0
```

```
# Remove the feature without variance from this list
number_cols = ['HP', 'Attack', 'Defense']
```

For the non\_number\_cols, Legendary column has 'False' in all 160 rows.

```
In [6]: pokemon_df[['Name', 'Type', 'Legendary']].describe()
Out[6]:
          Name     Type  Legendary
count      160       160        160
unique     160        15         1
top       Abra     Water      False
freq        1         31        160
```

```
# Remove the feature without variance from this list
non_number_cols = ['Name', 'Type']

# Create a new dataframe by subselecting the chosen features
df_selected = pokemon_df[number_cols + non_number_cols]

# Prints the first 5 lines of the new dataframe
print(df_selected.head())
```

<script.py> output:						
	HP	Attack	Defense		Name	Type
0	45	49	49		Bulbasaur	Grass
1	60	62	63		Ivysaur	Grass
2	80	82	83		Venusaur	Grass
3	80	100	123	VenusaurMega	Venusaur	Grass
4	39	52	43		Charmander	Fire

All Pokemon in this dataset are non-legends and from generation one so you could choose to drop those two features.

## Feature selection vs feature extraction

Why reduce dimensionality?

- dataset will be less complex
- dataset will take up less storage space
- dataset will require less computation time
- dataset will have lower chance of model overfitting

## Feature selection

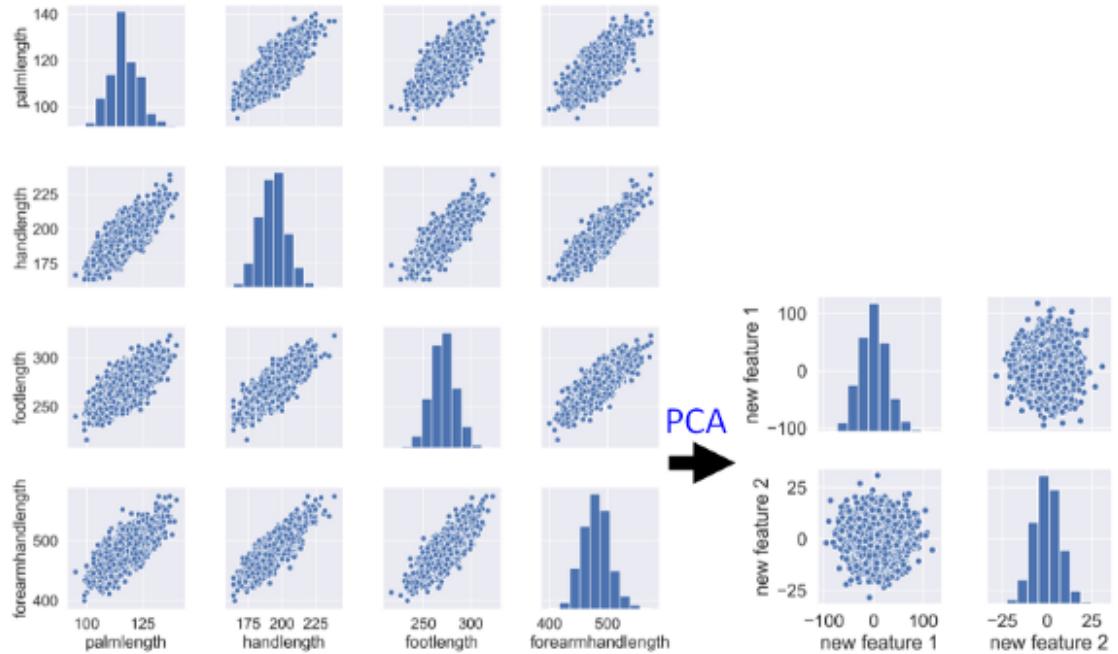


## Feature extraction





## Feature extraction - Example



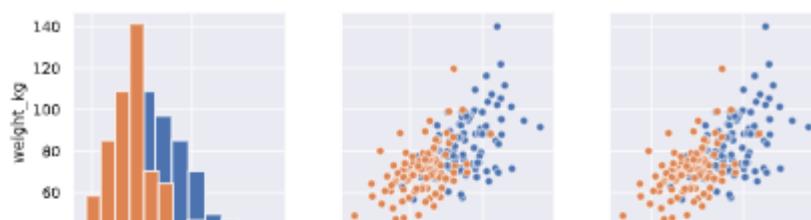
### Visually detecting redundant features

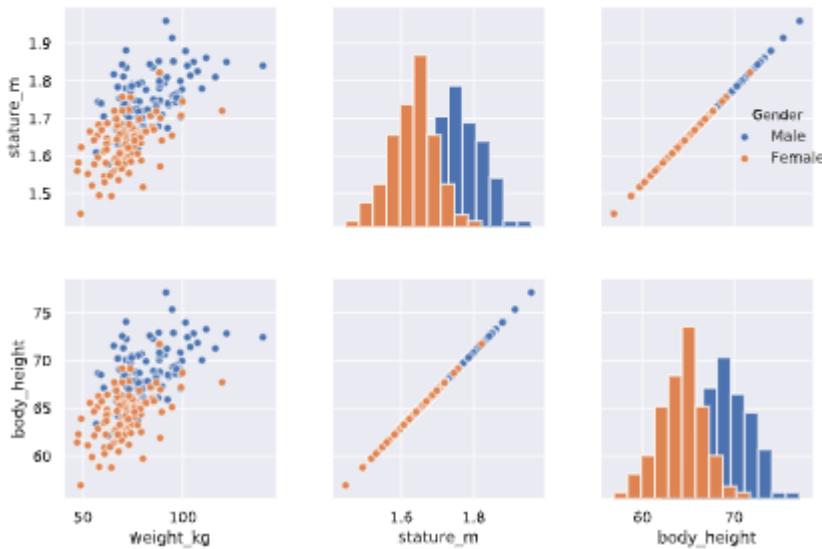
Data visualization is a crucial step in any data exploration. Let's use Seaborn to explore some samples of the US Army ANSUR body measurement dataset.

Two data samples are in `ansur_df_1` and `ansur_df_2`.

Seaborn has been imported as `sns`.

```
# Create a pairplot and color the points using the 'Gender' feature
sns.pairplot(ansur_df_1, hue='Gender', diag_kind='hist')
plt.show()
```



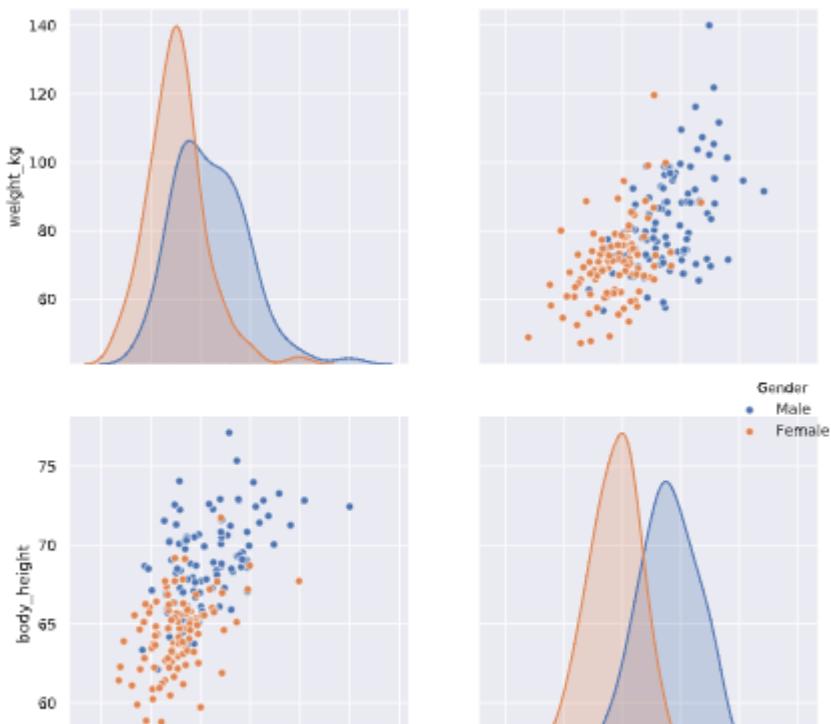


Two features are basically duplicates, remove one of them from the dataset.

```
# Remove one of the redundant features
reduced_df = ansur_df_1.drop('stature_m', axis=1)

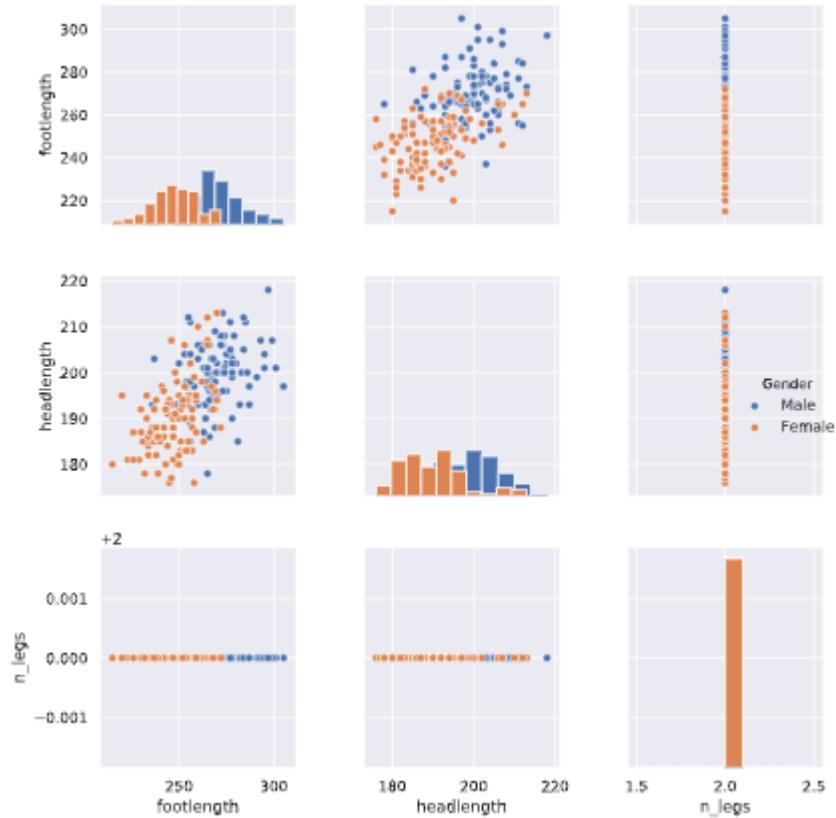
# Create a pairplot and color the points using the 'Gender' feature
sns.pairplot(reduced_df, hue='Gender')

# Show the plot
plt.show()
```





```
# Create a pairplot and color the points using the 'Gender' feature
sns.pairplot(ansur_df_2, hue='Gender', diag_kind='hist')
plt.show()
```



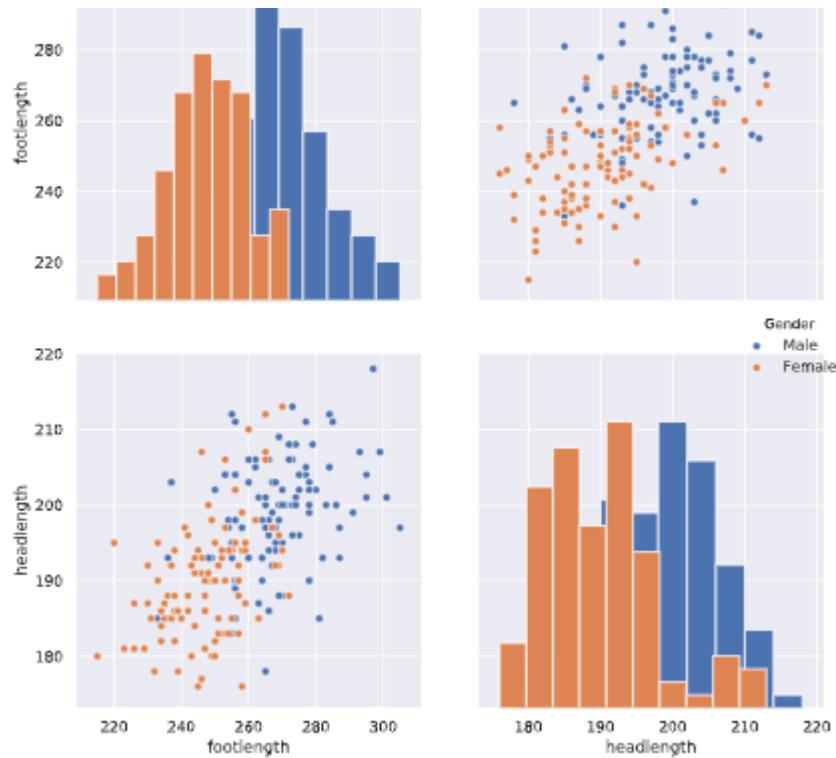
One feature has no variance, remove it from the dataset.

```
# Remove the redundant feature
reduced_df = ansur_df_2.drop('n_legs', axis=1)

# Create a pairplot and color the points using the 'Gender' feature
sns.pairplot(reduced_df, hue='Gender', diag_kind='hist')

# Show the plot
plt.show()
```





The body height (inches) and stature (meters) hold the same information in a different unit + all the individuals in the second sample have two legs.

## Advantage of feature selection

What advantage does feature selection have over feature extraction?

Answer: The selected features remain unchanged, and are therefore easy to interpret.

Extracted features can be quite hard to interpret.

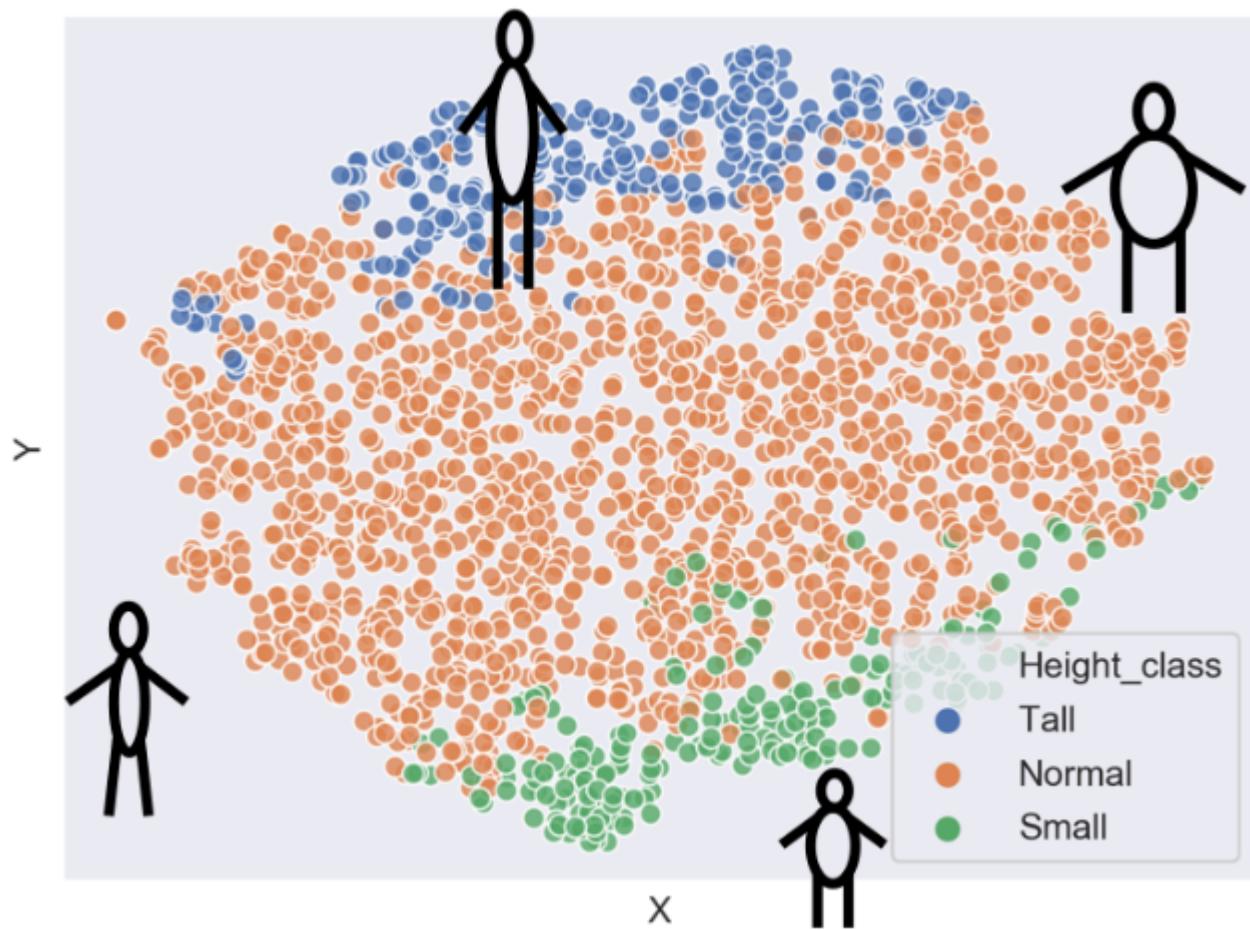
## t-SNE visualization of high-dimensional data

t-distributed Schohastic Neighur Embedding

To maximise distance in 2D space between observations that are most different in a high dimensional space. Observations that are similar will be clustered together.

High learning rate will cause the algorithm to be more adventurous, while low learning rate will cause it to be more conservative. Learning rate is usually between 10 to 1000 range.





## t-SNE intuition

t-SNE is super powerful. What is a good use case to use t-SNE?

Answer: When you want to visually explore the patterns in a high dimensional dataset.

## Fitting t-SNE to the ANSUR data

t-SNE is a great technique for visual exploration of high dimensional datasets. In this exercise, you'll apply it to the ANSUR dataset. You'll remove non-numeric columns from the dataset `df` and fit `TSNE` to his numeric dataset.

```
# Non-numerical columns in the dataset
non_numeric = ['Branch', 'Gender', 'Component']

# Drop the non-numerical columns from df
df_numeric = df.drop(non_numeric, axis=1)

# Create a t-SNE model with learning rate 50
m = TSNE(learning_rate=50)

# Fit and transform the t-SNE model on the numeric dataset
tsne_features = m.fit_transform(df_numeric)
print(tsne_features.shape)
```

```
<script.py> output:
(6068, 2)
```

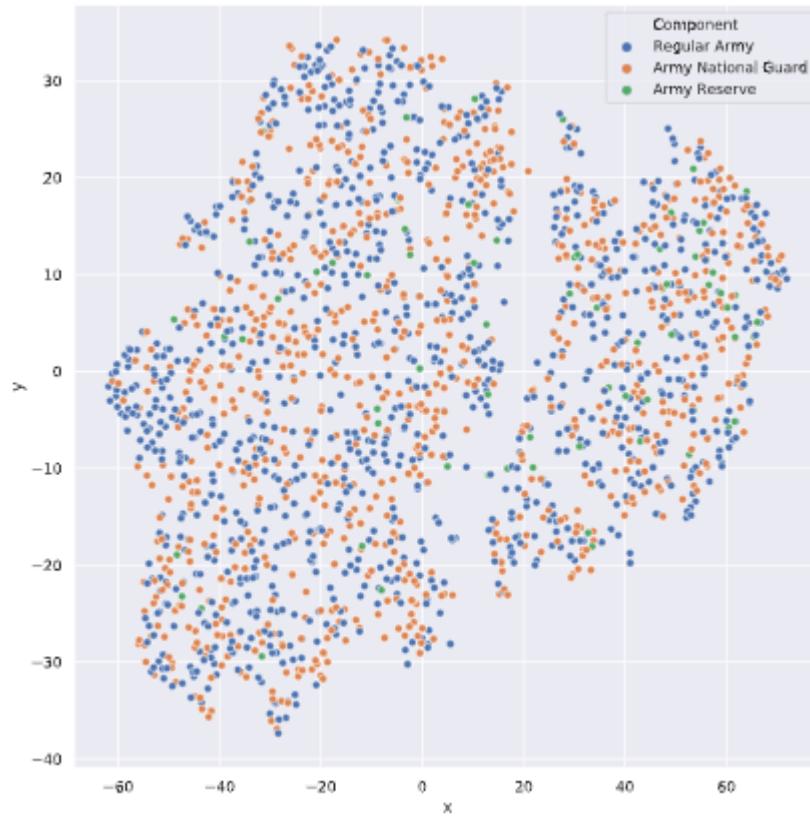
t-SNE reduced the more than 90 features in the dataset to just 2 which you can now plot.

## t-SNE visualisation of dimensionality

Time to look at the results of your hard work. In this exercise, you will visualize the output of t-SNE dimensionality reduction on the combined male and female Ansur dataset. You'll create 3 scatterplots of the 2 t-SNE features ('x' and 'y') which were added to the dataset `df`. In each scatterplot you'll color the points according to a different categorical variable.

`seaborn` has already been imported as `sns` and `matplotlib.pyplot` as `plt`.

```
# Color the points according to Army Component
sns.scatterplot(x="x", y="y", hue='Component', data=df)
plt.show()
```

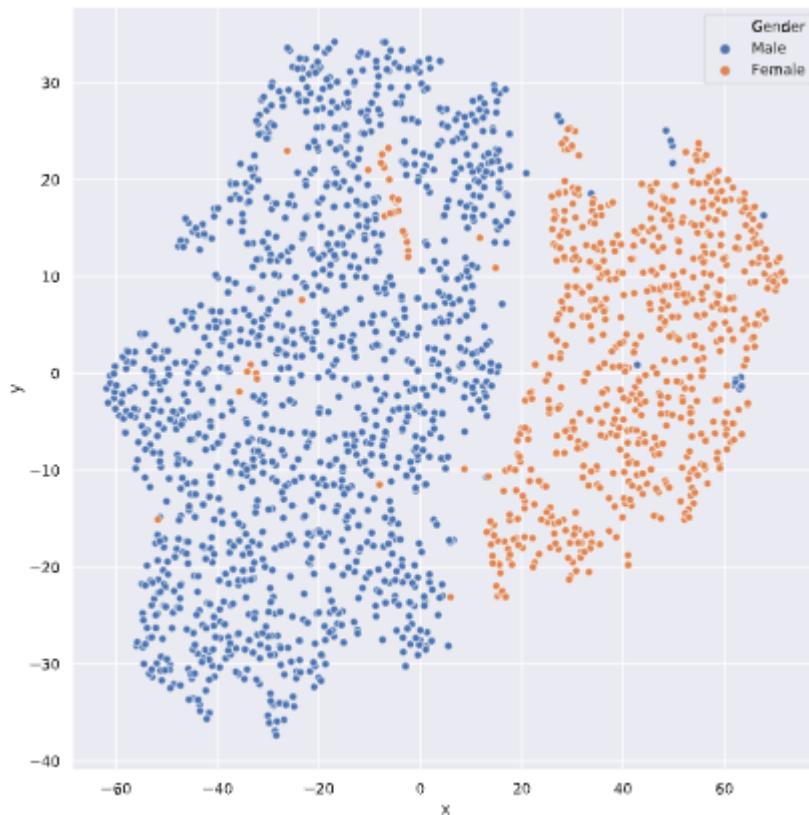


```
# Color the points by Army Branch  
sns.scatterplot(x="x", y="y", hue='Branch', data=df)  
plt.show()
```





```
# Color the points by Gender  
sns.scatterplot(x="x", y="y", hue='Gender', data=df)  
plt.show()
```



There is a Male and a Female cluster. t-SNE found these gender differences in body shape without being told about them explicitly! From the second plot you learned there are more males in the Combat Arms Branch.

• • •

## Chapter 2. Feature selection I, selecting for feature information

In this first out of two chapters on feature selection, you'll learn about the curse of dimensionality and how dimensionality reduction can help you overcome it. You'll be introduced to a number of techniques to detect and remove features that bring little

added value to the dataset. Either because they have little variance, too many missing values, or because they are strongly correlated to other features.

## The curse of dimensionality

Models tend to overfit in high dimensional data.

With more features added, observations should be increased (exponentially) to avoid overfitting and the curse of dimensionality.

## Train — test split

In this chapter, you will keep working with the ANSUR dataset (`ansur_df`). Before you can build a model on your dataset, you should first decide on which feature you want to predict. In this case, you're trying to predict gender.

You need to extract the column holding this feature from the dataset and then split the data into a training and test set. The training set will be used to train the model and the test set will be used to check its performance on unseen data.

```
# Import train_test_split()
from sklearn.model_selection import train_test_split

# Select the Gender column as the feature to be predicted (y)
y = ansur_df['Gender']

# Remove the Gender column to create the training data
X = ansur_df.drop('Gender', axis=1)

# Perform a 70% train and 30% test data split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3)

print("{} rows in test set vs. {} in training set. {}"
Features.".format(X_test.shape[0], X_train.shape[0],
X_test.shape[1]))
```

```
<script.py> output:
    300 rows in test set vs. 700 in training set. 91 Features.
```

You now have a split dataset to train and test the model on.

## Fitting and testing the model

In the previous exercise, you split the dataset into `x_train`, `x_test`, `y_train`, and `y_test`. You'll now create a support vector machine classifier model (`SVC()`) and fit that to the training data. You'll then calculate the accuracy on both the test and training set to detect overfitting.

```
# Import SVC from sklearn.svm and accuracy_score from sklearn.metrics
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score

# Create an instance of the Support Vector Classification class
svc = SVC()

# Fit the model to the training data
svc.fit(X_train, y_train)

# Calculate accuracy scores on both train and test data
accuracy_train = accuracy_score(y_train, svc.predict(X_train))
accuracy_test = accuracy_score(y_test, svc.predict(X_test))

print("{0:.1%} accuracy on test set vs. {1:.1%} on training
set".format(accuracy_test, accuracy_train))
```

```
<script.py> output:
    49.7% accuracy on test set vs. 100.0% on training set
```

Observation: Looks like the model badly overfits on the training data. On unseen data it performs worse than a random selector would.

## Accuracy after dimensionality reduction

Overfitting can be reduced with the help of dimensionality reduction. In this case, you'll apply a rather drastic form of dimensionality reduction by only selecting a single column that has some good information to distinguish between genders. You'll repeat the train-test split, model fit and prediction steps to compare the accuracy on test vs. training data.

```
# Assign just the 'neckcircumferencebase' column from ansur_df to X
X = ansur_df[['neckcircumferencebase']]
```

```
# Split the data, instantiate a classifier and fit the data
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3)
svc = SVC()
svc.fit(X_train, y_train)

# Calculate accuracy scores on both train and test data
accuracy_train = accuracy_score(y_train, svc.predict(X_train))
accuracy_test = accuracy_score(y_test, svc.predict(X_test))

print("{}% accuracy on test set vs. {}% on training set".format(accuracy_test, accuracy_train))
```

```
<script.py> output:
    93.3% accuracy on test set vs. 94.9% on training set
```

Observation: On the full dataset the model is rubbish, but with a single feature we can make good predictions. This is an example of the curse of dimensionality! The model badly overfits when we feed it too many features. It overlooks that neck circumference by itself is pretty different for males and females.

## Features with missing values or little variance

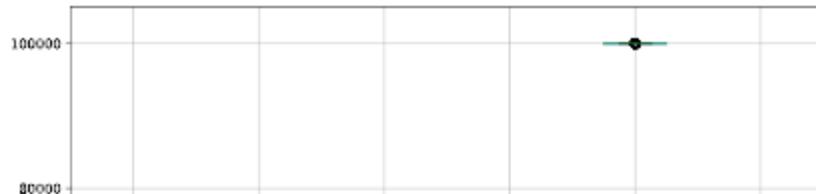
To automate the selection of features with sufficient variance

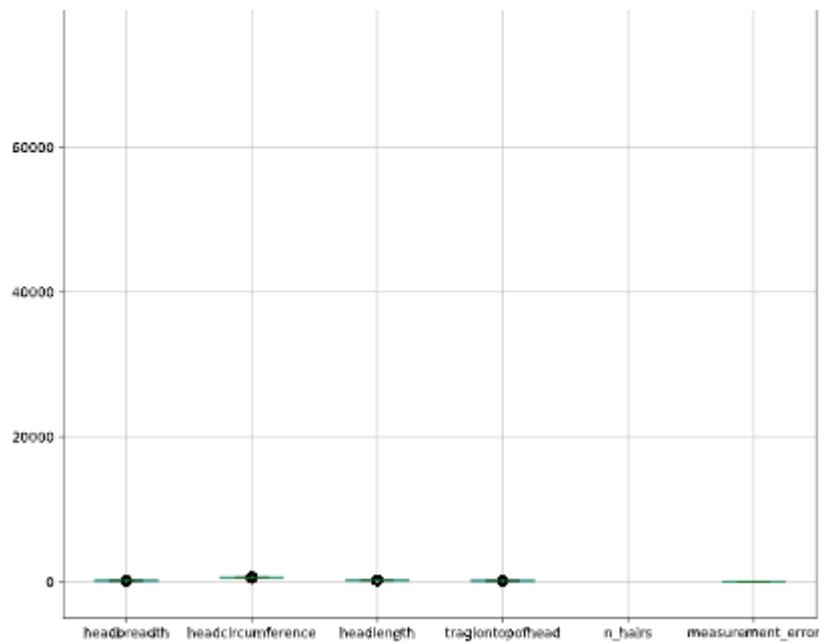
```
(VarianceThreshold(threshold=1)) and not too many null values (df.isna().sum() / len(df) < 0.3).
```

### Finding a good variance threshold

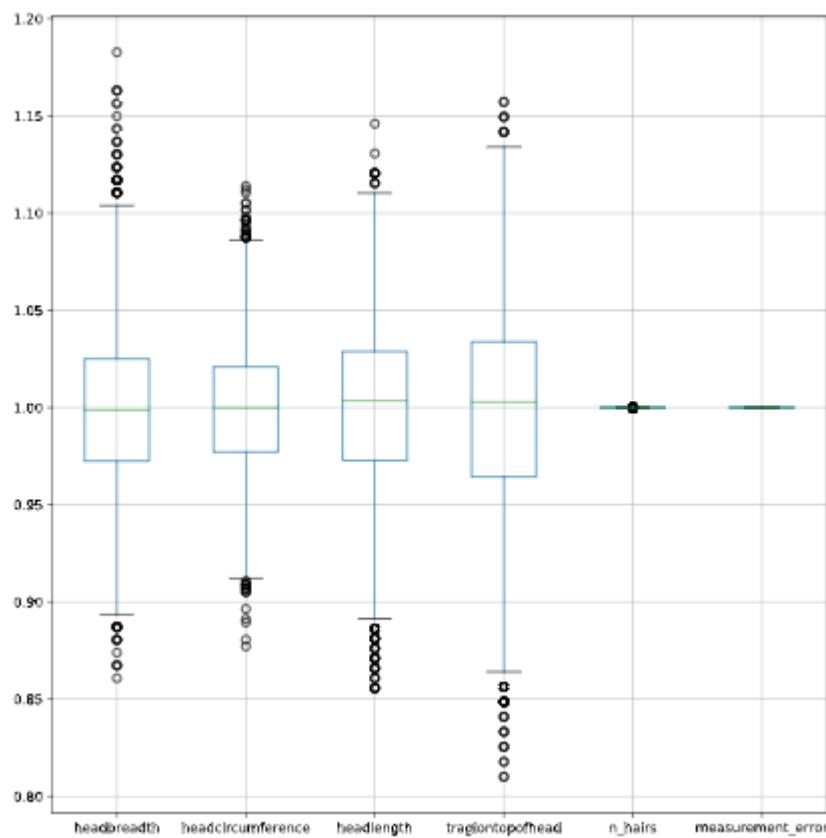
You'll be working on a slightly modified subsample of the ANSUR dataset with just head measurements in `head_df`.

```
# Create the boxplot
head_df.boxplot()
plt.show()
```





```
# Normalize the data
normalized_df = head_df / head_df.mean()
normalized_df.boxplot()
plt.show()
```



```
# Print the variances of the normalized data
print(normalized_df.var())
```

```
<script.py> output:
    headbreadth      1.678952e-03
    headcircumference 1.029623e-03
    headlength       1.867872e-03
    tragiontopofhead 2.639840e-03
    n_hairs          1.002552e-08
    measurement_error 3.231707e-27
```

Inspect the printed variances. If you want to remove the 2 very low variance features.  
What would be a good variance threshold?

Answer: 1.0e-03

A threshold of 1.0e-03 (0.001) will remove the two low variance features.

## Features with low variance

In the previous exercise you established that 0.001 is a good threshold to filter out low variance features in `head_df` after normalization. Now use the `VarianceThreshold` feature selector to remove these features.

```
from sklearn.feature_selection import VarianceThreshold

# Create a VarianceThreshold feature selector
sel = VarianceThreshold(threshold=0.001)

# Fit the selector to normalized head_df
sel.fit(head_df / head_df.mean())

# Create a boolean mask
mask = sel.get_support()

# Apply the mask to create a reduced dataframe
reduced_df = head_df.loc[:, mask]

print("Dimensionality reduced from {} to
{}.".format(head_df.shape[1], reduced_df.shape[1]))
```

```
<script.py> output:
    Dimensionality reduced from 6 to 4.
```

Observation: Successfully removed the 2 low-variance features.

## Removing features with many missing values

You'll apply feature selection on the Boston Public Schools dataset in `school_df`.

Calculate the missing value ratio per feature and then create a mask to remove features with many missing values.

In what range lies highest ratio of missing values for a single feature in the dataset?

Answer: Between 0.9 and 1.0.

```
In [1]: school_df.isna().sum() / len(school_df)
Out[1]:
x           0.000000
y           0.000000
objectid_1  0.000000
objectid    0.000000
bldg_id     0.000000
bldg_name   0.000000
address     0.000000
city        0.000000
zipcode     0.000000
csp_sch_id  0.000000
sch_id      0.000000
sch_name    0.000000
sch_label   0.000000
sch_type    0.000000
shared      0.877863
complex     0.984733
label       0.000000
tlt         0.000000
pl          0.000000
point_x     0.000000
point_y     0.000000
```

## Removing features with many missing values

You'll apply feature selection on the Boston Public Schools dataset in `school_df`.

Calculate the missing value ratio per feature and then create a mask to remove features with many missing values.

```
# Create a boolean mask on whether each feature less than 50% missing
values.
mask = school_df.isna().sum() / len(school_df) < 0.5
```

```
# Create a reduced dataset by applying the mask
reduced_df = school_df.loc[:,mask]

print(school_df.shape)
print(reduced_df.shape)
```

```
<script.py> output:
(131, 21)
(131, 19)
```

Observation: The number of features went down from 21 to 19.

## Pairwise correlation

Pairplot to visually identify strongly correlated features.

Correlation coefficient **r** measures the strength of correlation: -1 (negative correlation), 0 (no correlation), 1 (positive correlation)

Use `df.corr()` for correlation coefficient values.

Use `sns.heatmap(df.corr())` for heatmap.

## Correlation intuition

What statement on correlations is correct?

Answer: The correlation coefficient of A to B is equal to that of B to A.

This is why you can drop half of the correlation matrix without losing information.

## Inspecting the correlation matrix

A sample of the ANSUR body measurements dataset is in `ansur_df`. Use the terminal to create a correlation matrix for this dataset.

What is the correlation coefficient between wrist and ankle circumference?

Answer: 0.702178 (ie, quite a strong positive correlation)

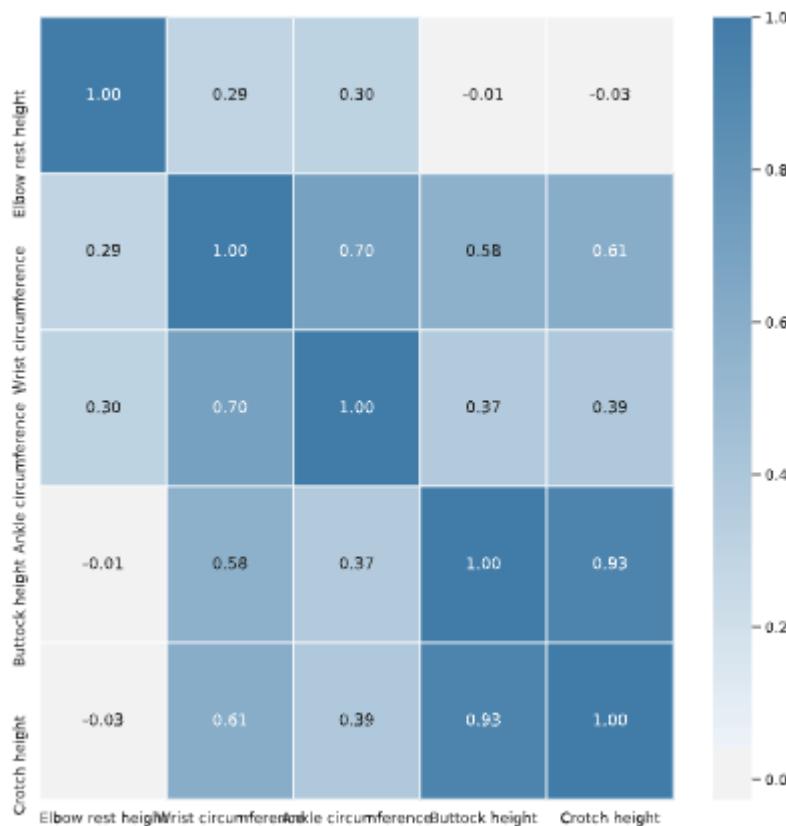
```
In [1]: ansur_df.corr()
Out[1]:
      Elbow rest height  Wrist circumference  Ankle circumference  Buttock height  Crotch height
Elbow rest height        1.000000          0.294753         0.301963       -0.007013       -0.026090
Wrist circumference     0.294753        1.000000          0.702178        0.576679       0.606582
Ankle circumference     0.301963          0.702178        1.000000        0.367548       0.386502
Buttock height        -0.007013          0.576679        0.367548       1.000000       0.929411
Crotch height         -0.026090          0.606582        0.386502       0.929411       1.000000
```

## Visualizing the correlation matrix

Reading the correlation matrix of `ansur_df` in its raw, numeric format doesn't allow us to get a quick overview. Let's improve this by removing redundant values and visualizing the matrix using seaborn.

```
# Create the correlation matrix
corr = ansur_df.corr()

# Draw the heatmap
sns.heatmap(corr, cmap=cmap, center=0, linewidths=1, annot=True,
fmt=".2f")
plt.show()
```

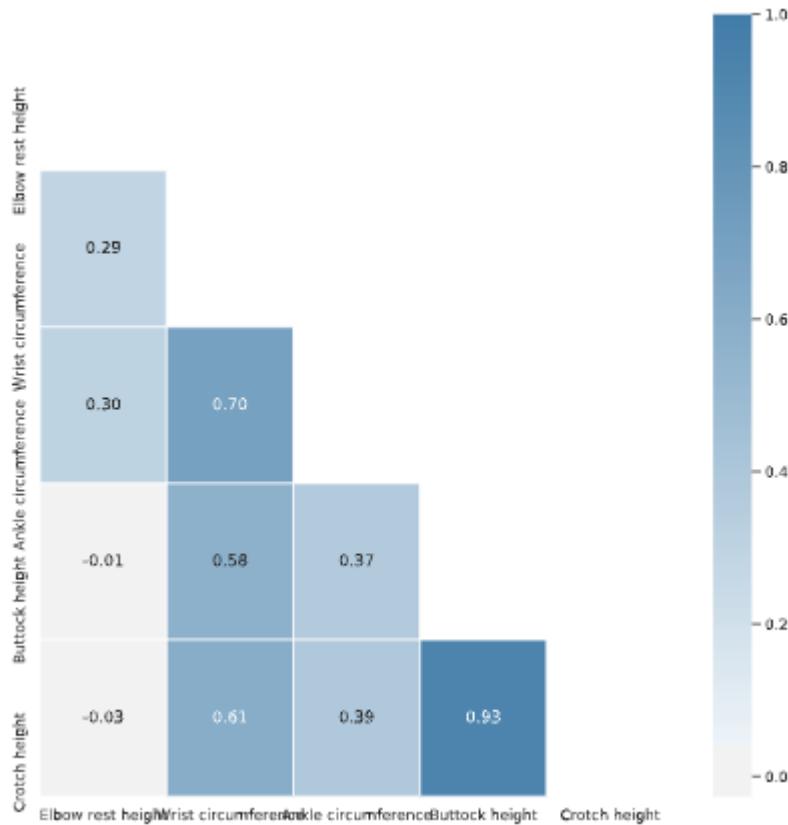


```
# Create the correlation matrix
corr = ansur_df.corr()

# Generate a mask for the upper triangle
mask = np.triu(np.ones_like(corr, dtype=bool))

# Add the mask to the heatmap
sns.heatmap(corr, mask=mask, cmap=cmap, center=0, linewidths=1,
```

```
annot=True, fmt=".2f")
plt.show()
```



Which two features have the strongest correlation?

Answer: Buttock height and Crotch height

The buttock and crotch height have a 0.93 correlation coefficient.

## Removing highly correlated features

Features that are highly correlated (correlation coefficient of close to 1 or -1) do not add more information to the dataset, but add to the complexity.

Note: highly correlated features do not imply causation. Example, number of hospital beds provided for COVID-19 patients are positively correlated to toilet paper sales. However, there is no causation between them.

## Filtering out highly correlated features

You're going to automate the removal of highly correlated features in the numeric ANSUR dataset. You'll calculate the correlation matrix and filter out columns that have a

correlation coefficient of more than 0.95 or less than -0.95.

Since each correlation coefficient occurs twice in the matrix (correlation of A to B equals correlation of B to A) you'll want to ignore half of the correlation matrix so that only one of the two correlated features is removed. Use a mask trick for this purpose.

```
# Calculate the correlation matrix and take the absolute value
corr_matrix = ansur_df.corr().abs()

# Create a True/False mask and apply it
mask = np.triu(np.ones_like(corr_matrix, dtype=bool))
tri_df = corr_matrix.mask(mask)

# List column names of highly correlated features (r > 0.95)
to_drop = [c for c in tri_df.columns if any(tri_df[c] > 0.95)]

# Drop the features in the to_drop list
reduced_df = ansur_df.drop(to_drop, axis=1)

print("The reduced dataframe has {} columns.".format(reduced_df.shape[1]))
```

```
The original dataframe has 99 columns.
<script.py> output:
    The reduced dataframe has 88 columns.
```

Observation: Automated the removal of highly correlated features.

## Nuclear energy and pool drownings

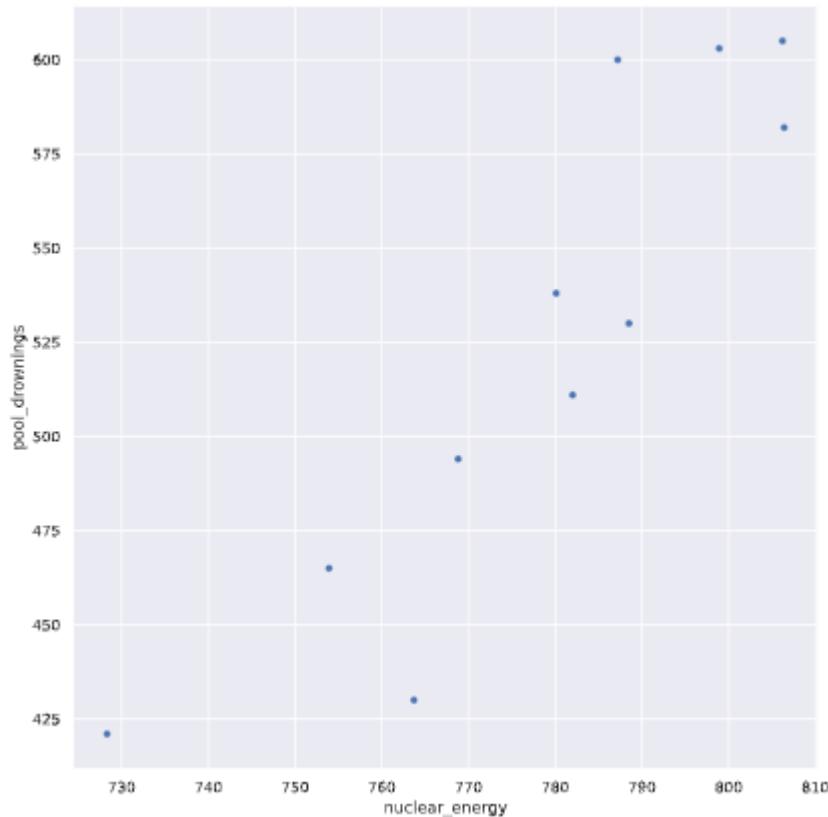
The dataset in `weird_df` contains actual data provided by the US Centers for Disease Control & Prevention and Department of Energy. Let's see if we can find a pattern.

```
# Print the first five lines of weird_df
print(weird_df.head())

# Put nuclear energy production on the x-axis and the number of pool
# drownings on the y-axis
sns.scatterplot(x='nuclear_energy', y='pool_drownings',
                 data=weird_df)
plt.show()
```

```
# Print out the correlation matrix of weird_df
print(weird_df.corr())
```

```
<script.py> output:
    pool_drownings  nuclear_energy
pool_drownings      1.000000      0.901179
nuclear_energy       0.901179      1.000000
```



What can you conclude from the strong correlation ( $r=0.9$ ) between these features?

Answer: Not much, correlation does not imply causation. While the example is silly, you'll be amazed how often people misunderstand correlation vs causation.

• • •

## Chapter 3. Feature selection II, selecting for model accuracy

In this second chapter on feature selection, you'll learn how to let models help you find the most important features in a dataset for predicting a particular target feature. In the

final lesson of this chapter, you'll combine the advice of multiple, different, models to decide on which features are worth keeping.

## Selecting features for model performance

Select (keep) features that largely affect model performance

Use RFE (Recursive Feature Elimination) to drop useless features

## Building a diabetes classifier

You'll be using the Pima Indians diabetes dataset to predict whether a person has diabetes using logistic regression. There are 8 features and one target in this dataset. The data has been split into a training and test set as `x_train`, `y_train`, `x_test`, and `y_test`.

A `StandardScaler()` instance has been predefined as `scaler` and a `LogisticRegression()` one as `lr`.

```
# Fit scaler on training features and transform these in one go
X_train_std = scaler.fit_transform(X_train)

# Fit the logistic regression model on the scaled training data
lr.fit(X_train_std, y_train)

# Scale the test features
X_test_std = scaler.transform(X_test)

# Predict diabetes presence on the scaled test set
y_pred = lr.predict(X_test_std)

# Prints accuracy metrics and feature coefficients
print("{}% accuracy on test set.".format(accuracy_score(y_test,
y_pred)))
print(dict(zip(X.columns, abs(lr.coef_[0]).round(2))))
```

```
<script.py> output:
    79.6% accuracy on test set.
    {'diastolic': 0.03, 'pregnant': 0.04, 'family': 0.34, 'age':
    0.34, 'glucose': 1.23, 'bmi': 0.38, 'insulin': 0.19, 'triceps': 0.24}
```

We get almost 80% accuracy on the test set. Take a look at the differences in model coefficients for the different features.

## Manual Recursive Feature Elimination

Now that we've created a diabetes classifier, let's see if we can reduce the number of features without hurting the model accuracy too much. On the second line of code, the features are selected from the original dataframe. Adjust this selection.

```
# Remove the feature with the lowest model coefficient
X = diabetes_df[['pregnant', 'glucose', 'diastolic', 'triceps',
'insulin', 'bmi', 'family', 'age']]

# Performs a 25-75% train test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=0)

# Scales features and fits the logistic regression model
lr.fit(scaler.fit_transform(X_train), y_train)

# Calculates the accuracy on the test set and prints coefficients
acc = accuracy_score(y_test, lr.predict(scaler.transform(X_test)))
print("{}% accuracy on test set.".format(acc))
print(dict(zip(X.columns, abs(lr.coef_[0]).round(2))))
```

<script.py> output:  
 79.6% accuracy on test set.  
 {'diastolic': 0.03, 'pregnant': 0.04, 'family': 0.34, 'age':  
 0.34, 'glucose': 1.23, 'bmi': 0.38, 'insulin': 0.19, 'triceps': 0.24}

Step 1: 79.6% accuracy, lowest=0.03, remove 'diastolic'

Step 2: 80.6% accuracy, lowest2=0.05&0.20, remove 'pregnant' & 'insulin'

Step 3: 79.6% accuracy, highest=1.13, keep 'glucose'

Finally we get 76.5% accuracy

Observation: Removing all but one feature only reduced the accuracy by a few percent.

## Automatic Recursive Feature Elimination

Now let's automate this recursive process. Wrap a Recursive Feature Eliminator (RFE) around our logistic regression estimator and pass it the desired number of features.

Important note: features have been scaled

```
# Create the RFE with a LogisticRegression estimator and 3 features
# to select
rfe = RFE(estimator=LogisticRegression(), n_features_to_select=3,
verbose=1)

# Fits the eliminator to the data
rfe.fit(X_train, y_train)

# Print the features and their ranking (high = dropped early on)
print(dict(zip(X.columns, rfe.ranking_)))

# Print the features that are not eliminated
print(X.columns[rfe.support_])

# Calculates the test set accuracy
acc = accuracy_score(y_test, rfe.predict(X_test))
print("{}% accuracy on test set.".format(acc))
```

```
<script.py> output:
    Fitting estimator with 8 features.
    Fitting estimator with 7 features.
    Fitting estimator with 6 features.
    Fitting estimator with 5 features.
    Fitting estimator with 4 features.
    {'diastolic': 6, 'pregnant': 5, 'family': 2, 'age': 1, 'glucose':
1, 'bmi': 1, 'insulin': 4, 'triceps': 3}
    Index(['glucose', 'bmi', 'age'], dtype='object')
    80.6% accuracy on test set.
```

Observation: When we eliminate all but the 3 most relevant features we get a 80.6% accuracy on the test set.

## Tree-based feature selection

Random forest classifier can inherently perform feature selection to avoid overfitting.

### Building a random forest model

You'll again work on the Pima Indians dataset to predict whether an individual has diabetes. This time using a random forest classifier. You'll fit the model on the training data after performing the train-test split and consult the feature importance values.

The feature and target datasets are `x` and `y`. Same goes for the necessary packages and functions.

```
# Perform a 75% training and 25% test data split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.25, random_state=0)

# Fit the random forest model to the training data
rf = RandomForestClassifier(random_state=0)
rf.fit(X_train, y_train)

# Calculate the accuracy
acc = accuracy_score(y_test, rf.predict(X_test))

# Print the importances per feature
print(dict(zip(X.columns, rf.feature_importances_.round(2)))))

# Print accuracy
print("{0:.1%} accuracy on test set.".format(acc))
```

```
<script.py> output:
{'triceps': 0.11, 'pregnant': 0.09, 'glucose': 0.21, 'insulin': 0.13, 'bmi': 0.09, 'family': 0.12, 'diastolic': 0.08, 'age': 0.16}
77.6% accuracy on test set.
```

Observation: The random forest model gets 78% accuracy on the test set and '`glucose`' is the most important feature (0.21).

## Random forest for feature selection

Now lets use the fitted random model to select the most important features from our input dataset `x`.

```
# Create a mask for features importances above the threshold
mask = rf.feature_importances_ > 0.15
# [False  True False False False False  True]

# Apply the mask to the feature dataset X
reduced_X = X.loc[:, mask]

# prints out the selected column names
print(reduced_X.columns)
```

```
<script.py> output:
    Index(['glucose', 'age'], dtype='object')
```

Observation: Only the features '`glucose`' and '`age`' were considered sufficiently important.

## Recursive Feature Elimination with random forests

You'll wrap a Recursive Feature Eliminator around a random forest model to remove features step by step. This method is more conservative compared to selecting features after applying a single importance threshold. Since dropping one feature can influence the relative importance of the others.

```
# Wrap the feature eliminator around the random forest model
rfe = RFE(estimator=RandomForestClassifier(), n_features_to_select=2,
           step=2, verbose=1)

# Fit the model to the training data
rfe.fit(X_train, y_train)

# Create a mask using an attribute of rfe
mask = rfe.support_

# Apply the mask to the feature dataset X and print the result
reduced_X = X.loc[:, mask]
print(reduced_X.columns)
```

```
<script.py> output:
    Fitting estimator with 8 features.
    Fitting estimator with 6 features.
    Fitting estimator with 4 features.
    Index(['glucose', 'insulin'], dtype='object')
```

Observation: Compared to the quick and dirty single threshold method from the previous exercise, one of the selected features is different.

## Regularized linear regression

Dimensionality reduction in linear regression using regularisation, eg. LASSO

## Creating a LASSO regressor

You'll be working on the numeric ANSUR body measurements dataset to predict a persons Body Mass Index (BMI) using the pre-imported `Lasso()` regressor. BMI is a metric derived from body height and weight but those two features have been removed from the dataset to give the model a challenge.

Note: To standardize the data using the `StandardScaler()` to make sure all coefficients face a comparable regularizing force trying to bring them down.

```
# Set the test size to 30% to get a 70-30% train test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=0)

# Fit the scaler on the training features and transform these in one
# go
X_train_std = scaler.fit_transform(X_train)

# Create the Lasso model
la = Lasso()

# Fit it to the standardized training data
la.fit(X_train_std, y_train)
```

After fitting the Lasso model to the standardized training data, let's look at the results.

## Lasso model results

Now that you've trained the Lasso model, you'll score its predictive capacity ( $R^2$ ) on the test set and count how many features are ignored because their coefficient is reduced to zero.

The `Lasso()` model and `StandardScaler()` have been instantiated as `la` and `scaler` respectively and both were fitted to the training data.

```
# Transform the test set with the pre-fitted scaler
X_test_std = scaler.transform(X_test)

# Calculate the coefficient of determination (R squared) on
# X_test_std
r_squared = la.score(X_test_std, y_test)
```

```

print("The model can predict {0:.1%} of the variance in the test
set.".format(r_squared))

# Create a list that has True values when coefficients equal 0
zero_coef = la.coef_ == 0

# Calculate how many features have a zero coefficient
n_ignored = sum(zero_coef)
print("The model has ignored {} out of {}
features.".format(n_ignored, len(la.coef_)))

```

```

<script.py> output:
    The model can predict 84.7% of the variance in the test set.
    The model has ignored 82 out of 91 features.

```

Observation: We can predict almost 85% of the variance in the BMI value using just 9 out of 91 of the features. The R<sup>2</sup> could be higher though.

## Adjusting the regularization strength

Your current Lasso model has an R2R2 score of 84.7%. When a model applies overly powerful regularization it can suffer from high bias, hurting its predictive power.

Let's improve the balance between predictive power and model simplicity by tweaking the `alpha` parameter.

```

# Find the highest alpha value with R-squared above 98%
la = Lasso(alpha=0.1, random_state=0)

# Fits the model and calculates performance stats
la.fit(X_train_std, y_train)
r_squared = la.score(X_test_std, y_test)
n_ignored_features = sum(la.coef_ == 0)

# Print peformance stats
print("The model can predict {0:.1%} of the variance in the test
set.".format(r_squared))
print("{} out of {} features were
ignored.".format(n_ignored_features, len(la.coef_)))

```

```

<script.py> output:
    The model can predict 98.3% of the variance in the test set.
    64 out of 91 features were ignored.

```

Observation: With this more appropriate regularization strength we can predict 98% of the variance in the BMI value while ignoring 2/3 of the features. Below is an automatic way to find optimal alpha.

## Combining feature selectors

To combine feature selection capability of multiple models

### Creating a LassoCV regressor

You'll be predicting biceps circumference on a subsample of the male ANSUR dataset using the `LassoCV()` regressor that automatically tunes the regularization strength (alpha value) using Cross-Validation.

Note: The training and test data has been **standardized**, as `x_train`, `x_test`, `y_train`, and `y_test`.

```
from sklearn.linear_model import LassoCV

# Create and fit the LassoCV model on the training set
lcv = LassoCV()
lcv.fit(X_train, y_train)
print('Optimal alpha = {:.3f}'.format(lcv.alpha_))

# Calculate R squared on the test set
r_squared = lcv.score(X_test, y_test)
print('The model explains {:.1%} of the test set variance'.format(r_squared))

# Create a mask for coefficients not equal to zero
lcv_mask = lcv.coef_ != 0
print('{0} features out of {1} selected'.format(sum(lcv_mask), len(lcv_mask)))
```

```
<script.py> output:
    Optimal alpha = 0.089
    The model explains 88.2% of the test set variance
    26 features out of 32 selected
```

We got a decent R squared and removed 6 features. We'll save the `lcv_mask` for later on.

## Ensemble models for extra votes

The `LassoCV()` model selected 26 out of 32 features. Not bad, but not a spectacular dimensionality reduction either. Let's use two more models to select the 10 features they consider most important using the Recursive Feature Eliminator (RFE).

Note: The training and test data has been **standardized**, as `x_train`, `x_test`, `y_train`, and `y_test`.

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

# Select 10 features with RFE on a GradientBoostingRegressor, drop 3
# features on each step
rfe_gb = RFE(estimator=GradientBoostingRegressor(),
              n_features_to_select=10, step=3, verbose=1)
rfe_gb.fit(X_train, y_train)

# Calculate the R squared on the test set
r_squared = rfe_gb.score(X_test, y_test)
print('The model can explain {:.1%} of the variance in the test
      set'.format(r_squared))

# Assign the support array to gb_mask
gb_mask = rfe_gb.get_support()
```

```
<script.py> output:
    Fitting estimator with 32 features.
    Fitting estimator with 29 features.
    Fitting estimator with 26 features.
    Fitting estimator with 23 features.
    Fitting estimator with 20 features.
    Fitting estimator with 17 features.
    Fitting estimator with 14 features.
    Fitting estimator with 11 features.
    The model can explain 85.6% of the variance in the test set
```

Repeat using `RandomForestRegressor()`

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

```
# Select 10 features with RFE on a RandomForestRegressor, drop 3
# features on each step
rfe_rf = RFE(estimator=RandomForestRegressor(),
              n_features_to_select=10, step=3, verbose=1)
rfe_rf.fit(X_train, y_train)

# Calculate the R squared on the test set
r_squared = rfe_rf.score(X_test, y_test)
print('The model can explain {0:.1%} of the variance in the test
      set'.format(r_squared))

# Assign the support array to gb_mask
rf_mask = rfe_rf.support_
```

```
<script.py> output:
    Fitting estimator with 32 features.
    Fitting estimator with 29 features.
    Fitting estimator with 26 features.
    Fitting estimator with 23 features.
    Fitting estimator with 20 features.
    Fitting estimator with 17 features.
    Fitting estimator with 14 features.
    Fitting estimator with 11 features.
    The model can explain 84.0% of the variance in the test set
```

Including the Lasso linear model from the previous exercise, we now have the votes from 3 models on which features are important.

## Combining 3 feature selectors

We'll combine the votes of the 3 models you built in the previous exercises, to decide which features are important into a meta mask. We'll then use this mask to reduce dimensionality and see how a simple linear regressor performs on the reduced dataset.

The per model votes are `lcv_mask`, `rf_mask`, and `gb_mask`;

The feature and target datasets are `x` and `y`.

```
# Sum the votes of the three models
votes = np.sum([lcv_mask, rf_mask, gb_mask], axis=0)
# [1 0 2 2 0 1 0 3 1 1 1 3 1 1 1 3 0 1 1 2 1 1 2 1 1 3 2 1 3 1 3 3]

# Create a mask for features selected by all 3 models
meta_mask = votes >= 3
# [False False False False False False False True False False False]
```

```

True False False False  True False False False False False
False False False  True False False  True False  True  True]

# Apply the dimensionality reduction on X
X_reduced = X.loc[:, meta_mask]

# Plug the reduced dataset into a linear regression pipeline
X_train, X_test, y_train, y_test = train_test_split(X_reduced, y,
test_size=0.3, random_state=0)
lm.fit(scaler.fit_transform(X_train), y_train)
r_squared = lm.score(scaler.transform(X_test), y_test)
print('The model can explain {0:.1%} of the variance in the test set
using {1:} features.'.format(r_squared, len(lm.coef_)))

```

```

<script.py> output:
Index(['chestcircumference', 'forearmcircumferenceflexed',
'hipbreadth', 'thighcircumference', 'waistcircumference',
'wristheight', 'BMI'], dtype='object')

<script.py> output:
The model can explain 86.8% of the variance in the test set using
7 features.

```

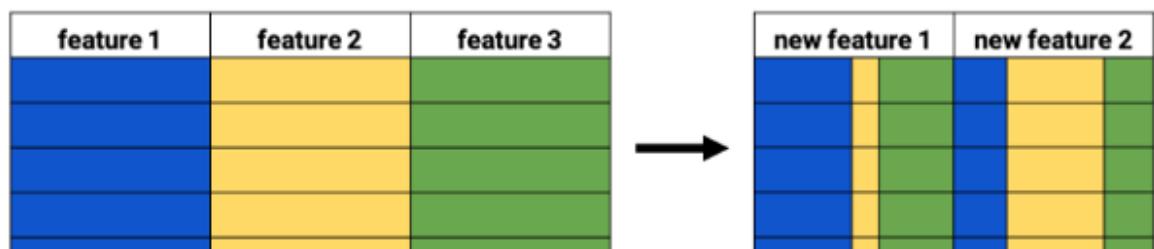
Observation: Using the votes from 3 models you were able to select just 7 features that allowed a simple linear model to get a high accuracy!

• • •

## Chapter 4. Feature extraction

This chapter is a deep-dive on the most frequently used dimensionality reduction algorithm, Principal Component Analysis (PCA). You'll build intuition on how and why this algorithm is so powerful and will apply it both for data exploration and data pre-processing in a modeling pipeline. You'll end with a cool image compression use case.

## Feature extraction





## Manual feature extraction I

You want to compare prices for specific products between stores. The features in `sales_df` are: `storeID`, `product`, `quantity` and `revenue`. The `quantity` and `revenue` features tell you how many items of a particular product were sold in a store and what the total revenue was. For the purpose of your analysis it's more interesting to know the average price per product.

```
# Calculate the price from the quantity sold and revenue
sales_df['price'] = sales_df['revenue'] / sales_df['quantity']

# Drop the quantity and revenue features
reduced_df = sales_df.drop(['revenue', 'quantity'], axis=1)

print(reduced_df.head())
```

```
<script.py> output:
    storeID  product      price
0        A  Apples  5.135616
1        A  Bananas  3.365105
2        A  Oranges  5.317020
3        B  Apples  5.143417
4        B  Bananas  3.898517
```

Observation: When you understand the dataset well, always check if you can calculate relevant features and drop irrelevant ones.

## Manual feature extraction II

You're working on a variant of the ANSUR dataset, `height_df`, where a person's height was measured 3 times. Add a feature with the mean height to the dataset, then drop the 3 original features.

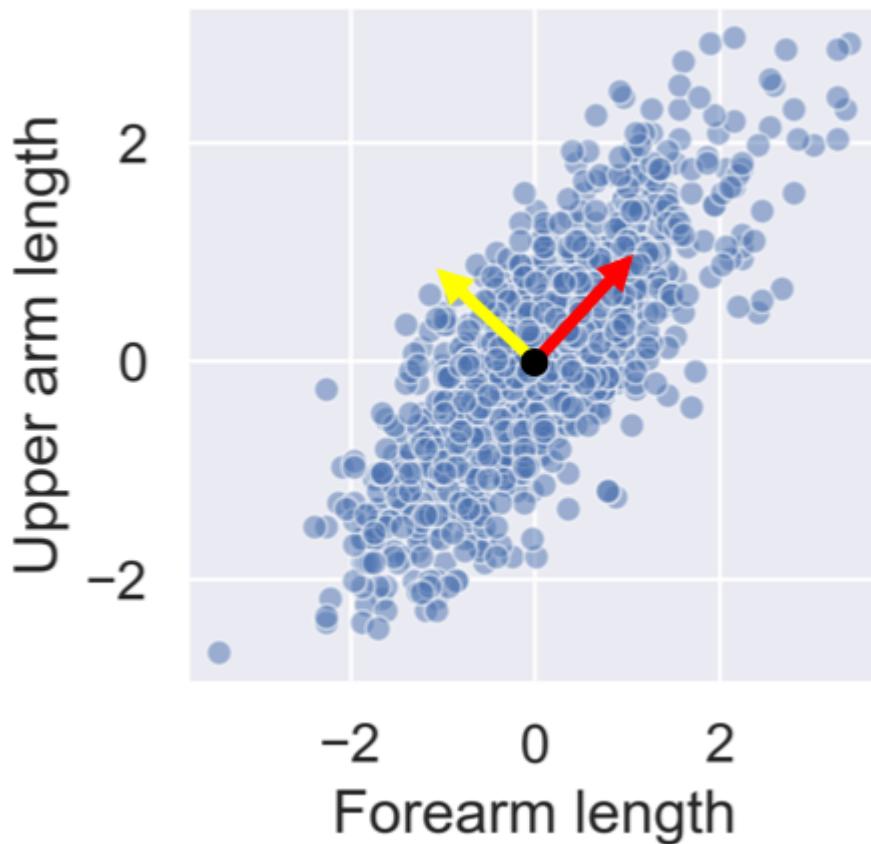
```
# Calculate the mean height
height_df['height'] = height_df[['height_1', 'height_2',
                                'height_3']].mean(axis=1)

# Drop the 3 original height features
reduced_df = height_df.drop(['height_1', 'height_2', 'height_3'],
```

```
axis=1)  
  
print(reduced_df.head())
```

Observation: To calculate a new feature that is still easy to understand compared to, for instance, principal components.

## Principal component intuition



After standardizing the lower and upper arm lengths from the ANSUR dataset we've added two perpendicular vectors that are aligned with the main directions of variance. We can describe each point in the dataset as a combination of these two vectors multiplied with a value each. These values are then called principal components.

Which of the following statements is true?

- People with a negative component for the yellow vector have long forearms relative to their upper arms.
- People with a positive component for the yellow vector have long forearms relative to their upper arms.

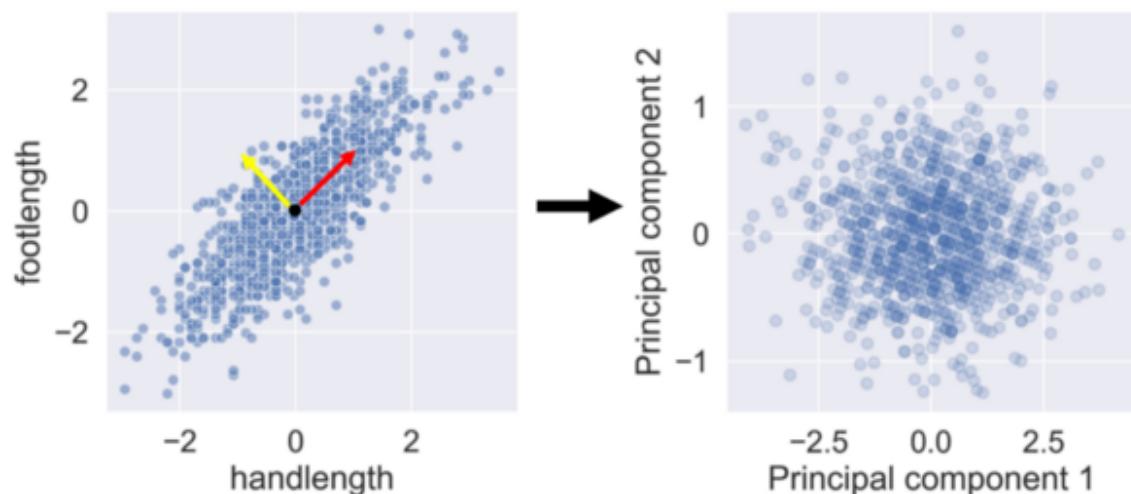
- People with a negative component for the red vector have above average arm lengths.
- People with a positive component for the red vector have below average arm lengths.

Answer: People with a negative component for the yellow vector have long forearms relative to their upper arms.

## Principal component analysis (PCA)

Always scale data (using StandardScaler) before performing PCA

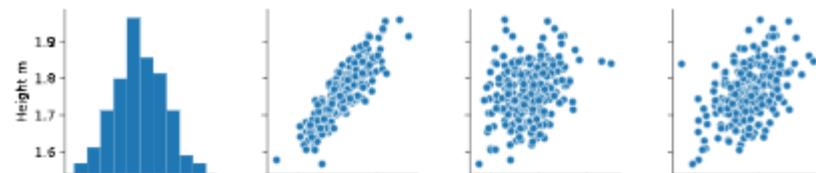
### PCA removes correlation

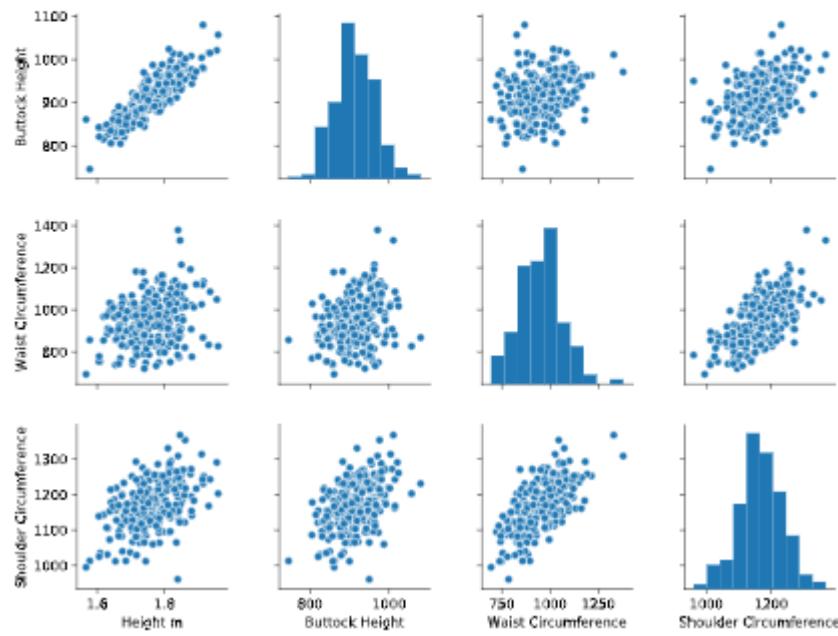


### Calculating Principal Components

You'll visually inspect a 4 feature sample of the ANSUR dataset (`ansur_df`) before and after PCA using Seaborn's `pairplot()`. This will allow you to inspect the pairwise correlations between the features.

```
# Create a pairplot to inspect ansur_df
sns.pairplot(ansur_df)
plt.show()
```





```

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA

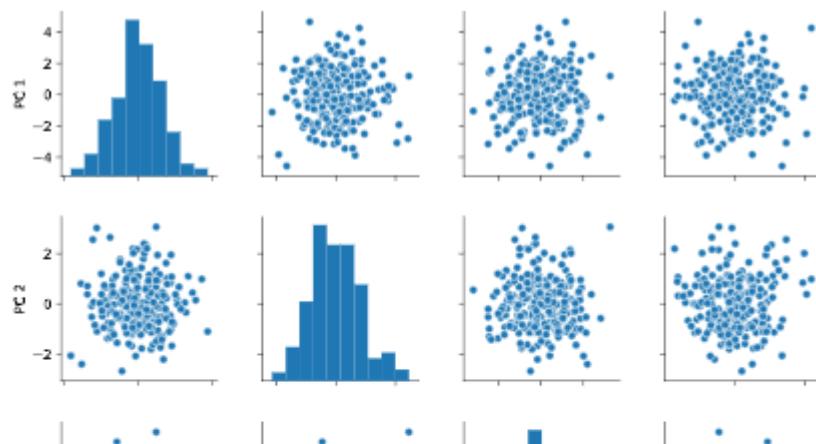
# Create the scaler
scaler = StandardScaler()
ansur_std = scaler.fit_transform(ansur_df)

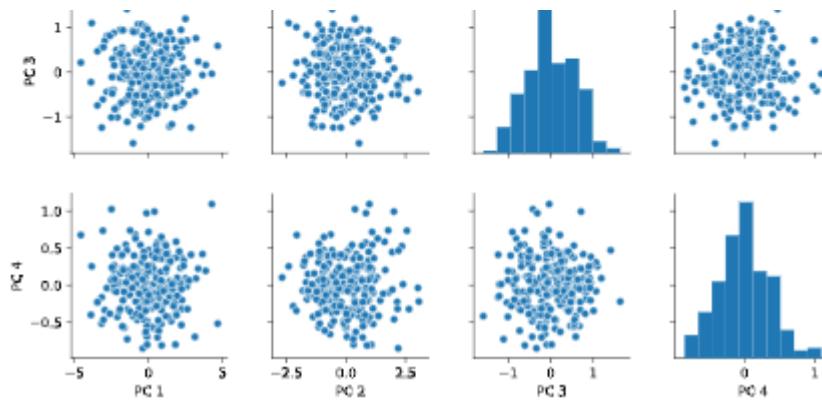
# Create the PCA instance and fit and transform the data with pca
pca = PCA()
pc = pca.fit_transform(ansur_std)

# This changes the numpy array output back to a dataframe
pc_df = pd.DataFrame(pc, columns=['PC 1', 'PC 2', 'PC 3', 'PC 4'])

# Create a pairplot of the principal component dataframe
sns.pairplot(pc_df)
plt.show()

```





Observation: In contrast to the input features, none of the principal components are correlated to one another.

## PCA on a larger dataset

You'll now apply PCA on a somewhat larger ANSUR datasample with 13 dimensions in `ansur_df`. The fitted model will be used in the next exercise. Since we are not using the principal components themselves there is no need to transform the data, instead, it is sufficient to fit `pca` to the data.

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

# Scale the data
scaler = StandardScaler()
ansur_std = scaler.fit_transform(ansur_df)

# Apply PCA
pca = PCA()
pca.fit(ansur_std)
```

You've fitted PCA on our 13 feature datasample. Now let's see how the components explain the variance.

## PCA explained variance

You'll be inspecting the variance explained by the different principal components of the `pca` instance you created in the previous exercise.

```
# Inspect the explained variance ratio per component
print(pca.explained_variance_ratio_)
```

```
<script.py> output:
[0.61449404 0.19893965 0.06803095 0.03770499 0.03031502 0.0171759
0.01072762 0.00656681 0.00634743 0.00436015 0.0026586 0.00202617
0.00065268]
```

How much of the variance is explained by the 4th principal component?

Answer: About 3.77%

```
# Print the cumulative sum of the explained variance ratio
print(pca.explained_variance_ratio_.cumsum())
```

```
<script.py> output:
[0.61449404 0.81343368 0.88146463 0.91916962 0.94948464
0.96666054 0.97738816 0.98395496 0.99030239 0.99466254 0.99732115
0.99934732 1.]
```

What's the lowest number of principal components you should keep if you don't want to lose more than 10% of explained variance during dimensionality reduction?

Answer: 4 principal components

Observation: Using just 4 principal components we can explain more than 90% of the variance in the 13 feature dataset.

## PCA applications

The remaining components can be hard to interpret. To understand the components, look at the components' attribute.

A pipeline can include the scaling operation, applying PCA operation, and a prediction model.

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

## Understanding the components

You'll apply PCA to the numeric features of the Pokemon dataset, `poke_df`, using a pipeline to combine the feature scaling and PCA in one go. You'll then interpret the meanings of the first two components.

```
# Build the pipeline
pipe = Pipeline([('scaler', StandardScaler()),
                 ('reducer', PCA(n_components=2))])

# Fit it to the dataset and extract the component vectors
pipe.fit(poke_df)
vectors = pipe.steps[1][1].components_.round(2)

# Print feature effects
print('PC 1 effects = ' + str(dict(zip(poke_df.columns,
                                         vectors[0]))))
print('PC 2 effects = ' + str(dict(zip(poke_df.columns,
                                         vectors[1]))))

<script.py> output:
    PC 1 effects = {'Attack': 0.44, 'Speed': 0.34, 'Sp. Def': 0.45,
'Defense': 0.36, 'Sp. Atk': 0.46, 'HP': 0.39}
    PC 2 effects = {'Attack': -0.01, 'Speed': -0.67, 'Sp. Def': 0.24,
'Defense': 0.63, 'Sp. Atk': -0.31, 'HP': 0.08}
```

Inspect the PC 1 effects. Which statement is true?

- Sp. Atk has the biggest effect on this feature by far. PC 1 can be interpreted as a measure of how good a Pokemon's special attack is.
- All features have a similar positive effect. PC 1 can be interpreted as a measure of overall quality (high stats).

Answer: All features have a similar positive effect. PC 1 can be interpreted as a measure of overall quality (high stats).

Inspect the PC 2 effects. Which statement is true?

- Defense has a strong positive effect on the second component and speed a strong negative one. This component quantifies an agility vs. armor & protection trade-off.
- Fast Pokemon have high values for this component.

Answer: Defense has a strong positive effect on the second component and speed a strong negative one. This component quantifies an agility vs. armor & protection trade-off.

## PCA for feature exploration

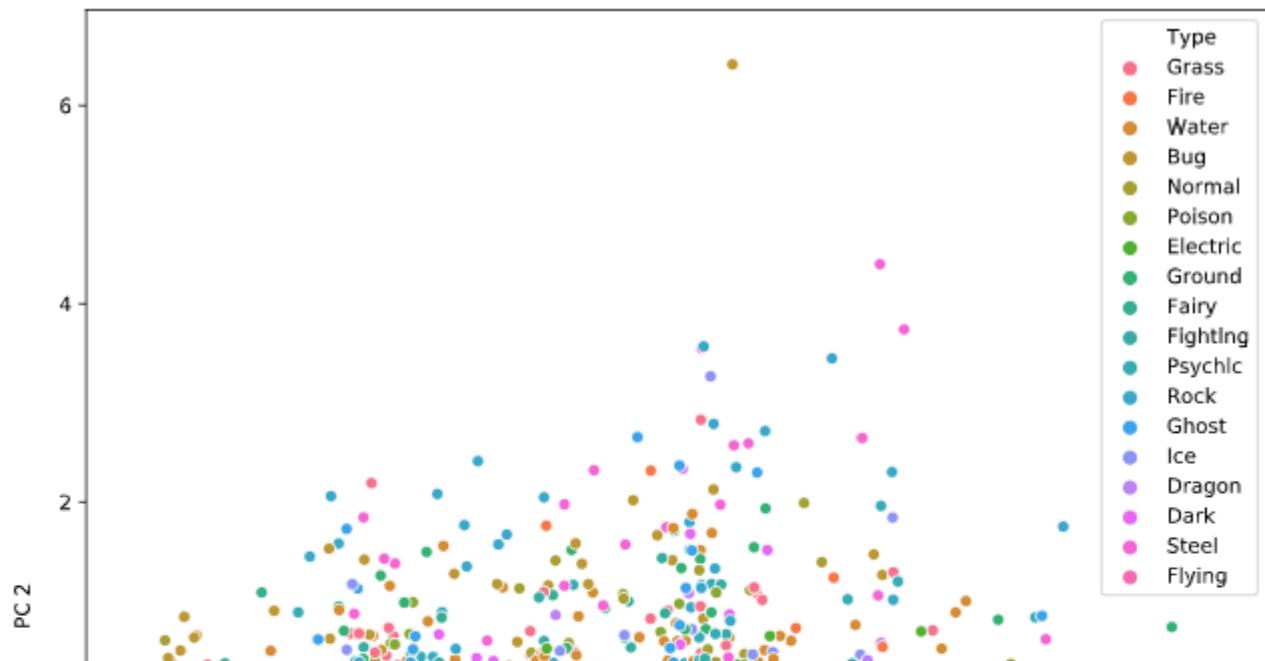
You'll use the PCA pipeline you've built in the previous exercise to visually explore how some categorical features relate to the variance in `poke_df`. These categorical features (`Type` & `Legendary`) can be found in a separate dataframe `poke_cat_df`.

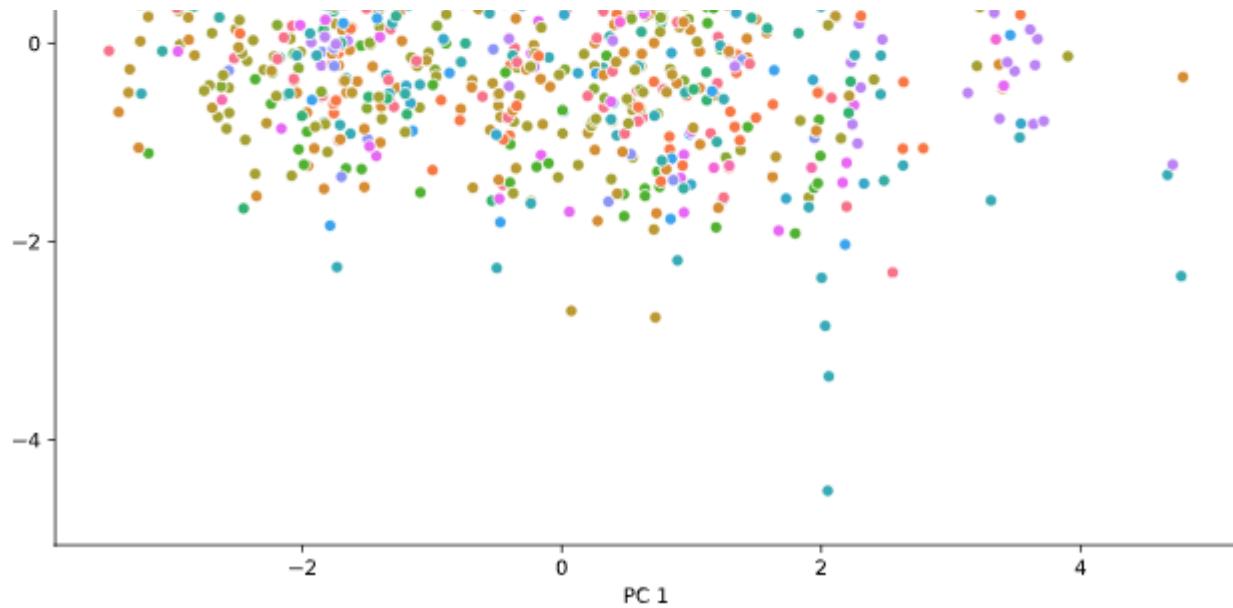
```
# Build the pipeline
pipe = Pipeline([('scaler', StandardScaler()),
                 ('reducer', PCA(n_components=2))])

# Fit the pipeline to poke_df and transform the data
pc = pipe.fit_transform(poke_df)

# Add the 2 components to poke_cat_df
poke_cat_df['PC 1'] = pc[:, 0]
poke_cat_df['PC 2'] = pc[:, 1]

# Use the Type feature to color the PC 1 vs PC 2 scatterplot
sns.scatterplot(data=poke_cat_df,
                  x='PC 1', y='PC 2', hue='Type')
plt.show()
```



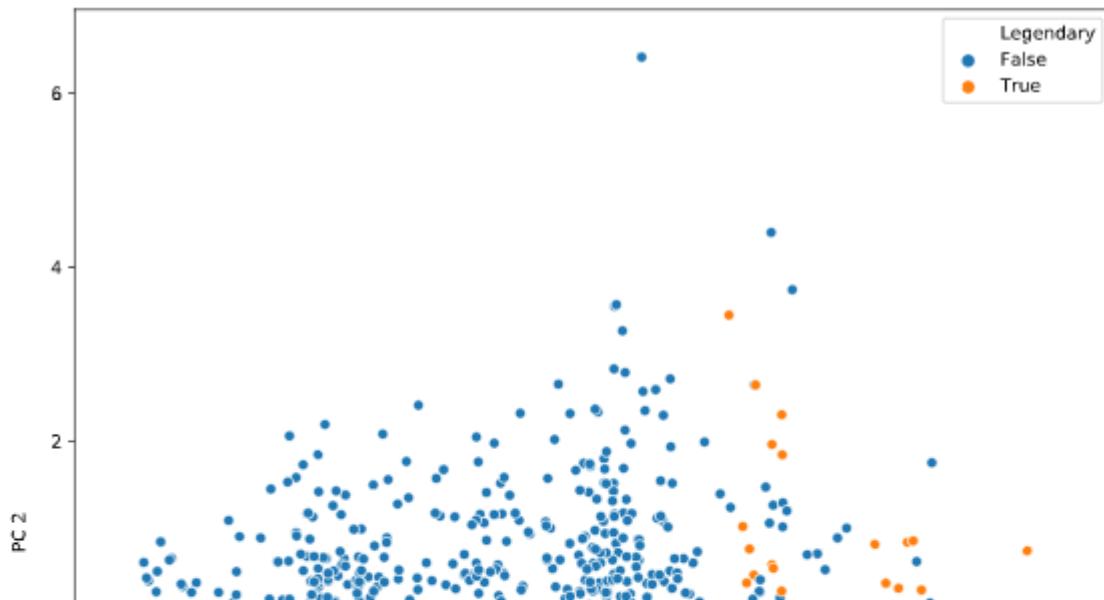


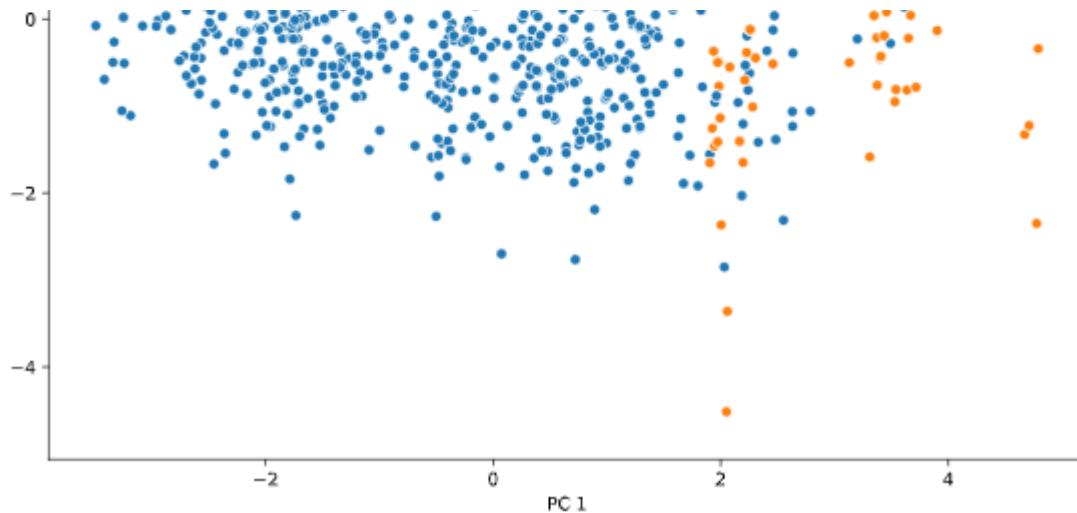
```
pipe = Pipeline([('scaler', StandardScaler()),
                 ('reducer', PCA(n_components=2))])

# Fit the pipeline to poke_df and transform the data
pc = pipe.fit_transform(poke_df)

# Add the 2 components to poke_cat_df
poke_cat_df['PC 1'] = pc[:, 0]
poke_cat_df['PC 2'] = pc[:, 1]

# Use the Legendary feature to color the PC 1 vs PC 2 scatterplot
sns.scatterplot(data=poke_cat_df,
                  x='PC 1', y='PC 2', hue='Legendary')
plt.show()
```





**Observation:** Looks like the different types are scattered all over the place while the legendary Pokemon always score high for PC 1 meaning they have high stats overall. Their spread along the PC 2 axis tells us they aren't consistently fast and vulnerable or slow and armored.

## PCA in a model pipeline

We just saw that legendary Pokemon tend to have higher stats overall. Let's see if we can add a classifier to our pipeline that detects legendary versus non-legends Pokemon based on the principal components.

```
# Build the pipeline
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('reducer', PCA(n_components=2)),
    ('classifier', RandomForestClassifier(random_state=0))])

# Fit the pipeline to the training data
pipe.fit(X_train, y_train)

# Prints the explained variance ratio
print(pipe.steps[1][1].explained_variance_ratio_)
# [0.45624044 0.17767414]

# Score the accuracy on the test set
accuracy = pipe.score(X_test, y_test)

# Prints the model accuracy
print('{0:.1%} test set accuracy'.format(accuracy))
```

```
<script.py> output:  
95.8% test set accuracy
```

Repeat the process with 3 extracted components.

```
# Build the pipeline
pipe = Pipeline([
    ('scaler', StandardScaler()),
    ('reducer', PCA(n_components=3)),
    ('classifier', RandomForestClassifier(random_state=0))])

# Fit the pipeline to the training data
pipe.fit(X_train, y_train)

# Score the accuracy on the test set
accuracy = pipe.score(X_test, y_test)

# Prints the explained variance ratio and accuracy
print(pipe.steps[1][1].explained_variance_ratio_)
print('{0:.1%} test set accuracy'.format(accuracy))
```

```
<script.py> output:  
[0.45624044 0.17767414 0.12858833]  
95.0% test set accuracy
```

Observation: Looks like adding the third component does not increase the model accuracy, even though it adds information to the dataset.

## Principal Component selection

Tell PCA the minimal proportion of variance we want to keep, and let the algorithm decide on the number of components it needs to achieve that, eg. `PCA(n_components=0.9)` to explain 90% of the variance.

## An optimal number of components

```
pipe.fit(poke_df)

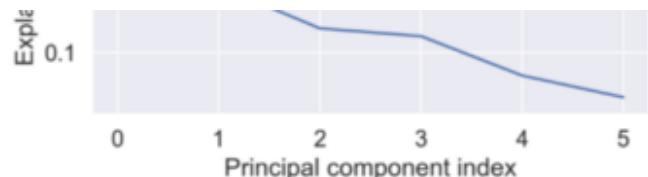
var = pipe.steps[1][1].explained_variance_ratio_

plt.plot(var)

plt.xlabel('Principal component index')
```



```
plt.ylabel('Explained variance ratio')
plt.show()
```



Note that the x-axis is zero indexed. Elbow at principal component index 1 means to keep 2 principal components.

## Selecting the proportion of variance to keep

You'll let PCA determine the number of components to calculate based on an explained variance threshold that you decide. You'll work on the numeric ANSUR female dataset

```
ansur_df .
```

```
# Pipe a scaler to PCA selecting 80% of the variance
pipe = Pipeline([('scaler', StandardScaler()),
                 ('reducer', PCA(n_components=0.8))])

# Fit the pipe to the data
pipe.fit(ansur_df)

print('{} components selected'.format(len(pipe.steps[1])
                                         [1].components_))
```

```
<script.py> output:
    11 components selected
```

Increase the proportion of variance to keep to 90%.

```
# Let PCA select 90% of the variance
pipe = Pipeline([('scaler', StandardScaler()),
                 ('reducer', PCA(n_components=0.9))])

# Fit the pipe to the data
pipe.fit(ansur_df)

print('{} components selected'.format(len(pipe.steps[1])
                                         [1].components_))
```

```
<script.py> output:
```

23 components selected

How many **additional** features do you need to explain 90% instead of 80% of the variance?

Answer: 12. We need to more than double the components to go from 80% to 90% explained variance.

## Choosing the number of components

You'll now make a more informed decision on the number of principal components to reduce your data to using the “elbow in the plot” technique. One last time, you'll work on the numeric ANSUR female dataset `ansur_df`.

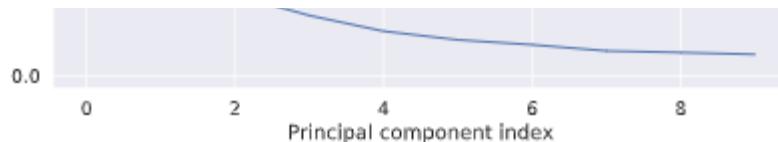
```
# Pipeline a scaler and pca selecting 10 components
pipe = Pipeline([('scaler', StandardScaler()),
                 ('reducer', PCA(n_components=10))])

# Fit the pipe to the data
pipe.fit(ansur_df)

# Plot the explained variance ratio
plt.plot(pipe.steps[1][1].explained_variance_ratio_)

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





To how many components can you reduce the dataset without compromising too much on explained variance?

Answer: 3.

The ‘elbow’ in the plot is at 3 components (the 3rd component has index 2).

## PCA for image compression

You’ll reduce the size of 16 images with hand written digits (MNIST dataset) using PCA.

The samples are 28 by 28 pixel gray scale images that have been flattened to arrays with 784 elements each ( $28 \times 28 = 784$ ) and added to the 2D numpy array `x_test`. Each of the 784 pixels has a value between 0 and 255 and can be regarded as a feature.

A pipeline with a scaler and PCA model to select 78 components has been coded as `pipe`. This pipeline has already been fitted to the entire MNIST dataset, except for the 16 samples in `x_test`.

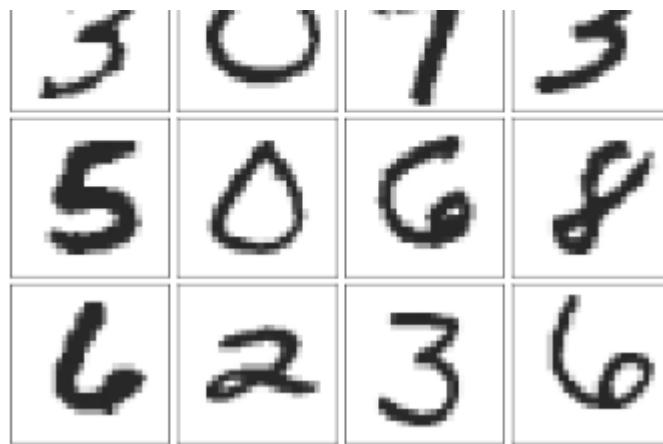
Finally, a function `plot_digits` has been created for you that will plot 16 images in a grid.

```
# Pipeline a scaler and pca selecting 10 components
pipe = Pipeline([('scaler', StandardScaler()),
                 ('reducer', PCA(n_components=78))])

# Fit the pipe to the data
pipe.fit(ansur_df)

# Plot the MNIST sample data
plot_digits(X_test)
```





```
# Transform the input data to principal components
pc = pipe.transform(X_test)

# Prints the number of features per dataset
print("X_test has {} features".format(X_test.shape[1]))
print("pc has {} features".format(pc.shape[1]))
```

```
<script.py> output:
    X_test has 784 features
    pc has 78 features
```

Inverse transform the components back to the original feature space.

```
# Transform the input data to principal components
pc = pipe.transform(X_test)

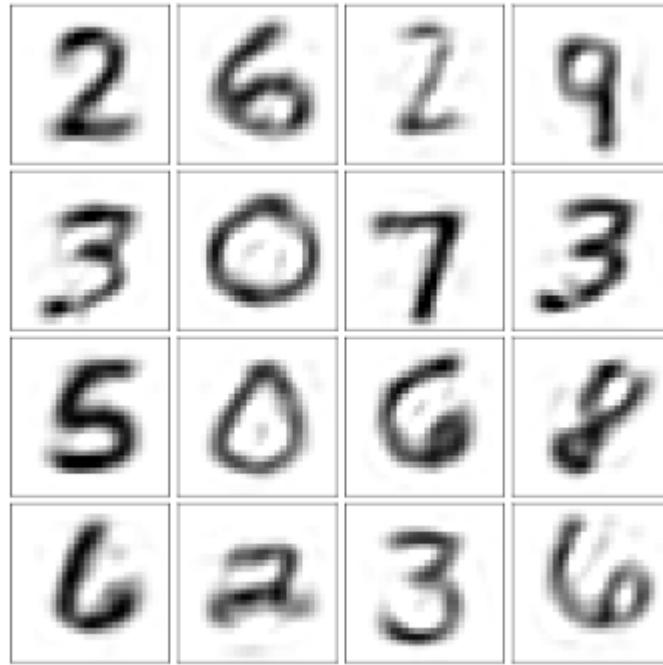
# Inverse transform the components to original feature space
X_rebuilt = pipe.inverse_transform(pc)

# Prints the number of features
print("X_rebuilt has {} features".format(X_rebuilt.shape[1]))
```

```
<script.py> output:
    X_rebuilt has 784 features
```

Plot the reconstructed data.

```
# Plot the reconstructed data  
plot_digits(X_rebuilt)
```



Observation: The size of the data has been reduced by 10 fold, but the images could be reconstructed with reasonable quality.

## Congratulations!

What you have learnt:

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

---

**Sign up for The Daily Pick**

By Towards Data Science

Hands-on real-world examples, research, tutorials, and cutting-edge techniques delivered Monday to Thursday. Make learning your daily ritual. [Take a look](#)

[Get this newsletter](#)

Emails will be sent to jamesngyh@gmail.com.  
[Not you?](#)

Python

Principal Component

Dimensionality Reduction

Feature Extraction

Data

[About](#) [Help](#) [Legal](#)

Get the Medium app

