**1. Why generate features?**

00:00 - 00:11

Hello and welcome to Feature Engineering for Machine Learning in Python. My name is Robert O’Callaghan and I am a Data Scientist.

**2. Feature Engineering**

00:11 - 00:48

Feature engineering is the act of taking raw data and extracting features from it that are suitable for tasks like machine learning. Most machine learning algorithms work with tabular data. When we talk about features, we are referring to the information stored in the columns of these tables. For example, if we were looking at information on houses, the features would be things like square foot, number of rooms, etc. This course is designed for data scientists who want to expand their knowledge of how to incorporate feature engineering into their data science workflow.

**3. Different types of data**

00:48 - 01:39

Most machine learning algorithms require their input data to be represented as a vector or a matrix, and many assume that the data is distributed normally. In the real world, more often than not you will receive data that is not in this format. You will also need to work with many different types of data, some data types you will often encounter are: continuous variables, categorical data, ordinal data, boolean values, and dates and times. Dealing with these is manageable, but requires a well thought out approach. Feature engineering is often overlooked in machine learning discussions, but any real-world practitioner will confirm that data manipulation and feature engineering is the most important aspect of the project.

**4. Course structure**

01:39 - 02:16

Over the span of this course, we will be addressing how to deal with many different types of data and how to convert them into a format that can be easily used for machine learning. In the first chapter, you will ingest and create basic features from tabular data. In the second chapter, you will learn how to deal with data that has missing values. You will then move on to transforming your data so that it conforms to statistical assumptions often necessary for machine learning models, and finally, you will convert free form text into tabular data so it can be used with machine learning models.

**5. Pandas**

02:16 - 02:48

Now lets jump straight in with some examples. During this course we will be leveraging the pandas package substantially as it is very useful when working with data in tabular form. It is a common practice to import pandas using the pd alias. You can use the read\_csv() function to import a CSV file and use the head() method to quickly look at the first few rows of the DataFrame.

**6. Dataset**

02:48 - 03:06

For the first three chapters of this course, you will be working with a modified subset of the Stackoverflow survey response data. This dataset records the details and preferences of hundreds of users of the StackOverflow website.

**7. Column names**

03:06 - 03:17

To see the features used in this subset, you can use the DataFrame columns attribute to print the names of all the columns in the DataFrame.

**8. Column types**

03:17 - 03:32

To print the data type of each column, you can use the dtypes attribute. Here you can see three different data types - integers, floats and objects - in pandas objects are columns that contain strings.

**9. Selecting specific data types**

03:32 - 03:58

Knowing the types of each column can be very useful if you are performing analysis based on a subset of specific data types. To do this, you can use the select\_dtypes() method and pass a list of relevant data types to the include argument. For example, if you want to select only the integer columns, call the select\_dtypes() method on df and set the include argument to 'int'.

**10. Lets get going!**

03:58 - 04:04

Lets get right into it and start practicing.

# Getting to know your data

Pandas is one the most popular packages used to work with tabular data in Python. It is generally imported using the alias pd and can be used to load a CSV (or other delimited files) using read\_csv().

You will be working with a modified subset of the [Stackoverflow survey response data](https://insights.stackoverflow.com/survey/2018/#overview) in the first three chapters of this course. This dataset records the details, and preferences of thousands of users of the StackOverflow website.

## Instructions 1/4

Import the pandas library as pd.

so\_survey\_csv contains the URL to a CSV file. Import it using Pandas into so\_survey\_df.

# Import pandas

import pandas as pd

# Import so\_survey\_csv into so\_survey\_df

so\_survey\_df = pd.read\_csv(so\_survey\_csv)

Print the first five rows of so\_survey\_df.

# Print the first five rows of the DataFrame

print(so\_survey\_df.head(5))

**# Import pandas**

**import pandas as pd**

**# Import so\_survey\_csv into so\_survey\_df**

**so\_survey\_df = pd.read\_csv(so\_survey\_csv)**

**# Print the first five rows of the DataFrame**

**print(so\_survey\_df.head(5))**

**SurveyDate FormalEducation ConvertedSalary Hobby Country ... VersionControl Age Years Experience Gender RawSalary**

**0 2/28/18 20:20 Bachelor's degree (BA. BS. B.Eng.. etc.) NaN Yes South Africa ... Git 21 13 Male NaN**

**1 6/28/18 13:26 Bachelor's degree (BA. BS. B.Eng.. etc.) 70841.0 Yes Sweeden ... Git;Subversion 38 9 Male 70,841.00**

**2 6/6/18 3:37 Bachelor's degree (BA. BS. B.Eng.. etc.) NaN No Sweeden ... Git 45 11 NaN NaN**

**3 5/9/18 1:06 Some college/university study without earning ... 21426.0 Yes Sweeden ... Zip file back-ups 46 12 Male 21,426.00**

**4 4/12/18 22:41 Bachelor's degree (BA. BS. B.Eng.. etc.) 41671.0 Yes UK ... Git 39 7 Male £41,671.00**

Print the data type of each column in so\_survey\_df.

# Print the data type of each column

print(so\_survey\_df.dtypes)

SurveyDate object FormalEducation object ConvertedSalary float64 Hobby object Country object StackOverflowJobsRecommend float64 VersionControl object Age int64 Years Experience int64 Gender object RawSalary object dtype: object

## Question

What type of data is the ConvertedSalary column?

### Possible answers

Datetime

Numeric

String

Boolean

Correct! ConvertedSalary contains floats which are numeric.

# Selecting specific data types

Often a dataset will contain columns with several different data types (like the one you are working with). The majority of machine learning models require you to have a consistent data type across features. Similarly, most feature engineering techniques are applicable to only one type of data at a time. For these reasons among others, you will often want to be able to access just the columns of certain types when working with a DataFrame.

The DataFrame (so\_survey\_df) from the previous exercise is available in your workspace.

## Instructions

100 XP

* Create a subset of so\_survey\_df consisting of only the numeric (int and float) columns.
* Print the column names contained in so\_survey\_df\_num.
* **# Create subset of only the numeric columns**
* **so\_numeric\_df = so\_survey\_df.select\_dtypes(include=['int', 'float'])**
* **# Print the column names contained in so\_survey\_df\_num**
* **print(so\_numeric\_df.columns)**
* **Index(['ConvertedSalary', 'StackOverflowJobsRecommend', 'Age', 'Years Experience'], dtype='object')**

**Well done! In the next lesson, you will learn the most common ways of dealing with categorical data.**

**1. Dealing with Categorical Variables**

00:00 - 00:25

Categorical variables are used to represent groups that are qualitative in nature. Some examples are colors, such as blue, red, black etc. or country of birth, such as Ireland, England or USA. While these can easily be understood by a human, you will need to encode categorical features as numeric values to use them in your machine learning models.

**2. Encoding categorical features**

00:25 - 01:04

As an example, here is a table which consists of the country of residence of different respondents in the Stackoverflow survey. To get from qualitative inputs to quantitative features, one may naively think that assigning every category in a column a number would suffice, for example India could be 1, USA 2 etc. But these categories are unordered, so assigning this order may greatly penalize the effectiveness of your model. Thus, you cannot allocate arbitrary numbers to each category as that would imply some form of ordering in the categories.

**3. Encoding categorical features**

01:04 - 01:28

Instead, values can be encoded by creating additional binary features corresponding to whether each value was picked or not as shown in the table on the right. In doing so your model can leverage the information of what country is given, without inferring any order between the different options.

**4. Encoding categorical features**

01:28 - 01:46

There are two main approaches when representing categorical columns in this way, one hot encoding and dummy encoding. These are very similar and often confused. In fact, by default, pandas performs one-hot encoding when you use the get\_dummies() function.

**5. One-hot encoding**

01:46 - 02:20

One-hot encoding converts n categories into n features as shown here. You can use the get\_dummies() function to one-hot encode columns. The function takes a DataFrame and a list of categorical columns you want converted into one hot encoded columns, and returns an updated DataFrame with these columns included. Specifying a prefix with the prefix argument can improve readability like the letter C for country has been used here.

**6. Dummy encoding**

02:20 - 02:58

On the other hand, dummy encoding creates n-1 features for n categories, omitting the first category. Notice that this time there is no feature for France, the first category. In dummy encoding, the base value, France in this case, is encoded by the absence of all other countries as you can see on the last row here and its value is represented by the intercept. For dummy encoding, you can use the same get\_dummies() function with an additional argument, drop\_first set to True as shown here.

**7. One-hot vs. dummies**

02:58 - 03:25

Both these methods have different advantages. One-hot encoding generally creates much more explainable features, as each country will have its own weight that can be observed after training. But one must be aware that one hot encoding may create features that are entirely collinear due to the same information being represented multiple times.

**8. One-hot vs. dummies**

03:25 - 03:50

Take for example a simpler categorical column recording the sex of the survey takers. By recording a 1 for male the information of whether the person is female is already known when the male column is 0. This double representation can lead to instability in your models and dummy values would be more appropriate.

**9. Limiting your columns**

03:50 - 04:18

However, both one-hot encoding and dummy encoding may result in a huge number of columns being created if there are too many different categories in a column. In these cases, you may want to only create columns for the most common values. You can check the number of occurrences of different features in a column using the value\_counts() method on a specific column.

**10. Limiting your columns**

04:18 - 04:57

Once you have your counts of occurrences, you can use it to limit what values you will include by first creating a mask of the values that occur less than n times. A mask is a list of booleans outlining which values in a column should be affected. First we find the categories that occur less than n times using the index attribute and wrap this inside the isin() method. After you create the mask, you can use it to replace these categories that occur less than n times with a value of your choice as shown here.

**11. Now you deal with categorical variables**

04:57 - 05:05

Lets put what has been learned into practice and work with some categorical variables.

# One-hot encoding and dummy variables

To use categorical variables in a machine learning model, you first need to represent them in a quantitative way. The two most common approaches are to one-hot encode the variables using or to use dummy variables. In this exercise, you will create both types of encoding, and compare the created column sets. We will continue using the same DataFrame from previous lesson loaded as so\_survey\_df and focusing on its Country column.

## Instructions 1/2

One-hot encode the Country column, adding "OH" as a prefix for each column.

Create dummy variables for the Country column, adding "DM" as a prefix for each column.

**# Convert the Country column to a one hot encoded Data Frame**

**one\_hot\_encoded = pd.get\_dummies(so\_survey\_df, columns=['Country'], prefix='OH')**

**# Print the columns names**

**print(one\_hot\_encoded.columns)**

**Index(['SurveyDate', 'FormalEducation', 'ConvertedSalary', 'Hobby', 'StackOverflowJobsRecommend', 'VersionControl', 'Age', 'Years Experience', 'Gender', 'RawSalary', 'OH\_France', 'OH\_India',**

**'OH\_Ireland', 'OH\_Russia', 'OH\_South Africa', 'OH\_Spain', 'OH\_Sweeden', 'OH\_UK', 'OH\_USA', 'OH\_Ukraine'],**

**# Create dummy variables for the Country column**

**dummy = pd.get\_dummies(so\_survey\_df, columns=['Country'], drop\_first=True, prefix='DM')**

**# Print the columns names**

**print(dummy.columns)**

**<script.py> output:**

**Index(['SurveyDate', 'FormalEducation', 'ConvertedSalary', 'Hobby', 'StackOverflowJobsRecommend', 'VersionControl', 'Age', 'Years Experience', 'Gender', 'RawSalary', 'DM\_India', 'DM\_Ireland',**

**'DM\_Russia', 'DM\_South Africa', 'DM\_Spain', 'DM\_Sweeden', 'DM\_UK', 'DM\_USA', 'DM\_Ukraine'],**

**dtype='object')**

**Great job! Did you notice that the column for France was missing when you created dummy variables? Now you can choose to use one-hot encoding or dummy variables where appropriate.**

# Dealing with uncommon categories

Some features can have many different categories but a very uneven distribution of their occurrences. Take for example Data Science's favorite languages to code in, some common choices are Python, R, and Julia, but there can be individuals with bespoke choices, like FORTRAN, C etc. In these cases, you may not want to create a feature for each value, but only the more common occurrences.

## Instructions 1/3

Extract the Country column of so\_survey\_df as a series and assign it to countries.

Find the counts of each category in the newly created countries series.

# Create a series out of the Country column

countries = so\_survey\_df[‘Country’]

# Get the counts of each category

country\_counts = countries.\_\_\_\_

# Print the count values for each category

print(country\_counts)

* Create a mask for values occurring less than 10 times in country\_counts.
* Print the first 5 rows of the mask.

**<script.py> output:**

**South Africa 166**

**USA 164**

**Spain 134**

**Sweeden 119**

**France 115**

**Russia 97**

**UK 95**

**India 95**

**Ukraine 9**

**Ireland 5**

**Name: Country, dtype: int64**

**# Create a series out of the Country column**

**countries = so\_survey\_df['Country']**

**# Get the counts of each category**

**country\_counts = countries.value\_counts()**

**# Create a mask for only categories that occur less than 10 times**

**mask = countries.isin(country\_counts[country\_counts<10].index)**

**# Print the top 5 rows in the mask series**

**print(mask.head(5))**

**0 False**

**1 False**

**2 False**

**3 False**

**4 False**

**Name: Country, dtype: bool**

* Label values occurring less than the mask cutoff as 'Other'.
* Print the new category counts in countries.

Good work, now you can work with large datasets while grouping low frequency categories.

**# Create a series out of the Country column**

**countries = so\_survey\_df['Country']**

**# Get the counts of each category**

**country\_counts = countries.value\_counts()**

**# Create a mask for only categories that occur less than 10 times**

**mask = countries.isin(country\_counts[country\_counts < 10].index)**

**# Label all other categories as Other**

**countries[mask] = 'Other'**

**# Print the updated category counts**

**print(pd.value\_counts(countries))**

**<script.py> output:**

**South Africa 166**

**USA 164**

**Spain 134**

**Sweeden 119**

**France 115**

**Russia 97**

**UK 95**

**India 95**

**Other 14**

**Name: Country, dtype: int64**

**1. Numeric variables**

00:00 - 00:17

As mentioned in the previous lesson, most machine learning models will require your data to be in numeric format. However, even if your raw data is all numeric, there is still a lot you can do to improve your features.

**2. Types of numeric features**

00:17 - 00:54

Numeric features can be used to represent a huge array of different characteristics and measurements. Pretty much anything that can be quantitatively measured can be recorded as numeric data. For example, age, the price of an item, counts, and even spatial data such as coordinates. Depending on the use case, numeric features can be treated in several different ways. We will work through a few of the considerations and possible feature engineering steps to keep in mind when dealing with numeric data.

**3. Does size matter?**

00:54 - 01:51

One of the first questions you should ask when working with numeric features is whether the magnitude of the feature is its most important trait, or just its direction. For example, if you had a dataset of restaurant health and safety ratings containing the number of times a restaurant had major violations, you might care far more about whether the restaurant had any major violations at all (as you would rather not take any chances), over whether it was a repeat offender. Looking at this toy dataset containing restaurant IDs and the number of times they had major violations, we can see that some restaurants have no major violations but many have one or more. We will be creating a new binary column representing whether or not a restaurant committed any violation.

**4. Binarizing numeric variables**

01:51 - 02:10

Here we first create a new column Binary\_Violation and set it to zero. Then, we use the dot loc notation to find all rows where Number\_of\_Violations is greater than zero and set the Binary\_Violation column to 1.

**5. Binarizing numeric variables**

02:10 - 02:30

As you can see here, all rows where Number\_of\_Violations is equal to 0 are also zeros in Binary\_Violation. However, for all rows where Number\_of\_Violations is greater than zero is 1 in Binary\_Violation.

**6. Binning numeric variables**

02:30 - 03:33

An extension of this is perhaps you wish to group a numeric variable into more than two bins. This is often useful for variables such as age, wage brackets, etc where exact numbers are less relevant than the general magnitude of the value. Consider the same dataset of restaurant health and safety ratings containing the number of times a restaurant has had major violations. This time we will be creating three groups, Group 1, for restaurants with no offenses, Group 2 for restaurants with one or two offenses and group 3 for all restaurants with three or more offenses. Bins are created by using the pandas' cut() function. You can define the intervals using the bins argument as shown here, which in this case is a list of 4 values. You can also pass a list of labels like so.

**7. Binning numeric variables**

03:33 - 03:54

Note as we want to include 0 in the first bin, we must set the leftmost edge to lower than that, so all values between negative infinity and 0 are labeled as 1, all values equal to 1 or 2 are labeled as 2, and values greater than 2 are labeled as 3.

**8. Lets start practicing!**

03:54 - 04:06

Now you know how to binarize and bin numeric columns, it's time for you to put this into practice.

# Binarizing columns

While numeric values can often be used without any feature engineering, there will be cases when some form of manipulation can be useful. For example on some occasions, you might not care about the magnitude of a value but only care about its direction, or if it exists at all. In these situations, you will want to binarize a column. In the so\_survey\_df data, you have a large number of survey respondents that are working voluntarily (without pay). You will create a new column titled Paid\_Job indicating whether each person is paid (their salary is greater than zero).

## Instructions

100 XP

* Create a new column called Paid\_Job filled with zeros.
* Replace all the Paid\_Job values with a 1 where the corresponding ConvertedSalary is greater than 0.
* **# Create the Paid\_Job column filled with zeros**
* **so\_survey\_df['Paid\_Job'] = 0**
* **# Replace all the Paid\_Job values where ConvertedSalary is > 0**
* **so\_survey\_df.loc[so\_survey\_df['ConvertedSalary']>0, 'Paid\_Job'] = 1**
* **# Print the first five rows of the columns**
* **print(so\_survey\_df[['Paid\_Job', 'ConvertedSalary']].head(5))**
* **<script.py> output:**
* **Paid\_Job ConvertedSalary**
* **0 0 0.0**
* **1 1 70841.0**
* **2 0 0.0**
* **3 1 21426.0**
* **4 1 41671.0**

**Good work, binarizing columns can also be useful for your target variables.**

# Binning values

For many continuous values you will care less about the exact value of a numeric column, but instead care about the bucket it falls into. This can be useful when plotting values, or simplifying your machine learning models. It is mostly used on continuous variables where accuracy is not the biggest concern e.g. age, height, wages.

Bins are created using pd.cut(df['column\_name'], bins) where bins can be an integer specifying the number of evenly spaced bins, or a list of bin boundaries.

## Instructions 1/2

Bin the value of the ConvertedSalary column in so\_survey\_df into 5 equal bins, in a new column called equal\_binned.

Bin the ConvertedSalary column using the boundaries in the list bins and label the bins using labels.

**# Bin the continuous variable ConvertedSalary into 5 bins**

**so\_survey\_df['equal\_binned'] = pd.cut(so\_survey\_df['ConvertedSalary'], bins=5)**

**# Print the first 5 rows of the equal\_binned column**

**print(so\_survey\_df[['equal\_binned', 'ConvertedSalary']].head())**

**# Import numpy**

**import numpy as np**

**# Specify the boundaries of the bins**

**bins = [-np.inf, 10000, 50000, 100000, 150000, np.inf]**

**# Bin labels**

**labels = ['Very low', 'Low', 'Medium', 'High', 'Very high']**

**# Bin the continuous variable ConvertedSalary using these boundaries**

**so\_survey\_df['boundary\_binned'] = \_\_\_\_(so\_survey\_df['ConvertedSalary'],**

**\_\_\_\_, \_\_\_\_)**

**# Print the first 5 rows of the boundary\_binned column**

**print(so\_survey\_df[['boundary\_binned', 'ConvertedSalary']].head())**

**# Import numpy**

**import numpy as np**

**# Specify the boundaries of the bins**

**bins = [-np.inf, 10000, 50000, 100000, 150000, np.inf]**

**# Bin labels**

**labels = ['Very low', 'Low', 'Medium', 'High', 'Very high']**

**# Bin the continuous variable ConvertedSalary using these boundaries**

**so\_survey\_df['boundary\_binned'] = pd.cut(so\_survey\_df['ConvertedSalary'],**

**bins=bins, labels=labels)**

**# Print the first 5 rows of the boundary\_binned column**

**print(so\_survey\_df[['boundary\_binned', 'ConvertedSalary']].head(5))**

**<script.py> output:**

**equal\_binned ConvertedSalary**

**0 (-2000.0, 400000.0] 0.0**

**1 (-2000.0, 400000.0] 70841.0**

**2 (-2000.0, 400000.0] 0.0**

**3 (-2000.0, 400000.0] 21426.0**

**4 (-2000.0, 400000.0] 41671.0**

**# Import numpy**

**import numpy as np**

**# Specify the boundaries of the bins**

**bins = [-np.inf, 10000, 50000, 100000, 150000, np.inf]**

**# Bin labels**

**labels = ['Very low', 'Low', 'Medium', 'High', 'Very high']**

**# Bin the continuous variable ConvertedSalary using these boundaries**

**so\_survey\_df['boundary\_binned'] = pd.cut(so\_survey\_df['ConvertedSalary'],**

**bins=bins, labels=labels)**

**# Print the first 5 rows of the boundary\_binned column**

**print(so\_survey\_df[['boundary\_binned', 'ConvertedSalary']].head(5))**

**<script.py> output:**

**boundary\_binned ConvertedSalary**

**0 Very low 0.0**

**1 Medium 70841.0**

**2 Very low 0.0**

**3 Low 21426.0**

**4 Low 41671.0**

**Correct, now you can bin columns with equal spacing and predefined boundaries.**

**1. Why do missing values exist?**

00:00 - 00:17

In the first chapter, we looked at the different types of data one may find when analyzing data. In this lesson, we will explore the concept of messy and missing values, how to find them, and once identified, how to deal with them.

**2. How gaps in data occur**

00:17 - 01:02

While in an ideal world every dataset you come across would be perfectly complete and contain no gaps, unfortunately, this is rarely the case. Real world data often has noise or omissions. This can stem from many sources, for example: Data not being collected properly (paper surveys not being filled out fully). Collection and management errors (someone transcribing the data making a mistake). Data intentionally being omitted (people may want to skip the age box in an online form). Or gaps could be created due to transformations of the data (average of a field with missing data). This list is far from comprehensive.

**3. Why we care?**

01:02 - 01:49

You may wonder why are we discussing this? Does missing data even matter? Yes, it does, and it is extremely important to identify and deal with missing data. Many machine learning models cannot work with missing values, for example if you were performing linear regression, you would need a value for every row and column used in your dataset. Missing data may be indicative of a problem in your data pipeline. If data is consistently missing in a certain column, you should investigate as to why this is the case. Missing data may provide information in itself. For example, if the number of children of a person is missing they may have no children.

**4. Missing value discovery**

01:49 - 02:19

You can use the info() method to have a preliminary look at how complete the dataset is. Right from the get go you can see that the StackOverflowJobsRecommend, Gender, and RawSalary columns are highly underpopulated and we should examine where these missing values occur. This list output is useful but becomes limited with larger datasets that have missing values scattered all over their features.

**5. Finding missing values**

02:19 - 02:32

To find where these missing values exist, you can use the isnull() method as shown here. All cells where missing values exist are shown as True.

**6. Finding missing values**

02:32 - 02:42

You can also count the number of missing values in a specific column by chaining the isnull() and sum() methods as shown here.

**7. Finding non-missing values**

02:42 - 03:05

The inverse (or the non missing values) can also be found using the notnull() method. Here, all missing values are shown as False. Note that you can call the isnull() and notnull() methods on both the DataFrame as a whole, and on each of it's individual columns.

**8. Go ahead and find missing values!**

03:05 - 03:14

It's time for you to find missing values in the Stackoverflow data!

## Exercise

# How sparse is my data?

Most datasets contain missing values, often represented as NaN (Not a Number). If you are working with Pandas you can easily check how many missing values exist in each column.

Let's find out how many of the developers taking the survey chose to enter their age (found in the Age column of so\_survey\_df) and their gender (Gender column of so\_survey\_df).

## Instructions 1/2

Subset the DataFrame to only include the 'Age' and 'Gender' columns.

Print the number of non-missing values in both columns.

# Subset the DataFrame

sub\_df = so\_survey\_df[['Age', 'Gender']]

# Print the number of non-missing values

print(sub\_df.notnull())

**# Subset the DataFrame**

**sub\_df = so\_survey\_df[['Age', 'Gender']]**

**# Print the number of non-missing values**

**print(sub\_df.notnull().sum())**

**Age 999**

**Gender 693**

**dtype: int64**

## Question

Based on the results, how many non-missing entries are there in the Gender column?

### Possible answers

999

**693**

**Correct, there are 693 non-missing entries in the Gender column.**

# Finding the missing values

While having a summary of how much of your data is missing can be useful, often you will need to find the exact locations of these missing values. Using the same subset of the StackOverflow data from the last exercise (sub\_df), you will show how a value can be flagged as missing.

## Instructions 1/3

Print the first 10 entries of the DataFrame.

 Print the locations of the missing values in the first 10 rows.

 Print the locations of the non-missing values in the first 10 rows.

**# Print the top 10 entries of the DataFrame**

**print(sub\_df.head(10))**

**Age Gender**

**0 21 Male**

**1 38 Male**

**2 45 NaN**

**3 46 Male**

**4 39 Male**

**5 39 Male**

**6 34 Male**

**7 24 Female**

**8 23 Male**

**9 36 NaN**

**# Print the locations of the missing values**

**print(sub\_df.head(10).isnull())**

**# Print the locations of the missing values**

**print(sub\_df.head(10).isnull())**

**Age Gender**

**0 False False**

**1 False False**

**2 False True**

**3 False False**

**4 False False**

**5 False False**

**6 False False**

**7 False False**

**8 False False**

**9 False True**

**# Print the locations of the non-missing values**

**print(sub\_df.head(10).notnull())**

**# Print the locations of the non-missing values**

**print(sub\_df.head(10).notnull())**

**Age Gender**

**0 True True**

**1 True True**

**2 True False**

**3 True True**

**4 True True**

**5 True True**

**6 True True**

**7 True True**

**8 True True**

**9 True False**

**Well done, finding where the missing values exist can often be important.**

**1. Dealing with missing values (I)**

00:00 - 00:10

Now that you can recognize why missing values occur and how to locate them, you need to know how they can be dealt with.

**2. Listwise deletion**

00:10 - 00:51

If you are confident that the missing values in your dataset are occurring at random, (in other words not being intentionally omitted) the most effective and statistically sound approach to dealing with them is called 'complete case analysis' or listwise deletion. In this method, a record is fully excluded from your model if any of its values are missing. Take for example the dataset shown here. Although most of the information is available in the first and third rows, because values in the ConvertedSalary column are missing, these rows will be dropped.

**3. Listwise deletion in Python**

00:51 - 01:06

To implement listwise deletion using pandas, you can use the dropna() method, by setting the how argument to 'any'. This will delete all rows with at least one missing value.

**4. Listwise deletion in Python**

01:06 - 01:22

On the other hand, if you want to delete rows with missing values in only a specific column, you can use the subset argument. Pass a list of columns to this argument to specify which columns to consider when deleting rows.

**5. Issues with deletion**

01:22 - 01:54

While the preferable approach in situations where missing data occurs purely at random is listwise deletion, it does have its drawbacks. First, it deletes perfectly valid data points that share a row with a missing value. Second, if the missing values do not occur entirely at random it can negatively affect the model. Lastly, if you were to remove a feature instead of a row it can reduce the degrees of freedom of your model.

**6. Replacing with strings**

01:54 - 02:30

The most common way to deal with missing values is to simply fill these values using the fillna() method. To use the fillna() method on a specific column, you need to provide the value you want to replace the missing values with. In the case of categorical columns, it is common to replace missing values with strings like 'Other', 'Not Given' etc. To replace the missing values in place, in other words to modify the original DataFrame, you need to set the inplace argument to True.

**7. Recording missing values**

02:30 - 03:12

In situations where you believe that the absence or presence of data is more important than the values themselves, you can create a new column that records the absence of data and then drop the original column. To do this, all you need to do is call the notnull() method on a specific column. This will output a list of True/False values, thus recording the presence/absence of data. To drop columns from a DataFrame, you can use the drop() method and specify a list of column names which you want to drop as the columns argument.

**8. Practice time**

03:12 - 03:26

With this in mind you will now work through applying listwise deletion, and some alternatives for replacing missing values in categorical columns.

# Listwise deletion

The simplest way to deal with missing values in your dataset when they are occurring entirely at random is to remove those rows, also called 'listwise deletion'.

Depending on the use case, you will sometimes want to remove all missing values in your data while other times you may want to only remove a particular column if too many values are missing in that column.

## Instructions ¼

## Print the number of rows and columns in so\_survey\_df.

Drop all rows with missing values in so\_survey\_df.

Drop all columns with missing values in so\_survey\_df.

Drop all rows in so\_survey\_df where 'Gender' is missing

**# Print the number of rows and columns**

**print(so\_survey\_df.shape)**

**(999, 11)**

**# Create a new DataFrame dropping all incomplete rows**

**no\_missing\_values\_rows = so\_survey\_df.dropna(how='any')**

**# Print the shape of the new DataFrame**

**print(no\_missing\_values\_rows.shape)**

**(264, 11)**

**# Create a new DataFrame dropping all columns with incomplete rows**

**no\_missing\_values\_cols = so\_survey\_df.dropna(how='any', axis=1)**

**# Print the shape of the new DataFrame**

**print(no\_missing\_values\_cols.shape)**

**(999, 7)**

**# Drop all rows where Gender is missing**

**no\_gender = so\_survey\_df.dropna(subset=['Gender'])**

**# Print the shape of the new DataFrame**

**print(no\_gender.shape)**

**<script.py> output:**

**(693, 11)**

**Correct, as you can see dropping all rows that contain any missing values may greatly reduce the size of your dataset. So you need to think carefully and consider several trade-offs when deleting missing values.**

# Replacing missing values with constants

While removing missing data entirely maybe a correct approach in many situations, this may result in a lot of information being omitted from your models.

You may find categorical columns where the missing value is a valid piece of information in itself, such as someone refusing to answer a question in a survey. In these cases, you can fill all missing values with a new category entirely, for example 'No response given'.

## Instructions 1/2

Print the count of occurrences of each category in so\_survey\_df's Gender column.

**# Print the count of occurrences**

**print(so\_survey\_df['Gender'].value\_counts())**

**Male 632**

**Female 53**

**Female;Male 2**

**Transgender 2**

**Female;Male;Transgender;Non-binary. genderqueer. or gender non-conforming 1**

**Male;Non-binary. genderqueer. or gender non-conforming 1**

**Non-binary. genderqueer. or gender non-conforming 1**

**Female;Transgender 1**

**Name: Gender, dtype: int64**

Replace all missing values in the Gender column with the string 'Not Given'. Make changes to the original DataFrame.

**# Replace missing values**

**so\_survey\_df['Gender'].fillna(value='Not Given', inplace=True)**

**# Print the count of each value**

**print(so\_survey\_df['Gender'].value\_counts())**

**Male 632**

**Not Given 306**

**Female 53**

**Female;Male 2**

**Transgender 2**

**Female;Male;Transgender;Non-binary. genderqueer. or gender non-conforming 1**

**Male;Non-binary. genderqueer. or gender non-conforming 1**

**Non-binary. genderqueer. or gender non-conforming 1**

**Female;Transgender 1**

**Name: Gender, dtype: int64**

**Wonderful! By filling in these missing values you can use the columns in your analyses.**

**1. Fill continuous missing values**

00:00 - 00:15

While listwise deletion is often the most statistically sound method of dealing with missing values in cases where you believe the gaps are at random, this will often not be feasible in real world use cases.

**2. Deleting missing values**

00:15 - 00:41

One of the most common issues with removing all rows with missing values is if you were building a predictive model. If you were to remove all cases that had missing values when training your model, you would quickly run into problems when you received missing values in your test set, where you do not have the option of just not predicting these rows.

**3. What else can you do?**

00:41 - 01:11

So what's the alternative? Replacing missing values. For categorical columns, as you saw in the last lesson you can either replace missing values with a string that flags missing values such as 'None', or you can use the most common occurring value. However, for numeric columns, you may want to replace missing values with a more suitable value. So what is a suitable value?

**4. Measures of central tendency**

01:11 - 01:50

In cases like this we often turn to the measures of central tendency, which are the central or typical value for a distribution. The most commonly used values are the mean and the median. One caveat that you must keep in mind when using these methods is that it can lead to biased estimates of the variances and covariances of the features. Similarly, the standard error and test statistics can be incorrectly estimated so if these metrics are needed they should be calculated before the missing values have been filled.

**5. Calculating the measures of central tendency**

01:50 - 02:07

You can calculate these measures directly from a pandas series by simply calling the required method on the series as shown here. Note that the missing values are excluded by default when calculating these statistics.

**6. Fill the missing values**

02:07 - 02:41

Then leveraging what you implemented in previous lesson, you can directly fill all missing values using the fillna() method. Only this time you are filling missing values in the ConvertedSalary column with the mean of this column. Since you filled in the missing values with the mean, you may end up with too many decimal places. You can get rid of all the decimal values by changing the data type to integer using the astype() method like so.

**7. Rounding values**

02:41 - 02:49

or you can first round the mean before filling in the missing values as shown here.

**8. Let's Practice!**

02:49 - 02:57

Now its your turn to put what you have learned into practice.

# Filling continuous missing values

In the last lesson, you dealt with different methods of removing data missing values and filling in missing values with a fixed string. These approaches are valid in many cases, particularly when dealing with categorical columns but have limited use when working with continuous values. In these cases, it may be most valid to fill the missing values in the column with a value calculated from the entries present in the column.

## Instructions 1/3

Print the first five rows of the StackOverflowJobsRecommend column of so\_survey\_df.

**# Print the first five rows of StackOverflowJobsRecommend column**

**print(so\_survey\_df['StackOverflowJobsRecommend'].head(5))**

**0 NaN**

**1 7.0**

**2 8.0**

**3 NaN**

**4 8.0**

**Name: StackOverflowJobsRecommend, dtype: float64**

Replace the missing values in the StackOverflowJobsRecommend column with its mean. Make changes directly to the original DataFrame.

**so\_survey\_df['StackOverflowJobsRecommend'].fillna(so\_survey\_df['StackOverflowJobsRecommend'].mean(), inplace=True)**

**# Print the first five rows of StackOverflowJobsRecommend column**

**print(so\_survey\_df['StackOverflowJobsRecommend'].head())**

**0 7.062**

**1 7.000**

**2 8.000**

**3 7.062**

**4 8.000**

**Name: StackOverflowJobsRecommend, dtype: float64**

Round the decimal values that you introduced in the StackOverflowJobsRecommend column.

**# Fill missing values with the mean**

**so\_survey\_df['StackOverflowJobsRecommend'].fillna(so\_survey\_df['StackOverflowJobsRecommend'].mean(), inplace=True)**

**# Round the StackOverflowJobsRecommend values**

**so\_survey\_df['StackOverflowJobsRecommend'] = round(so\_survey\_df['StackOverflowJobsRecommend'])**

**# Print the top 5 rows**

**print(so\_survey\_df['StackOverflowJobsRecommend'].head())**

**0 7.0**

**1 7.0**

**2 8.0**

**3 7.0**

**4 8.0**

**Name: StackOverflowJobsRecommend, dtype: float64**

**<script.py> output:**

**0 7.0**

**1 7.0**

**2 8.0**

**3 7.0**

**4 8.0**

**Name: StackOverflowJobsRecommend, dtype: float64**

Nicely done, remember you should only round your values if you are certain it is applicable.

# Imputing values in predictive models

When working with predictive models you will often have a separate train and test DataFrames. In these cases you want to ensure no information from your test set leaks into your train set. When filling missing values in data to be used in these situations how should approach the two datasets?

##### Answer the question

#### Possible Answers

* Only fill the train set.
* Only fill the test set.
* Apply the measures of central tendency (mean/median etc.) calculated on the train set to both the train and test sets.
* Apply the measures of central tendency (mean/median etc.) calculated on the test set to both the train and test sets.
* Apply the measures of central tendency (mean/median etc.) calculated on the train set to the train set, and the measures calculated on the test set, to the test set.

Correct, values calculated on the train test should be applied to both DataFrames.

**1. Dealing with other data issues**

00:00 - 00:37

Up to this point you have used multiple approaches to creating and updating features when missing values are present in the data, but data issues are of course not limited to just this. In some instances, you will come across features that need to be updated in some other way. Take for example the case of a column containing a monetary value. If this dataset has been imported from excel it may contain characters such as currency signs or commas that prevents pandas from reading it as numeric values.

**2. Bad characters**

00:37 - 00:52

For example, lets look at the data type of the RawSalary column. It's an object, although intuitively, you know that it should be numeric. So why is that?

**3. Bad characters**

00:52 - 01:03

Let's take a quick peek at the data. Numeric columns should not contain any non-numeric characters. So you need to remove these commas.

**4. Dealing with bad characters**

01:03 - 03:41

Although you want the column to be a numeric column, it is of type object, which means you can use string methods to fix this column. In this case, we want to remove all occurrences of comma. We can easily achieve this by accessing the str accessor and using the replace() method. The first argument is the string you want to replace, which is the comma, and the second argument is the string you want to replace it with, which here is an empty string, which simply means you want to remove all the commas. However, the data type of this column is still object. Now you can convert your column to the relevant type as shown here.

**5. Finding other stray characters**

00:00 - 02:31

But what if attempting to change the data type raises an error? This may indicate that there are additional stray characters which you didn't account for. Instead of manually searching for values with other stray characters you can use the to\_numeric() function from pandas along with the errors argument. If you set the errors argument to 'coerce', Pandas will convert the column to numeric, but all values that can't be converted to numeric will be changed to NaNs, that is missing values.

**6. Finding other stray characters**

02:31 - 02:47

You can now use the isna() method like you did earlier to find out which values failed to parse. So it looks like we also have dollar signs. You can again use the replace() method as before to remove the dollar signs.

**7. Chaining methods**

02:47 - 03:31

Before you get going onto trying these for yourself, it will be useful to delve a little deeper into method chaining. If you are applying different methods or in fact the same method several times on a column, instead of assigning the result back to the column after each iteration, you can simply chain the methods, that is, call one method after the other to obtain the desired result. For example, cleaning up characters, changing the data type, normalizing the values etc. can all be achieved by simply calling the methods one after the other as seen here.

**8. Go ahead and fix bad characters!**

03:31 - 03:41

Now that you know how to deal with stray characters, let's put it into practice.

# Dealing with stray characters (I)

In this exercise, you will work with the RawSalary column of so\_survey\_df which contains the wages of the respondents along with the currency symbols and commas, such as $42,000. When importing data from Microsoft Excel, more often that not you will come across data in this form.

## Instructions 1/2

Remove the commas (,) from the RawSalary column

Remove the dollar ($) signs from the RawSalary column.

**# Remove the commas in the column**

**so\_survey\_df['RawSalary'] = so\_survey\_df['RawSalary'].str.replace(',', '')**

**# Remove the dollar signs in the column**

**so\_survey\_df['RawSalary'] = so\_survey\_df['RawSalary'].str.replace('$', '')**

Congratulations! Replacing/removing specific characters is a very useful skill.

# Dealing with stray characters (II)

In the last exercise, you could tell quickly based off of the df.head() call which characters were causing an issue. In many cases this will not be so apparent. There will often be values deep within a column that are preventing you from casting a column as a numeric type so that it can be used in a model or further feature engineering.

One approach to finding these values is to force the column to the data type desired using pd.to\_numeric(), coercing any values causing issues to NaN, Then filtering the DataFrame by just the rows containing the NaN values.

Try to cast the RawSalary column as a float and it will fail as an additional character can now be found in it. Find the character and remove it so the column can be cast as a float.

## Instructions 1/2

Attempt to convert the RawSalary column of so\_survey\_df to numeric values coercing all failures into null values.

Find the indexes of the rows containing NaNs.

Print the rows in RawSalary based on these indexes.

**# Attempt to convert the column to numeric values**

**numeric\_vals = pd.to\_numeric(so\_survey\_df['RawSalary'], errors='coerce')**

**# Find the indexes of missing values**

**idx = numeric\_vals.isna()**

**# Print the relevant rows**

**print(so\_survey\_df['RawSalary'][idx])**

**0 NaN**

**2 NaN**

**4 £41671.00**

**6 NaN**

**8 NaN**

**...**

**989 NaN**

**990 NaN**

**992 NaN**

**994 NaN**

**997 NaN**

**Name: RawSalary, Length: 401, dtype: object**

* Did you notice the pound (£) signs in the RawSalary column? Remove these signs like you did in the previous exercise.
* Convert the RawSalary column to float.
* **# Replace the offending characters**
* **so\_survey\_df['RawSalary'] = so\_survey\_df['RawSalary'].str.replace('£','')**
* **# Convert the column to float**
* **so\_survey\_df['RawSalary'] = so\_survey\_df['RawSalary'].astype('float')**
* **# Print the column**
* **print(so\_survey\_df['RawSalary'])**

**<script.py> output:**

* **0 NaN**
* **1 70841.0**
* **2 NaN**
* **3 21426.0**
* **4 41671.0**
* **...**
* **994 NaN**
* **995 58746.0**
* **996 55000.0**
* **997 NaN**
* **998 1000000.0**
* **Name: RawSalary, Length: 999, dtype: float64**

Nicely done! Remember that even after removing all the relevant characters, you still need to change the type of the column to numeric if you want to plot these continuous values.

# Method chaining

When applying multiple operations on the same column (like in the previous exercises), you made the changes in several steps, assigning the results back in each step. However, when applying multiple successive operations on the same column, you can "chain" these operations together for clarity and ease of management. This can be achieved by calling multiple methods sequentially:

# Method chaining

df['column'] = df['column'].method1().method2().method3()

# Same as

df['column'] = df['column'].method1()

df['column'] = df['column'].method2()

df['column'] = df['column'].method3()

In this exercise you will repeat the steps you performed in the last two exercises, but do so using method chaining.

## Instructions

100 XP

* Remove the commas (,) from the RawSalary column of so\_survey\_df.
* Remove the dollar ($) signs from the RawSalary column.
* Remove the pound (£) signs from the RawSalary column.
* Convert the RawSalary column to float.

**# Use method chaining**

**so\_survey\_df['RawSalary'] = so\_survey\_df['RawSalary']\**

**.str.replace(',','')\**

**.str.replace('$','')\**

**.str.replace('£','')\**

**.astype('float')**

**# Print the RawSalary column**

**print(so\_survey\_df['RawSalary'])**

**<script.py> output:**

**0 NaN**

**1 70841.0**

**2 NaN**

**3 21426.0**

**4 41671.0**

**...**

**994 NaN**

**995 58746.0**

**996 55000.0**

**997 NaN**

**998 1000000.0**

**Name: RawSalary, Length: 999, dtype: float64**

Great job! Custom functions can be also used when method chaining using the .apply() method.

**1. Data distributions**

00:00 - 00:39

An important consideration before building a machine learning model is to understand what the distribution of your underlying data looks like. A lot of algorithms make assumptions about how your data is distributed or how different features interact with each other. For example almost all models besides tree based models require your features to be on the same scale. Feature engineering can be used to manipulate your data so that it can fit the assumptions of the distribution, or at least fit it as closely as possible.

**2. Distribution assumptions**

00:39 - 01:09

Almost every model besides tree based models assume that your data is normally distributed. Normal distributions follow a bell shape like shown here, the main characteristics of a normal distribution is that 68 percent of the data lies within 1 standard deviation of the mean,95% percent lies within 2 standard deviations from the mean and 99.7% fall within 3 standard deviations from the mean.

**3. Observing your data**

01:09 - 01:45

To understand the shape of your own data you can create histograms of each of the continuous features. To do so, once you have the matplotlib library loaded, along with your DataFrame, run hist() on your data frame followed by calling plt dot show to observe the graph. Here we see the first column has a fairly normal looking distribution, but the second looks quite different, with the majority of the data skewed to the lower values. This is also referred to having a long right tail.

**4. Delving deeper with box plots**

01:45 - 02:39

While histograms can be useful to show the high level distribution of the data, it does not show details such as where the middle chunk of your data sits in an easily readable fashion. For this you can use the box plot. The box plot shows the distribution of the data by calculating where the middle 50% of the data sits, this is also known as the Inter quartile range or IQR (it sits between the 1st and 3rd quartile) and marking it with the box. The whiskers extend to the minimum of 1.5 times the IQR from the edge of the box or the maximum range of the data. Any points outside this are marked as outliers. This can be useful to also see if there are points in your dataset that may be unwanted outliers.

**5. Box plots in pandas**

02:39 - 02:49

To create a box plot in pandas, you can call the boxplot() method on a list of columns you wish to plot.

**6. Paring distributions**

02:49 - 03:29

One final approach to looking at the distribution of data is to examine how different features in your DataFrame interact with each other. This type of chart is called a pairplot and can be useful to see if multiple columns are correlated with each other or whether they have any association at all. To generate a pairplot, first you need to import the seaborn package and then call the pairplot() function on your DataFrame. In this example we can see that the first and last columns are somewhat related.

**7. Further details on your distributions**

03:29 - 03:52

While all these plots are very useful to get an understanding of your data's shape, you will at times want to quickly get summary statistics of your data's distribution. This can be found using the describe() method as seen here on the same dummy dataset we have been using to demonstrate the plots.

**8. Let's practice!**

03:52 - 04:09

Why is this important? Now that you are capable of seeing the underlying structure of the data, in later lessons, you will remove outliers and ensure all features are on comparable scales.

## Instructions 1/3

Generate a histogram of all columns in the so\_numeric\_df DataFrame.

**# Create a histogram**

**so\_numeric\_df.hist()**

**plt.show()**

Generate box plots of the Age and Years Experience columns in the so\_numeric\_df DataFrame.

# Create a boxplot of two columns

**so\_numeric\_df[['Age', 'Years Experience']].boxplot()**

**plt.show()**

Generate a box plot of the ConvertedSalary column in the so\_numeric\_df DataFrame.

# Create a boxplot of ConvertedSalary

**so\_numeric\_df[['ConvertedSalary']].boxplot()**

**plt.show()**

Great, as you can see the distrbutions of columns in a dataset can vary quite a bit.

# What does your data look like? (II)

In the previous exercise you looked at the distribution of individual columns. While this is a good start, a more detailed view of how different features interact with each other may be useful as this can impact your decision on what to transform and how.

## Instructions 1/2

Import matplotlib's pyplot module as plt.

Import seaborn as sns.

Plot pairwise relationships in the so\_numeric\_df dataset.

Show the plot.

**# Import packages**

**import matplotlib.pyplot as plt**

**import seaborn as sns**

**# Plot pairwise relationships**

**sns.pairplot(so\_numeric\_df)**

**# Show plot**

**plt.show()**

Print the summary statistics of the so\_numeric\_df DataFrame.

**Print(so\_numeric\_df.describe())**

**ConvertedSalary Age Years Experience**

**count 9.990e+02 999.000 999.000**

**mean 6.162e+04 36.003 9.962 std 1.761e+05 13.255 4.878**

**min 0.000e+00 18.000 0.000**

**25% 0.000e+00 25.000 7.000**

**50% 2.712e+04 35.000 10.000**

**75% 7.000e+04 45.000 13.000**

**max 2.000e+06 83.000 27.000**

**Good work, understanding these summary statistics of a column can be very valuable when deciding what transformations are necessary.**

# When don't you have to transform your data?

While making sure that all of your data is on the same scale is advisable for most analyses, for which of the following machine learning models is normalizing data not always necessary?

##### Answer the question

* K-Means
* Decision Trees
* Linear Regression
* K-nearest neighbors

**Correct! As decision trees split along a singular point, they do not require all the columns to be on the same scale.**

**1. Scaling and transformations**

00:00 - 00:09

As mentioned in the last video, most machine learning algorithms require your data to be on the same scale for them to be effective,

**2. Scaling data**

00:09 - 00:46

For example it is difficult to compare salary values (often measured in thousands) with ages as shown here. While this assumption of similar scales is necessary, it is rarely true in real world data. For this reason you need to rescale your data to ensure that it is on the same scale. There are many different approaches to doing this but we will discuss the two most commonly used approaches here, Min-Max scaling (sometimes referred to as normalization), and standardization.

**3. Min-Max scaling**

00:46 - 01:15

Min-Max scaling is when your data is scaled linearly between a minimum and maximum value, often 0 and 1, with 0 corresponding with the lowest value in the column, and 1 with the largest. As it is a linear scaling while the values will change, the distribution will not. Take for example the Age column from the stackoverflow dataset, the raw values lie between 20 and 80, approximately.

**4. Min-Max scaling**

01:15 - 01:26

While here after min-max scaling, although the distribution is the same, the values sit fully between 0 and 1.

**5. Min-Max scaling in Python**

01:26 - 02:06

To implement this on your dataset, you first need to import MinMaxScaler from scikit learn's preprocessing module, scikit learn is the most commonly used machine learning library for python. You then instantiate the MinMaxScaler() and fit it to your data. This tells the scaler how it should scale values when it performs the transformation. Finally, you need to actually transform the data with this new fitted scaler. Note that as this scaler assumes the max value it is created with is your upper bound, new data from outside this range may create unforeseen results.

**6. Standardization**

02:06 - 02:44

The other commonly used scaler is called standardization. As opposed to finding an outer boundary and squeezing everything within it, standardization instead finds the mean of your data and centers your distribution around it, calculating the number of standard deviations away from the mean each point is. These values (the number of standard deviations) are then used as your new values. This centers the data around 0 but technically has no limit to the maximum and minimum values as you can see here.

**7. Standardization in Python**

02:44 - 03:03

You can apply standardization in a similar fashion to how the min-max scaler was implemented. You first import StandardScaler from scikit-learn, instantiate and then fit it on your data. Once fitted you can apply it to your data.

**8. Log Transformation**

03:03 - 03:30

Both normalization and min-max scaling are types of scalers, in other words the data remained in the same shape but was squashed or scaled. A log transformation on the other hand can be used to make highly skewed distributions less skewed. Take for example one of the salary columns from the stack overflow dataset shown here where there is a very long right tail.

**9. Log transformation in Python**

03:30 - 04:02

Although it effects your data quite differently, a log transformation is implemented in Python the same way you have implemented scalers. To use a log transform you first import PowerTransformer from sklearn's preprocessing module, then you fit it to your dataset, and once fitted you can transform your data. Log transformation is a type of power transformation, hence the name.

**10. Final Slide**

04:02 - 04:14

Now it is your turn to apply these three techniques to the data you are familiar with, and see what the transformed data looks like.

# Normalization

As discussed in the video, in normalization you linearly scale the entire column between 0 and 1, with 0 corresponding with the lowest value in the column, and 1 with the largest.  
When using scikit-learn (the most commonly used machine learning library in Python) you can use a MinMaxScaler to apply normalization. (It is called this as it scales your values between a minimum and maximum value.)

## Instructions

100 XP

* Import MinMaxScaler from sklearn's preprocessing module.
* Instantiate the MinMaxScaler() as MM\_scaler.
* Fit the MinMaxScaler on the Age column of so\_numeric\_df.
* Transform the same column with the scaler you just fit.
* **# Import MinMaxScaler**
* **from sklearn.preprocessing import MinMaxScaler**
* **# Instantiate MinMaxScaler**
* **MM\_scaler = MinMaxScaler()**
* **# Fit MM\_scaler to the data**
* **MM\_scaler.fit(so\_numeric\_df[['Age']])**
* **# Transform the data using the fitted scaler**
* **so\_numeric\_df['Age\_MM'] = MM\_scaler.transform(so\_numeric\_df[['Age']])**
* **# Compare the origional and transformed column**
* **print(so\_numeric\_df[['Age\_MM', 'Age']].head())**
* **<script.py> output:**
* **Age\_MM Age**
* **0 0.046 21**
* **1 0.308 38**
* **2 0.415 45**
* **3 0.431 46**
* **4 0.323 39**
* **Wohoo! Did you notice that all values have been scaled between 0 and 1?**

# Standardization

While normalization can be useful for scaling a column between two data points, it is hard to compare two scaled columns if even one of them is overly affected by outliers. One commonly used solution to this is called standardization, where instead of having a strict upper and lower bound, you center the data around its mean, and calculate the number of standard deviations away from mean each data point is.

## Instructions

100 XP

* Import StandardScaler from sklearn's preprocessing module.
* Instantiate the StandardScaler() as SS\_scaler.
* Fit the StandardScaler on the Age column of so\_numeric\_df.
* Transform the same column with the scaler you just fit.
* **# Import StandardScaler**
* **from sklearn.preprocessing import StandardScaler**
* **# Instantiate StandardScaler**
* **SS\_scaler = StandardScaler()**
* **# Fit SS\_scaler to the data**
* **SS\_scaler.fit(so\_numeric\_df[['Age']])**
* **# Transform the data using the fitted scaler**
* **so\_numeric\_df['Age\_SS'] = SS\_scaler.transform(so\_numeric\_df[['Age']])**
* **# Compare the origional and transformed column**
* **print(so\_numeric\_df[['Age\_SS', 'Age']].head())**
* **<script.py> output:**
* **Age\_SS Age**
* **0 -1.132 21**
* **1 0.151 38**
* **2 0.679 45**
* **3 0.755 46**
* **4 0.226 39**
* **Correct, you can see that the values have been scaled linearly, but not between set values.**

# Log transformation

In the previous exercises you scaled the data linearly, which will not affect the data's shape. This works great if your data is normally distributed (or closely normally distributed), an assumption that a lot of machine learning models make. Sometimes you will work with data that closely conforms to normality, e.g the height or weight of a population. On the other hand, many variables in the real world do not follow this pattern e.g, wages or age of a population. In this exercise you will use a log transform on the ConvertedSalary column in the so\_numeric\_df DataFrame as it has a large amount of its data centered around the lower values, but contains very high values also. These distributions are said to have a long right tail.

## Instructions

100 XP

* Import PowerTransformer from sklearn's preprocessing module.
* Instantiate the PowerTransformer() as pow\_trans.
* Fit the PowerTransformer on the ConvertedSalary column of so\_numeric\_df.
* Transform the same column with the scaler you just fit.

**# Import PowerTransformer**

**from sklearn.preprocessing import PowerTransformer**

**# Instantiate PowerTransformer**

**pow\_trans = PowerTransformer()**

**# Train the transform on the data**

**pow\_trans.fit(so\_numeric\_df[['ConvertedSalary']])**

**# Apply the power transform to the data**

**so\_numeric\_df['ConvertedSalary\_LG'] = pow\_trans.transform(so\_numeric\_df[['ConvertedSalary']])**

**# Plot the data before and after the transformation**

**so\_numeric\_df[['ConvertedSalary', 'ConvertedSalary\_LG']].hist()**

**plt.show()**

**Superb! Did you notice the change in the shape of the distribution?**

**ConvertedSalary\_LG column looks much more normal than the original ConvertedSalary column.**

# When can you use normalization?

When could you use normalization (MinMaxScaler) when working with a dataset?

##### Answer the question

#### Possible Answers

When you know that your data may have outliers.

When you know the the data has a strict upper and lower bound.

When you know that your test data may contain smaller values than your training data.

When you know that your test data may contain larger values than your training data.

**Correct! Normalization scales all points linearly between the upper and lower bound.**

**1. Removing outliers**

00:00 - 00:13

You will often find that even after performing these transformations, your data is still very skewed. This can often be caused by outliers existing in your data.

**2. What are outliers?**

00:13 - 00:40

Outliers are data points that exist far away from the majority of your data. This can happen due to several reasons, such as incorrect data recording to genuine rare occurrences. Either way you will often want to remove these values as they can negatively impact your models. An example of the negative effect can be seen here where an outlier is causing almost all of the scaled data to be squashed to the lower bound.

**3. Quantile based detection**

00:40 - 01:19

The first approach we will discuss is to remove a certain percentage of the largest and/or smallest values in your data. For example you could remove the top 5%. This is achieved by finding the 95th quantile (the point below which 95% of your data resides) and removing everything above it. This approach is particularly useful if you are concerned that the highest values in your dataset should be avoided. When using this approach, you must remember that even if there are no real outliers, you will still be removing the top 5% of values from the dataset.

**4. Quantiles in Python**

01:19 - 01:36

To find the 95th quantile, you can call the quantile() method with 0.95 as the argument on the column. You can then create a mask to find which values lie below the 95th quantile and subset the data accordingly.

**5. Standard deviation based detection**

01:36 - 02:09

An alternative, and perhaps more statistically sound method of removing outliers is to instead choose what you consider to be outliers based on the mean and standard deviations of the dataset. For example you may want to eliminate all data greater than 3 standard deviations from the mean as you expect those data points to be outliers. This approach has the benefit of only removing genuinely extreme values, for example if only one value was an outlier, only that value would be effected.

**6. Standard deviation detection in Python**

02:09 - 02:54

To apply this in Python, you first need to find the mean and standard deviation of your column by calling the mean() and std() methods on the column, respectively. You then find upper bound by adding 3 times the standard deviation to the mean and similarly find the lower bound by subtracting 3 times the standard deviation from the mean. Once you have found these bounds, you can apply these bounds as a mask to the DataFrame as shown here. This method ensures that only data that is genuinely different from the rest is removed, and will remove fewer points if the data is close together.

**7. Let's practice!**

02:54 - 03:04

Now it's time for you to put what you have learned about outliers into practice.

# Percentage based outlier removal

One way to ensure a small portion of data is not having an overly adverse effect is by removing a certain percentage of the largest and/or smallest values in the column. This can be achieved by finding the relevant quantile and trimming the data using it with a mask. This approach is particularly useful if you are concerned that the highest values in your dataset should be avoided. When using this approach, you must remember that even if there are no outliers, this will still remove the same top N percentage from the dataset.

## Instructions

100 XP

* Find the 95th quantile of the ConvertedSalary column.
* Trim the so\_numeric\_df DataFrame to retain all rows where ConvertedSalary is less than it's 95th quantile.
* Plot the histogram of so\_numeric\_df[['ConvertedSalary']].
* Plot the histogram of trimmed\_df[['ConvertedSalary']].
* **# Find the 95th quantile**
* **quantile = so\_numeric\_df['ConvertedSalary'].quantile(0.95)**
* **# Trim the outliers**
* **trimmed\_df = so\_numeric\_df[so\_numeric\_df['ConvertedSalary'] < quantile]**
* **# The original histogram**
* **so\_numeric\_df[['ConvertedSalary']].hist()**
* **plt.show()**
* **plt.clf()**
* **# The trimmed histogram**
* **trimmed\_df[['ConvertedSalary']].hist()**
* **plt.show()**
* **Nicely done! In the next exercise, you will work with a more statistically sound approach in removing outliers.**

# Statistical outlier removal

While removing the top N% of your data is useful for ensuring that very spurious points are removed, it does have the disadvantage of always removing the same proportion of points, even if the data is correct. A commonly used alternative approach is to remove data that sits further than three standard deviations from the mean. You can implement this by first calculating the mean and standard deviation of the relevant column to find upper and lower bounds, and applying these bounds as a mask to the DataFrame. This method ensures that only data that is genuinely different from the rest is removed, and will remove fewer points if the data is close together.

## Instructions

100 XP

* Calculate the standard deviation and mean of the ConvertedSalary column of so\_numeric\_df.
* Calculate the upper and lower bounds as three standard deviations away from the mean in both the directions.
* Trim the so\_numeric\_df DataFrame to retain all rows where ConvertedSalary is within the lower and upper bounds.
* **# Find the mean and standard dev**
* **std = so\_numeric\_df['ConvertedSalary'].std()**
* **mean = so\_numeric\_df['ConvertedSalary'].mean()**
* **# Calculate the cutoff**
* **cut\_off = std \* 3**
* **lower, upper = mean - cut\_off, mean + cut\_off**
* **# Trim the outliers**
* **trimmed\_df = so\_numeric\_df[(so\_numeric\_df['ConvertedSalary'] < upper)**
* **& (so\_numeric\_df['ConvertedSalary'] > lower)]**
* **# The trimmed box plot**
* **trimmed\_df[['ConvertedSalary']].boxplot()**
* **plt.show()**
* **Amazing job! Did you notice the scale change on the y-axis?**

**1. Scaling and transforming new data**

00:00 - 00:18

One of the most important aspects of machine learning is the actual application of any model you create on a new dataset. For example if you built a model based on historical data, ultimately you will want to apply this model on new data to make predictions.

**2. Reuse training scalers**

00:18 - 00:50

How you go about doing this depends on what transformations you make to the dataset before you fit the model. For example, if you applied the StandardScaler() to your data before fitting the model, you need to make sure you transform the test data using the same scalar before making predictions. Please do note that the scaler is fitted only on the training data. That is, you fit and transform the training data, but only transform the test data.

**3. Training transformations for reuse**

00:50 - 01:14

Similarly, if you intend to remove outliers from your test set, you should use the thresholds found on your train set to do so. If you were to use the mean and standard deviation of the test set, it could negatively impact your predictions. Note that it is only in very rare cases that you would want to remove outliers from your test set.

**4. Why only use training data?**

01:14 - 01:38

So why did we not refit the scaler on the test data or use thresholds from the test data? To avoid data leakage. In real life, you won't have access to the test data, that is, when you have deployed your model in production, you won't have access to future data, so you can't rely on it to make predictions and assess model performance.

**5. Avoid data leakage!**

01:38 - 01:53

Thus, you should always make sure you calibrate your preprocessing steps only on your training data or else you will overestimate the accuracy of your models.

# Train and testing transformations (I)

So far you have created scalers based on a column, and then applied the scaler to the same data that it was trained on. When creating machine learning models you will generally build your models on historic data (train set) and apply your model to new unseen data (test set). In these cases you will need to ensure that the same scaling is being applied to both the training and test data.  
To do this in practice you train the scaler on the train set, and keep the trained scaler to apply it to the test set. You should never retrain a scaler on the test set.

For this exercise and the next, we split the so\_numeric\_df DataFrame into train (so\_train\_numeric) and test (so\_test\_numeric) sets.

## Instructions

100 XP

* Instantiate the StandardScaler() as SS\_scaler.
* Fit the StandardScaler on the Age column.
* Transform the Age column in the test set (so\_test\_numeric).

**To do this in practice you train the scaler on the train set, and keep the trained scaler to apply it to the test set. You should never retrain a scaler on the test set.**

**# Import StandardScaler**

**from sklearn.preprocessing import StandardScaler**

**# Apply a standard scaler to the data**

**SS\_scaler = StandardScaler()**

**# Fit the standard scaler to the data**

**SS\_scaler.fit(so\_train\_numeric[['Age']])**

**# Transform the test data using the fitted scaler**

**so\_test\_numeric['Age\_ss'] = SS\_scaler.transform(so\_test\_numeric[['Age']])**

**print(so\_test\_numeric[['Age', 'Age\_ss']].head())**

**script.py> output:**

**Age Age\_ss**

**700 35 -0.069**

**701 18 -1.343**

**702 47 0.830**

**703 57 1.579**

**704 41 0.380**

**Great job! Data leakage is one of the most common mistakes data scientists tend to make, and I hope that you won't!**

# Train and testing transformations (II)

Similar to applying the same scaler to both your training and test sets, if you have removed outliers from the train set, you probably want to do the same on the test set as well. Once again you should ensure that you use the thresholds calculated only from the train set to remove outliers from the test set.

Similar to the last exercise, we split the so\_numeric\_df DataFrame into train (so\_train\_numeric) and test (so\_test\_numeric) sets.

## Instructions

100 XP

* Calculate the standard deviation and mean of the ConvertedSalary column.
* Calculate the upper and lower bounds as three standard deviations away from the mean in both the directions.
* Trim the so\_test\_numeric DataFrame to retain all rows where ConvertedSalary is within the lower and upper bounds.
* **1. Introduction to Text Encoding**
* 00:00 - 00:12
* So far in this course you have dealt with data that, while sometimes messy, has been generally columnar in nature. When you are faced with text data this is often not going to be the case.
* **2. Standardizing your text**
* 00:12 - 00:55
* Data that is not in a predefined form is called unstructured data, and free text data is a good example of this. Before you can leverage text data in a machine learning model you must first transform it into a series of columns of numbers or vectors. There are many different approaches to doing this and in this chapter we will go through the most common approaches. In this chapter, you will be working with the United States inaugural address dataset, which contains the text for each President's inaugural speech. With George Washington's shown here. It is clear that free text like this is not in tabular form.
* **3. Dataset**
* 00:55 - 01:16
* Before any text analytics can be performed, you must ensure that the text data is in a format that can be used. The speeches have been loaded as a pandas DataFrame called speech\_df, with the body of the text in the 'text' column as can be seen by looking at the top five rows using the head() method as shown.
* **4. Removing unwanted characters**
* 01:16 - 02:30
* Most bodies of text will have non letter characters such as punctuation, that will need to be removed before analysis. This can be achieved by using the replace() method along with the str accessor. We have used this in an earlier chapter, but instead of specifying the exact characters you wish to replace, this time you will use patterns called regular expressions. Now unless you go through the text of all speeches, it is difficult to determine which non-letter characters are present in the data. So the easiest way to deal with this to specify a pattern which replaces all non letter characters as shown here. The pattern lowercase a to lowercase z followed by uppercase A to uppercase Z inside square brackets basically indicates include all letter characters. Placing a caret before this pattern inside square brackets negates this, that is, says all non letter characters. So we use the replace() method with this pattern to replace all non letter characters with a white-space as shown here.
* **5. Removing unwanted characters**
* 02:30 - 02:41
* Here you can see the text of the first speech before and after processing. Notice that the hyphen and the colon are missing.
* **6. Standardize the case**
* 02:41 - 03:02
* Once all unwanted characters have been removed you will want to standardize the remaining characters in your text so that they are all lower case. This will ensure that the same word with and without capitalization will not be counted as separate words. You can use the lower() method to achieve this as shown here.
* **7. Length of text**
* 03:02 - 03:23
* Later in this chapter you will work through the creation of features based on the content of different texts, but often there is value in the fundamental characteristics of a passage, such as its length. Using the len() method, you can calculate the number of characters in each speech.
* **8. Word counts**
* 03:23 - 03:48
* Along with the pure character length of the speech, you may want to know how many words are contained in it. The most straight forward way to do this is to split the speech based an any white-spaces, and then count how many words there are after the split. First, you will need to split the text with with the split() method as shown here and
* **9. Word counts**
* 03:48 - 03:55
* then chain the len() method to count the total number of words in each speech.
* **10. Average length of word**
* 03:55 - 04:08
* Finally, one other stat you can calculate is the average word length. Since you already have the total number of characters and the word count, you can simply divide them to obtain the average word length.
* **11. Let's practice!**
* 04:08 - 04:19
* Now it's time for you to practice what you have learned about how to manipulate text.

# Cleaning up your text

Unstructured text data cannot be directly used in most analyses. Multiple steps need to be taken to go from a long free form string to a set of numeric columns in the right format that can be ingested by a machine learning model. The first step of this process is to standardize the data and eliminate any characters that could cause problems later on in your analytic pipeline.

In this chapter you will be working with a new dataset containing the inaugural speeches of the presidents of the United States loaded as speech\_df, with the speeches stored in the text column.

## Instructions 1/2

Print the first 5 rows of the text column to see the free text fields.

**# Print the first 5 rows of the text column**

**print(speech\_df['text'].head(5))**

**0 Fellow-Citizens of the Senate and of the House...**

**1 Fellow Citizens: I AM again called upon by th...**

**2 WHEN it was first perceived, in early times, t...**

**3 Friends and Fellow-Citizens: CALLED upon to u...**

**4 PROCEEDING, fellow-citizens, to that qualifica...**

**Name: text, dtype: object**

* Replace all non letter characters in the text column with a whitespace.
* Make all characters in the newly created text\_clean column lower case.
* **# Print the first 5 rows of the text\_clean column**
* **print(speech\_df['text\_clean'].head())**
* **0 fellow citizens of the senate and of the house...**
* **1 fellow citizens i am again called upon by th...**
* **2 when it was first perceived in early times t...**
* **3 friends and fellow citizens called upon to u...**
* **4 proceeding fellow citizens to that qualifica...**
* **Name: text\_clean, dtype: object**
* **Great, now your text strings have been standardized and cleaned up. You can now use this new column (text\_clean) to extract information about the speeches.**

# High level text features

Once the text has been cleaned and standardized you can begin creating features from the data. The most fundamental information you can calculate about free form text is its size, such as its length and number of words. In this exercise (and the rest of this chapter), you will focus on the cleaned/transformed text column (text\_clean) you created in the last exercise.

## Instructions

100 XP

* Record the character length of each speech in the char\_count column.
* Record the word count of each speech in the word\_count column.
* Record the average word length of each speech in the avg\_word\_length column.
* **# Find the length of each text**
* **speech\_df['char\_cnt'] = speech\_df['text\_clean'].str.len()**
* **# Count the number of words in each text**
* **speech\_df['word\_cnt'] = speech\_df['text\_clean'].str.split().str.len()**
* **# Find the average length of word**
* **speech\_df['avg\_word\_length'] = speech\_df['char\_cnt'] / speech\_df['word\_cnt']**
* **# Print the first 5 rows of these columns**
* **print(speech\_df[['text\_clean', 'char\_cnt', 'word\_cnt', 'avg\_word\_length']])**
* **text\_clean char\_cnt word\_cnt avg\_word\_length**
* **0 fellow citizens of the senate and of the house... 8616 1432 6.017**
* **1 fellow citizens i am again called upon by th... 787 135 5.830**
* **2 when it was first perceived in early times t... 13871 2323 5.971**
* **3 friends and fellow citizens called upon to u... 10144 1736 5.843**
* **4 proceeding fellow citizens to that qualifica... 12902 2169 5.948**
* **5 unwilling to depart from examples of the most ... 7003 1179 5.940**
* **6 about to add the solemnity of an oath to the o... 7148 1211 5.903**
* **7 i should be destitute of feeling if i was not ... 19894 3382 5.882**
* **8 fellow citizens i shall not attempt to descr... 26322 4466 5.894**
* **9 in compliance with an usage coeval with the ex... 17753 2922 6.076**
* **10 fellow citizens about to undertake the arduo... 6818 1130 6.034**
* These features may appear basic but can be quite useful in ML models.

**1. Word Count Representation**

00:00 - 00:10

Once high level information has been recorded you can begin creating features based on the actual content of each text.

**2. Text to columns**

00:10 - 00:36

The most common approach to this is to create a column for each word and record the number of times each particular word appears in each text. This results in a set of columns equal in width to the number of unique words in the dataset, with counts filling each entry. Taking just one sentence we can see that "of" occurs 3 tines, "the" 2 times and the other words once.

**3. Initializing the vectorizer**

00:36 - 01:00

While you could of course write a script to do this counting yourself, scikit-learn already has this functionality built in with its CountVectorizer class. As usual, first import CountVectorizer from sklearn dot feature\_extraction dot text, then instantiate it by assigning it to a variable name, cv in this case.

**4. Specifying the vectorizer**

01:00 - 01:48

It may have become apparent that creating a column for every word will result in far too many values for analyses. Thankfully, you can specify arguments when initializing your CountVectorizer to limit this. For example, you can specify the minimum number of texts that a word must be contained in using the argument min\_df. If a float is given, the word must appear in at least this percent of documents. This threshold eliminates words that occur so rarely that they would not be useful when generalizing to new texts. Conversely, max\_df limits words to only ones that occur below a certain percentage of the data. This can be useful to remove words that occur too frequently to be of any value.

**5. Fit the vectorizer**

01:48 - 02:00

Once the vectorizer has been instantiated you can then fit it on the data you want to create your features around. This is done by calling the fit() method on relevant column.

**6. Transforming your text**

02:00 - 02:15

Once the vectorizer has been fit you can call the transform() method on the column you want to transform. This outputs a sparse array, with a row for every text and a column for every word that has been counted.

**7. Transforming your text**

02:15 - 02:23

To transform this to a non sparse array you can use the toarray() method.

**8. Getting the features**

02:23 - 02:42

You may notice that the output is an array, with no concept of column names. To get the names of the features that have been generated you can call the get\_feature\_names() method on the vectorizer which returns a list of the features generated, in the same order that the columns of the converted array are in.

**9. Fitting and transforming**

02:42 - 02:59

As an aside, while fitting and transforming separately can be useful, particularly when you need to transform a different dataset than the one that you fit the vectorizer to, you can accomplish both steps at once using the fit\_transform() method.

**10. Putting it all together**

02:59 - 03:17

Now that you have an array containing the count values of each of the words of interest, and a way to get the feature names you can combine these in a DataFrame as shown here. The add\_prefix() method allows you to be able to distinguish these columns in the future.

1. 1 ```out Counts\_aback Counts\_abandon Counts\_abandonment 0 1 0 0 1 0 0 1 2 0 1 0 3 0 1 0 4 0 0 0 ```

**11. Updating your DataFrame**

03:17 - 03:42

You can now combine this DataFrame with your original DataFrame so they can be used to generate future analytical models using pandas concat method. Checking the DataFrames shape shows the new much wider size. Remember to specify the axis argument to 1 as you want column bind these DataFrames.

**12. Let's practice!**

# Counting words (I)

Once high level information has been recorded you can begin creating features based on the actual content of each text. One way to do this is to approach it in a similar way to how you worked with categorical variables in the earlier lessons.

* For each unique word in the dataset a column is created.
* For each entry, the number of times this word occurs is counted and the count value is entered into the respective column.

These "count" columns can then be used to train machine learning models.

## Instructions

100 XP

* Import CountVectorizer from sklearn.feature\_extraction.text.
* Instantiate CountVectorizer and assign it to cv.
* Fit the vectorizer to the text\_clean column.
* Print the feature names generated by the vectorizer.
* **# Import CountVectorizer**
* **from sklearn.feature\_extraction.text import CountVectorizer**
* **# Instantiate CountVectorizer**
* **cv = CountVectorizer()**
* **# Fit the vectorizer**
* **cv.fit(speech\_df['text\_clean'])**
* **# Print feature names**
* **print(cv.get\_feature\_names())**
* **['abandon', 'abandoned', 'abandonment', 'abate', 'abdicated', 'abeyance', 'abhorring',**

**Great, this vectorizer can be applied to both the text it was trained on, and new texts.**

# Counting words (II)

Once the vectorizer has been fit to the data, it can be used to transform the text to an array representing the word counts. This array will have a row per block of text and a column for each of the features generated by the vectorizer that you observed in the last exercise.

The vectorizer to you fit in the last exercise (cv) is available in your workspace.

## Instructions 1/2

Apply the vectorizer to the text\_clean column.

Convert this transformed (sparse) array into a numpy array with counts.

**# Apply the vectorizer**

**cv\_transformed = cv.transform(speech\_df['text\_clean'])**

**# Print the full array**

**cv\_array = cv\_transformed.toarray()**

**print(cv\_array)**

**Print the dimensions of this numpy array.**

**# Apply the vectorizer**

**cv\_transformed = cv.transform(speech\_df['text\_clean'])**

**# Print the full array**

**cv\_array = cv\_transformed.toarray()**

**# Print the shape of cv\_array**

**print(cv\_array.shape)**

**(58, 9043)**

**The speeches have 9043 unique words, which is a lot! In the next exercise, you will see how to create a limited set of features**.

# Limiting your features

As you have seen, using the CountVectorizer with its default settings creates a feature for every single word in your corpus. This can create far too many features, often including ones that will provide very little analytical value.

For this purpose CountVectorizer has parameters that you can set to reduce the number of features:

* min\_df : Use only words that occur in more than this percentage of documents. This can be used to remove outlier words that will not generalize across texts.
* max\_df : Use only words that occur in less than this percentage of documents. This is useful to eliminate very common words that occur in every corpus without adding value such as "and" or "the".

## Instructions

100 XP

* Limit the number of features in the CountVectorizer by setting the minimum number of documents a word can appear to 20% and the maximum to 80%.
* Fit and apply the vectorizer on text\_clean column in one step.
* Convert this transformed (sparse) array into a numpy array with counts.
* Print the dimensions of the new reduced array.
* **# Import CountVectorizer**
* **from sklearn.feature\_extraction.text import CountVectorizer**
* **# Specify arguements to limit the number of features generated**
* **cv = CountVectorizer(min\_df=.20,max\_df=.80)**
* **# Fit, transform, and convert into array**
* **cv\_transformed = cv.fit\_transform(speech\_df['text\_clean'])**
* **cv\_array = cv\_transformed.toarray()**
* **# Print the array shape**
* **print(cv\_array.shape)**
* **(58, 818)**

Did you notice that the number of features (unique words) greatly reduced from 9043 to 818?

# Text to DataFrame

Now that you have generated these count based features in an array you will need to reformat them so that they can be combined with the rest of the dataset. This can be achieved by converting the array into a pandas DataFrame, with the feature names you found earlier as the column names, and then concatenate it with the original DataFrame.

The numpy array (cv\_array) and the vectorizer (cv) you fit in the last exercise are available in your workspace.

## Instructions

100 XP

* Create a DataFrame cv\_df containing the cv\_array as the values and the feature names as the column names.
* Add the prefix Counts\_ to the column names for ease of identification.
* Concatenate this DataFrame (cv\_df) to the original DataFrame (speech\_df) column wise.
* **# Create a DataFrame with these features**
* **cv\_df = pd.DataFrame(cv\_transformed.toarray(),**
* **columns=cv.get\_feature\_names()).add\_prefix('Counts\_')**
* **# Add the new columns to the original DataFrame**
* **speech\_df\_new = pd.concat([speech\_df, cv\_df], axis=1, sort=False)**
* **print(speech\_df\_new.head())**
* **Name Inaugural Address Date text text\_clean ... Counts\_years \**
* **0 George Washington First Inaugural Address Thursday, April 30, 1789 Fellow-Citizens of the Senate and of the House... fellow citizens of the senate and of the house... ... 1**
* **1 George Washington Second Inaugural Address Monday, March 4, 1793 Fellow Citizens: I AM again called upon by th... fellow citizens i am again called upon by th... ... 0**
* **2 John Adams Inaugural Address Saturday, March 4, 1797 WHEN it was first perceived, in early times, t... when it was first perceived in early times t... ... 3**
* **3 Thomas Jefferson First Inaugural Address Wednesday, March 4, 1801 Friends and Fellow-Citizens: CALLED upon to u... friends and fellow citizens called upon to u... ... 0**
* **4 Thomas Jefferson Second Inaugural Address Monday, March 4, 1805 PROCEEDING, fellow-citizens, to that qualifica... proceeding fellow citizens to that qualifica... ... 2**
* **Counts\_yet Counts\_you Counts\_young Counts\_your**
* **0 0 5 0 9**
* **1 0 0 0 1**
* **2 0 0 0 1**
* **3 2 7 0 7**
* **4 2 4 0 4**
* **[5 rows x 826 columns]**
* With the new features combined with the orginial DataFrame they can be now used for ML models or analysis.

**1. Tf-Idf Representation**

00:00 - 00:19

While counts of occurrences of words can be a good first step towards encoding your text to build models, it has some limitations. The main issue is counts will be much higher for very common even when they occur across all texts, providing little value as a distinguishing feature.

**2. Introducing TF-IDF**

00:19 - 00:45

Take for example the counts of the word "the" shown here, with plentiful occurrences in every row. To limit these common words from overpowering your model some form of normalization can be used. One of the most effective approaches to do this is called "Term Frequency Inverse Document Frequency" or TF-IDF.

**3. TF-IDF**

00:45 - 01:05

TF-IDF divides number of times a word occurs in the document by a measure of what proportion of the documents a word occurs in all documents. This has the effect of reducing the value of common words, while increasing the weight of words that do not occur in many documents.

**4. Importing the vectorizer**

01:05 - 01:26

To use a TF-IDF vectorizer, the approach is very similar to how you applied a count vectorizer. First you must import TfidfVectorizer() from sklearn dot feature\_extraction dot text, then you assign it to a variable name. Lets use tv in this case.

**5. Max features and stopwords**

01:26 - 02:04

Similar to when you were working with the Count vectorizer where you could limit the number of features created by specifying arguments when initializing TfidfVectorizer, you can specify the maximum number of features using max\_features which will only use the 100 most common words. We will also specify the vectorizer to omit a set of stop\_words, these are a predefined list of the most common english words such as "and" or "the". You can use scikit-learn's built in list, load your own, or use lists provided by other python libraries.

**6. Fitting your text**

02:04 - 02:16

Once the vectorizer has been specified you can fit it, and apply it to the text that you want to transform. Note that here we are fitting and transforming the train data, a subset of the original data.

**7. Putting it all together**

02:16 - 02:25

As before, you combine the TF-IDF values along with the feature names in a DataFrame as shown here.

**8. Inspecting your transforms**

02:25 - 03:01

After transforming your data you should always check how the different words are being valued, and see which words are receiving the highest scores through the process. This will help you understand if the features being generated make sense or not. One ad hoc method is to isolate a single row of the transformed DataFrame (`tv\_df` in this case), using the iloc accessor, and then sorting the values in the row in descending order as shown here. These top ranked values make sense for the text of a presidential speech.

**9. Applying the vectorizer to new data**

03:01 - 03:34

So how do you apply this transformation on the test set? As mentioned before, you should preprocess your test data using the transformations made on the train data only. To ensure that the same features are created you should use the same vectorizer that you fit on the training data. So first transform the test data using the tv vectorizer and then recreate the test dataset by combining the TF-IDF values, feature names, and other columns.

**10. Let's practice!**

03:34 - 03:44

So, now you also know about TF-IDF! Great, it's time for you to implement this.

# Tf-idf

While counts of occurrences of words can be useful to build models, words that occur many times may skew the results undesirably. To limit these common words from overpowering your model a form of normalization can be used. In this lesson you will be using Term frequency-inverse document frequency (Tf-idf) as was discussed in the video. Tf-idf has the effect of reducing the value of common words, while increasing the weight of words that do not occur in many documents.

## Instructions

100 XP

* Import TfidfVectorizer from sklearn.feature\_extraction.text.
* Instantiate TfidfVectorizer while limiting the number of features to 100 and removing English stop words.
* Fit and apply the vectorizer on text\_clean column in one step.
* Create a DataFrame tv\_df containing the weights of the words and the feature names as the column names.

**# Import TfidfVectorizer**

**from sklearn.feature\_extraction.text import TfidfVectorizer**

**# Instantiate TfidfVectorizer**

**tv = TfidfVectorizer(max\_features=100,stop\_words='english')**

**# Fit the vectroizer and transform the data**

**tv\_transformed = tv.fit\_transform(speech\_df['text\_clean'])**

**# Create a DataFrame with these features**

**tv\_df = pd.DataFrame(tv\_transformed.toarray(),**

**columns=tv.get\_feature\_names()).add\_prefix('TFIDF\_')**

**print(tv\_df.head())**

<script.py> output: TFIDF\_action TFIDF\_administration TFIDF\_america TFIDF\_american TFIDF\_americans ... TFIDF\_war TFIDF\_way TFIDF\_work TFIDF\_world TFIDF\_years 0 0.000 0.133 0.000 0.105 0.0 ... 0.000 0.061 0.000 0.046 0.053 1 0.000 0.261 0.266 0.000 0.0 ... 0.000 0.000 0.000 0.000 0.000 2 0.000 0.092 0.157 0.073 0.0 ... 0.024 0.000 0.000 0.064 0.073 3 0.000 0.093 0.000 0.000 0.0 ... 0.037 0.000 0.039 0.096 0.000 4 0.041 0.040 0.000 0.031 0.0 ... 0.094 0.000 0.000 0.055 0.063 [5 rows x 100 columns]

**A job well done! Did you notice that counting the word occurences and calculating the Tf-idf weights are very similar? This is one of the reasons scikit-learn is very popular, a consistent API.**

# Inspecting Tf-idf values

After creating Tf-idf features you will often want to understand what are the most highest scored words for each corpus. This can be achieved by isolating the row you want to examine and then sorting the the scores from high to low.

The DataFrame from the last exercise (tv\_df) is available in your workspace.

## Instructions

100 XP

* Assign the first row of tv\_df to sample\_row.
* sample\_row is now a series of weights assigned to words. Sort these values to print the top 5 highest-rated words.
* **# Isolate the row to be examined**
* **sample\_row = tv\_df.iloc[0]**
* **# Print the top 5 words of the sorted output**
* **print(sample\_row.sort\_values(ascending=False).head(5))**
* **TFIDF\_government 0.367**
* **TFIDF\_public 0.333**
* **TFIDF\_present 0.315**
* **TFIDF\_duty 0.239**
* **TFIDF\_country 0.230**
* **Name: 0, dtype: float64**
* Do you think these scores make sense for the corresponding words?

# Transforming unseen data

When creating vectors from text, any transformations that you perform before training a machine learning model, you also need to apply on the new unseen (test) data. To achieve this follow the same approach from the last chapter: fit the vectorizer only on the training data, and apply it to the test data.

For this exercise the speech\_df DataFrame has been split in two:

* train\_speech\_df: The training set consisting of the first 45 speeches.
* test\_speech\_df: The test set consisting of the remaining speeches.

## Instructions

100 XP

* Instantiate TfidfVectorizer.
* Fit the vectorizer and apply it to the text\_clean column.
* Apply the same vectorizer on the text\_clean column of the test data.
* Create a DataFrame of these new features from the test set.
* **# Instantiate TfidfVectorizer**
* **tv = TfidfVectorizer(max\_features=100, stop\_words='english')**
* **# Fit the vectroizer and transform the data**
* **tv\_transformed = tv.fit\_transform(train\_speech\_df['text\_clean'])**
* **# Transform test data**
* **test\_tv\_transformed = tv.transform(test\_speech\_df['text\_clean'])**
* **# Create new features for the test set**
* **test\_tv\_df = pd.DataFrame(test\_tv\_transformed.toarray(),**
* **columns=tv.get\_feature\_names()).add\_prefix('TFIDF\_')**
* **print(test\_tv\_df.head())**
* **TFIDF\_action TFIDF\_administration TFIDF\_america TFIDF\_american TFIDF\_authority ... TFIDF\_war TFIDF\_way TFIDF\_work TFIDF\_world TFIDF\_years**
* **0 0.000 0.030 0.234 0.083 0.000 ... 0.079 0.033 0.000 0.300 0.135**
* **1 0.000 0.000 0.547 0.037 0.000 ... 0.053 0.067 0.079 0.278 0.126**
* **2 0.000 0.000 0.127 0.135 0.000 ... 0.043 0.054 0.096 0.225 0.044**
* **3 0.037 0.067 0.267 0.031 0.040 ... 0.030 0.038 0.236 0.237 0.062**
* **4 0.000 0.000 0.222 0.157 0.028 ... 0.021 0.081 0.120 0.300 0.153**
* **[5 rows x 100 columns]**

Correct, the vectorizer should only be fit on the train set, never on your test set.

**1. Bag of words and N-grams**

00:00 - 00:30

So far you have looked at individual words on their own without any context or word order, this approach is called a bag-of-words model, as the words are treated as if they are being drawn from a bag with no concept of order or grammar. While analyzing the occurrences of individual words can be a valuable way to create features from a piece of text, you will notice that individual words can loose all their context/meaning when viewed independently.

**2. Issues with bag of words**

00:30 - 01:11

Take for example the word 'happy' shown here. One would assume it was used in a positive context, but if in reality it was used in the phrase 'not happy' this assumption would be incorrect. Similarly if the phrase was extended to 'never not happy' the connotation changes again. One common method to retain at least some concept of word order in a text is to instead use multiple consecutive words like pairs (bi-gram) or three consecutive words (tri-grams). This maintains at least some ordering information while at the same time allowing for the creation of a reasonable set of features.

**3. Using N-grams**

01:11 - 01:41

To leverage n-grams in your own models an additional argument "ngram\_range", can be specified when instantiating your TF-IDF vectorizer. The values assigned to the argument are the minimum and maximum length of n-grams to be included. In this case you would only be looking at bi-grams (n-grams with two words) Printing the bi-gram features created we can see the pairs of words instead of single words.

**4. Finding common words**

01:41 - 02:06

As mentioned in the last video, when creating new features, you should always take time to check your work, and ensure that the features you are creating make sense. A good way to check your n-grams is to see what are the most common values being recorded. This can be done by summing the values of your DataFrame of count values that you created using the sum() method.

**5. Finding common words**

02:06 - 02:25

After sorting the values in descending order you can see the most commonly occurring values. It comes as no surprise that the most commonly occurring bi-gram in a dataset of US president's speeches is "United States" which indicates that the features being created make sense.

**6. Let's practice!**

02:25 - 02:43

You should now be able to try out many different combinations of text based features. It can be interesting to go further and explore the most common longer n-grams such as three word sequences called tri-grams.

# Using longer n-grams

So far you have created features based on individual words in each of the texts. This can be quite powerful when used in a machine learning model but you may be concerned that by looking at words individually a lot of the context is being ignored. To deal with this when creating models you can use n-grams which are sequence of n words grouped together. For example:

* bigrams: Sequences of two consecutive words
* trigrams: Sequences of two consecutive words

These can be automatically created in your dataset by specifying the ngram\_range argument as a tuple (n1, n2) where all n-grams in the n1 to n2 range are included.

## Instructions

100 XP

* Import CountVectorizer from sklearn.feature\_extraction.text.
* Instantiate CountVectorizer while considering only trigrams.
* Fit the vectorizer and apply it to the text\_clean column in one step.
* Print the feature names generated by the vectorizer.
* **# Import CountVectorizer**
* **from sklearn.feature\_extraction.text import CountVectorizer**
* **# Instantiate a trigram vectorizer**
* **cv\_trigram\_vec = CountVectorizer(max\_features=100,**
* **stop\_words='english',**
* **ngram\_range=(3,3))**
* **# Fit and apply trigram vectorizer**
* **cv\_trigram = cv\_trigram\_vec.fit(speech\_df['text\_clean'])**
* **# Print the trigram features**
* **print(cv\_trigram\_vec.get\_feature\_names())**
* **['ability preserve protect', 'agriculture commerce manufactures', 'america ideal freedom', 'amity mutual concession', 'anchor peace home', 'ask bow heads', 'best ability preserve', 'best interests country', 'bless god bless', 'bless united states', 'chief justice mr', 'children children children', 'citizens united states', 'civil religious liberty', 'civil service reform', 'commerce united states', 'confidence fellow citizens', 'congress extraordinary session', 'constitution does expressly', 'constitution united states', 'coordinate branches government', 'day task people', 'defend constitution united', 'distinction powers granted', 'distinguished guests fellow',**

Here you can see that by taking sequential word pairings, some context is preserved.

# Finding the most common words

Its always advisable once you have created your features to inspect them to ensure that they are as you would expect. This will allow you to catch errors early, and perhaps influence what further feature engineering you will need to do.

The vectorizer (cv) you fit in the last exercise and the sparse array consisting of word counts (cv\_trigram) is available in your workspace.

## Instructions

100 XP

* Create a DataFrame of the features (word counts).
* Add the counts of word occurrences and print the top 5 most occurring words.
* **# Create a DataFrame of the features**
* **cv\_tri\_df = pd.DataFrame(cv\_trigram.toarray(),**
* **columns=cv\_trigram\_vec.get\_feature\_names()).add\_prefix('Counts\_')**
* **# Print the top 5 words in the sorted output**
* **print(cv\_tri\_df.sum().sort\_values(ascending=False).head())**
* **Counts\_constitution united states 20**
* **Counts\_people united states 13**
* **Counts\_mr chief justice 10**
* **Counts\_preserve protect defend 10**
* **Counts\_president united states 8**
* **dtype: int64**
* **Great, that the most common trigram is constitution united states makes a lot of sense for US presidents speeches.**
* **1. Wrap-up**
* 00:00 - 00:14
* Congratulations on completing the course “Feature Engineering for Machine Learning in Python”. This course set out to teach you about understanding your data types and how best to prepare your dataset for a machine learning model. Let's take a moment to recap what you have covered.
* **2. Chapter 1**
* 00:14 - 00:28
* In chapter one, you learned how to better understand the underlying types of data contained in your dataset, how to create features out of categorical columns and how to bin continuous columns.
* **3. Chapter 2**
* 00:28 - 00:39
* In chapter two, we moved on to exploring how to deal with some of the challenges of real world data, such as missing values and non desirable characters in your data.
* **4. Chapter 3**
* 00:39 - 00:52
* Chapter 3 discussed how different distributions can effect your models and how to mitigate it, and different ways to deal with spurious outlier values in your dataset.
* **5. Chapter 4**
* 00:52 - 01:02
* Finally in chapter 4, we explored how to deal with non tabular data such as free text and different ways to encode it for use with a machine learning model.
* **6. Next steps**
* 01:02 - 01:27
* Hopefully these newly learned skills should benefit both your personal projects and your professional careers. A great place to test out these skills is to try applying them to kaggle competitions or any of your own pet projects to see if they improve your results. Or, if you want to explore these topics further, perhaps you could try out some of the other related courses on DataCamp.
* **7. Thank You!**
* 01:27 - 01:41
* This is the final video, and would like to thank you for going through this course. I hope you have learned from it and it provides value in your machine learning work ahead.