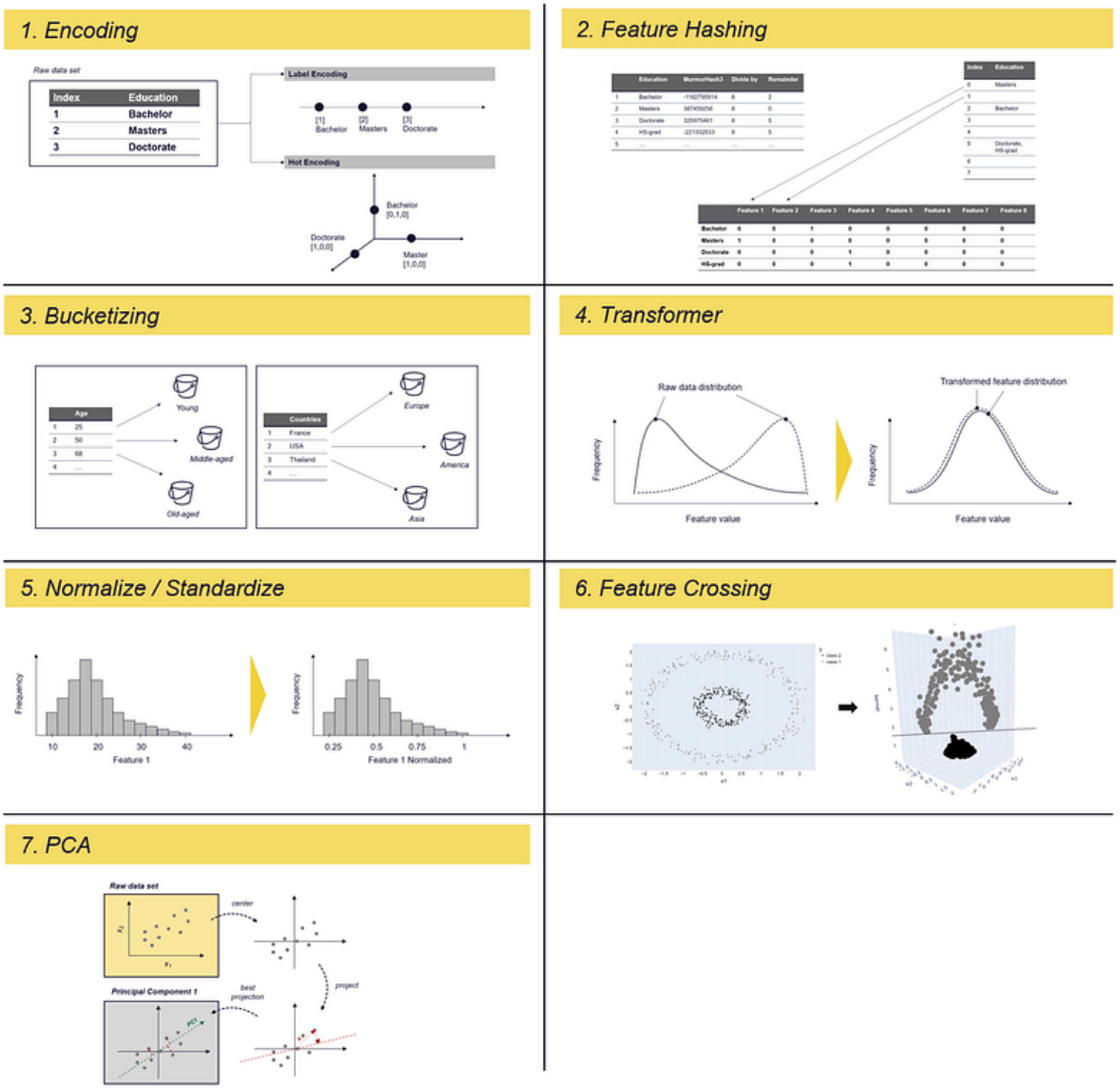
7 of the Most Used Feature Engineering Techniques

Hands-on Feature Engineering with Scikit-Learn, Tensorflow, Pandas and Scipy



7 of the most used Feature Engineering Techniques—Image by the author

Table of content

Introduction

Feature engineering describes the process of formulating relevant features that describe the underlying data science problem as accurately as possible and make it possible for algorithms to understand and learn patterns. In other words:

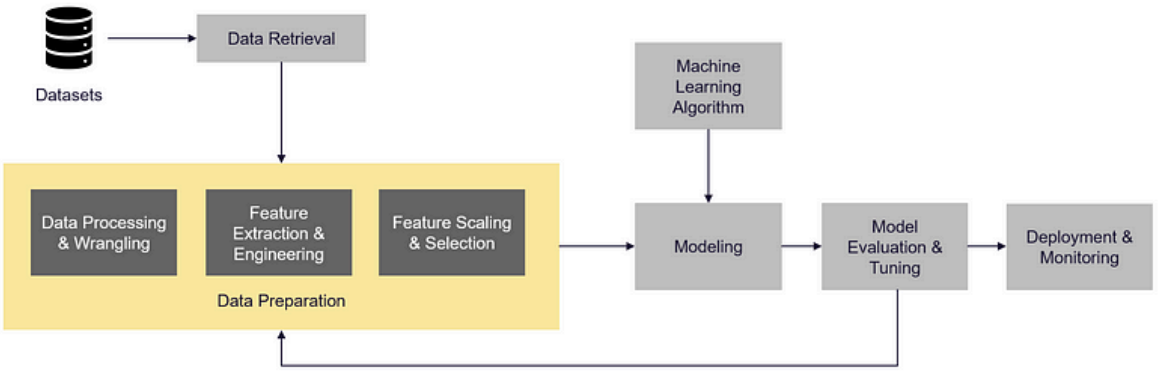
> Features you provide serve as a way to communicate your own understanding and knowledge about the world to your model

Each feature describes a kind of information “piece”. The sum of these pieces allows the algorithm to draw conclusions about the target variables—at least if you have a data set that actually contains information about your target variable.

According to the Forbes magazine, Data Scientists spend about 80% of their time collecting and preparing relevant data, with the data cleaning and data organizing alone taking up about 60% of the time.

> But this time is well spent.

I believe that the quality of the data, as well as the proper preparation of the data set features, have a greater impact on the success of a machine learning model than any other part of the ML pipeline:



*A standard Machine Learning pipeline—Inspired by [*Sarkar et al., 2018*]*

What Forbes magazine considers as “cleaning and organizing” is usually broken down into two to three subcategories in the ML pipeline (I have highlighted them with a yellow background in the image above):

(1) Data (Pre-)Processing: The initial preparation of the data—for example, smoothing a signal, dealing with outliers, etc.

(2) **Feature engineering:** Defining input features for our model—e.g., by converting an (acoustic) signal to the frequency domain using the fast Fourier transform (FFT) allows us to extract crucial information from the raw signal.

(3) **Feature Selection:** Selection of features that have a significant impact on the target variable. By selecting the important features and thus reducing the dimensionality, we can significantly reduce the modeling costs and increase the robustness and performance of the model.

> Why do we need Feature Engineering?

Andrew Ng frequently advocates a so-called ***data-centric approach***, which emphasizes the importance of selecting and curating data, rather than simply trying to collect more and more data. The goal is to ensure that the data is of high quality and relevance to the problem being addressed and to continually improve the data set through data cleaning, feature engineering and data augmentation.

> Why do we need feature engineering and a data-centric approach when we have Deep Learning?

This approach is particularly useful for use cases where it is costly or otherwise difficult to collect large amounts of data. [Brown, 2022] This might be the case when the data is difficult to access, or when there are strict regulations or other barriers to collecting and storing large amounts of data, e.g.:

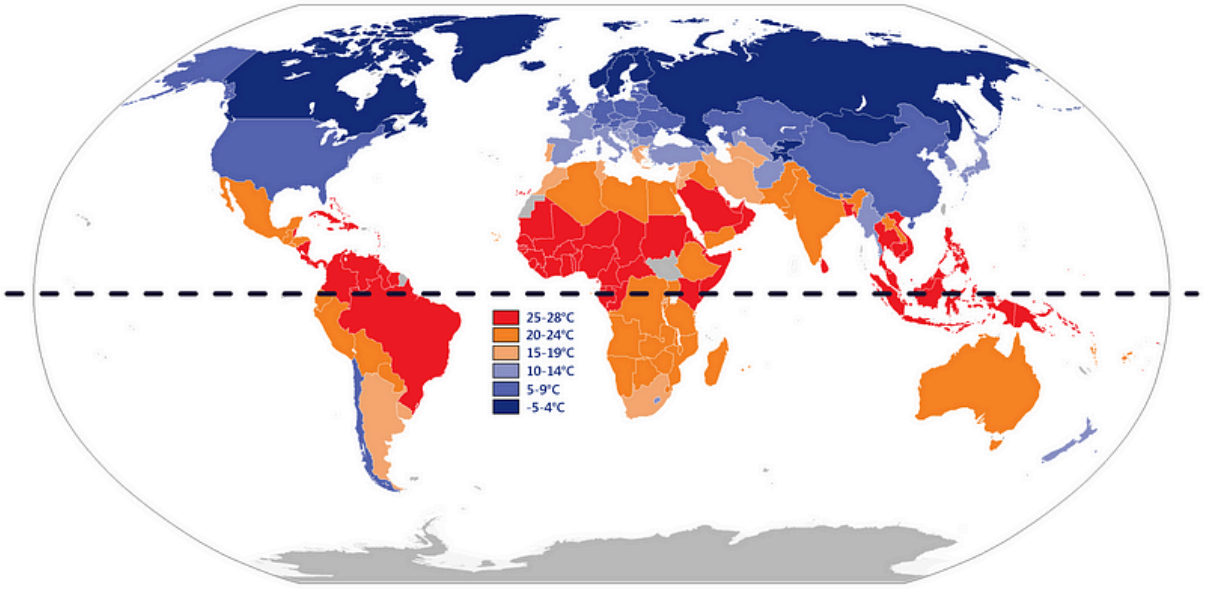
- ***In the manufacturing sector***, equipping production facilities with comprehensive sensor technology and connecting them to databases is expensive. As a result, many plants do not yet collect data at all. Even when machines are able to collect and store data, they are rarely connected to a central data lake.

- I***n the supply chain context***, every company has only a certain number of orders that it processes every day. So if we want to predict the demand for a product that is in very irregular demand, we sometimes have only a few data points available.

In such areas, feature engineering probably has the greatest leverage to increase the performance of the models. Here, the creativity of the engineers and data scientists is required to enhance the data set quality to a sufficient level. The process is rarely straightforward, but rather experimental and iterative.

When humans analyze data, they often use their past knowledge and experience to help them understand patterns and make predictions. For example, if someone is trying to estimate the temperature in different countries, they might consider the location of the country relative to the equator, since they know that temperatures tend to be higher near the equator.

However, a machine learning model does not have the same inherent understanding of concepts and relationships as a human does. It can only learn from the data it is given. Therefore, any background information or context that a human would use to solve a problem must be explicitly included in the data set in a numerical form.



Average yearly temperature per country [Wikimedia]

> So what data do we need to make the model smarter than a human?

We could use the Google Maps API to find the location coordinates (longitude and latitude) of each country. Additionally, we can gather information about the altitude of the region and the distance from the nearest body of water for each country. By collecting this extra data, we hope to identify and consider possible factors that could affect the temperature in each country.

> Let's say we have collected some data that might have an impact on the temperature, what next?

Once we have enough data that describes the characteristics of the problem, we still have to make sure that the computer can understand the data. Categorical data, dates, etc. must be converted into numerical values.

In this article, I am describing a few commonly used techniques for preparing raw data. Some techniques are used to convert categorical data into numerical values that can be understood by the machine learning model, such as **encoding** and **vectorizing**. Other techniques are used to address the distribution of the data, such as **transformers** and **binning**, which can help to normalize or standardize the data in some way. Still, other techniques are used to reduce the dimensionality of the dataset by generating new features, such as **hashing** and **principal component analysis (PCA).**

> Try it yourself…

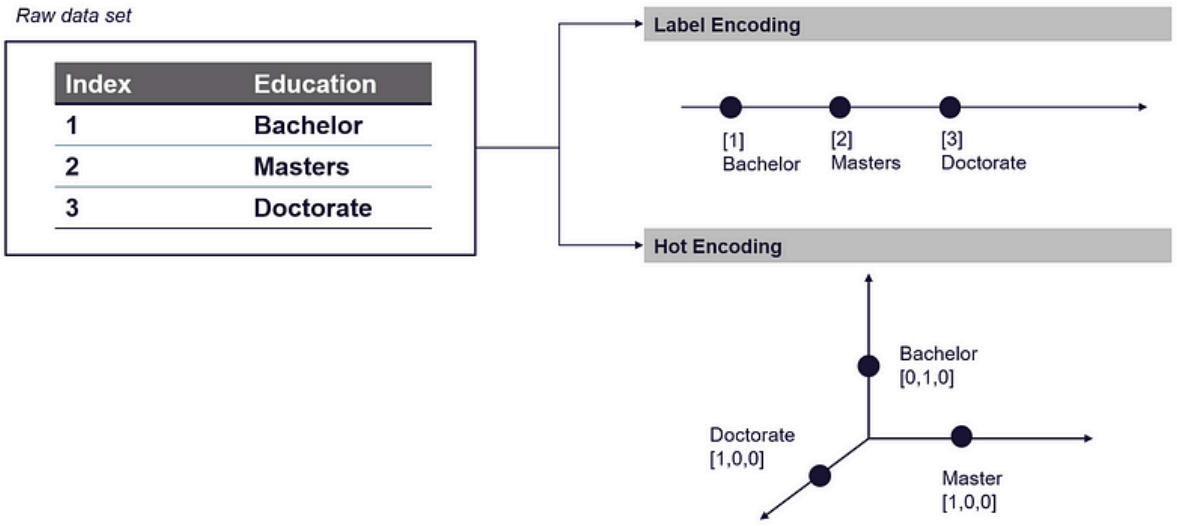
If you want to follow the article and try the methods yourself, you can use the repo below, which contains the code snippets in a Jupyter Notebook and the data sets used:

GitHub - polzerdo55862/7-feature-engineering-techniques
*You can't perform that action at this time. You signed in with another tab or window. You signed out in another tab or…*github.com

---

1. Encoding

Feature encoding is a process used to transform categorical data into numerical values that can be understood by ML algorithms. There are several types of encoding, including label encoding and one-hot encoding.



Label and Hot Encoding—Image by the author

Label encoding involves assigning a numeric value to each categorical value. This can be effective if there is an inherent order to the categorical values, such as grades from A to F, which can be encoded as numeric values from 1 to 5 (or 6). However, if there is no inherent order to the categorical values, label encoding may not be the best approach.

Alternatively, you can use **One-hot encoding** to transform categorical values into numerical values. In one-hot encoding, the column of categorical values is split into several new columns, one for each unique categorical value.

For example, if the categorical values are grades from A to F:

- we would have five new columns, one for each grade.
- Each row in the dataset would have a value of 1 in the column corresponding to its grade and 0 in all the other columns.
- This results in a so-called sparse matrix, where most of the values are 0.

The disadvantage of one-hot encoding is that it can significantly increase the size of the dataset, which can be a problem if the column you want to encode contains hundreds or thousands of unique categorical values.

In the following I am using Pandas, Scikit-learn or Tensorflow to apply label and one-hot encoding to your data set. For the example below, we are using the ***Census Income data set:***

> **Data set: Census Income** [License: CC0: Public Domain]
> https://archive.ics.uci.edu/ml/datasets/census+income
> https://www.kaggle.com/datasets/uciml/adult-census-income
> The Census Income Data set describes the income of individuals in the United States. It includes their age, sex, marital status and other demographic information as well as their annual income, which is divided into two categories: over $50,000 or under $50,000.

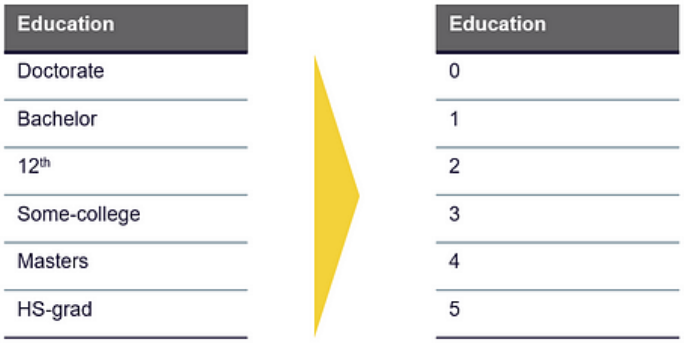
For the labelling example, we are using the “education” column of the census data set, which describes the highest level of education achieved by individuals within the population. It contains information such as whether an individual has completed high school, completed college, earned a graduate degree, or some other form of education.

First, let’s load the data set:

```
import pandas as pd

# load dataset - census income
census_income = pd.read_csv(r'../input/income/train.csv')
```

1.1 Label Encoding using Scikit-learn

Label encoding is the simplest way to convert categorical values into numerical values. It is a simple process of assigning a numerical value to each category.



Label Encoding— Image by the author

You can find suitable libraries in Pandas, Scikit-Learn and Tensorflow. I am using the Scikit-Learns Label Encoder function. It is randomly assigning integers to the unique categorical values, which is the simplest way of encoding:

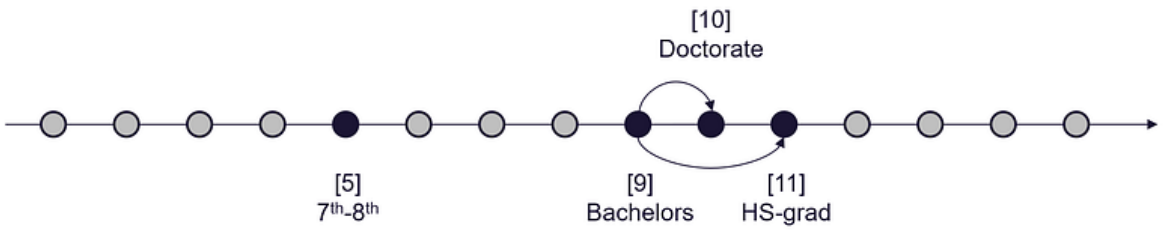
```
from sklearn import preprocessing

# define and fit LabelEncoder
le = preprocessing.LabelEncoder()
le.fit(census_income["education"])

# Use the trained LabelEncoder to label the education column
census_income["education_labeled"] = le.transform(census_income["education"])

display(census_income[["education", "education_labeled"]])
```

	education	education_labeled
0	Doctorate	10
1	12th	2
2	Bachelors	9
3	7th-8th	5
4	Some-college	15
...	...	...
43952	Bachelors	9
43953	HS-grad	11
43954	Some-college	15
43955	Bachelors	9
43956	HS-grad	11

43957 rows × 2 columns

This way of encoding can cause problems for some algorithms because the assigned integers do not necessarily reflect any inherent order or relationship between the categories. For example, in the case described above, the algorithm may assume that the categories

Doctorate (10) and **HS-grad** (11) are more similar to each other than categories **Doctorate** (10) and **Bachelor** (9) and that **HS-grad** (11) is “higher” than **Doctorate** (10).



Label Encoding result interpretation—Image by the author

If we have some knowledge about the specific domain or subject matter of a data set, we can use it to ensure that the label encoding process reflects any inherent order. For our example, we could try to label the education levels considering the order in which different degrees are obtained, e.g.:

Doctorate is a higher academic degree compared to a Master's and Bachelor's. The Master's is higher than the Bachelor's.

- HS-grad → 1
- Bachelors → 2
- Masters → 3
- Doctorate → 4

To apply this manual mapping to the data set, we can use the *pandas.map* function:

```
education_labels = {'Doctorate':5, 'Masters':4, 'Bachelors':3, 'HS-grad':2, '12th':1, '11th':0}

census_income['education_labeled_pandas']=census_income['education'].map(education_labels)

census_income[["education", "education_labeled_pandas"]]
```

	education	education_labeled_pandas
0	Doctorate	5.0
1	12th	1.0
2	Bachelors	3.0
3	7th-8th	NaN
4	Some-college	NaN
...	...	...

But how does this affect the model-building process?

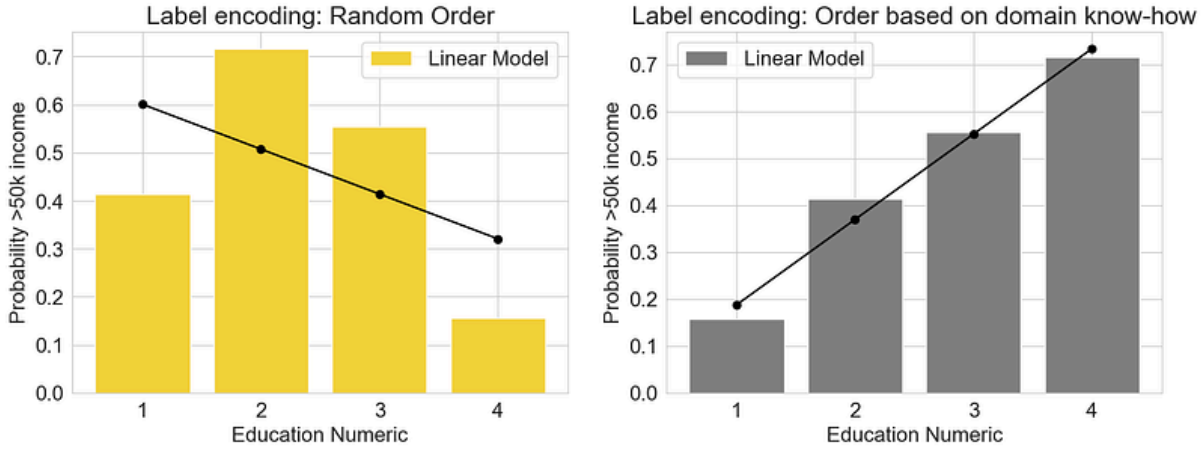
Let's build a simple linear regression model with the labeled encoded data:

For the figure below, the target variable is defined as the **probability that an individual has an income of more than $50,000.**

- **In the left figure**, the categorical values (such as “Bachelor” and “Doctorate”) have been assigned randomly to numerical values.
- **In the right figure**, the numerical values assigned to the categorical values reflect the order in which the degrees are typically obtained, with higher education degrees being assigned higher numerical values.

→ **The right figure** shows a clear correlation between education level and income, which could be represented by a simple linear model.

→ **In contrast, the left figure** shows a relationship between the education attribute and the target variable, which would require a more complex model to accurately represent. This could affect the interpretability of the results, increase the risk of overfitting and increase the computational effort required to fit the model.



Label Encoding: Linear Regression Model trained on ordered and unordered “education” feature—Image by the author

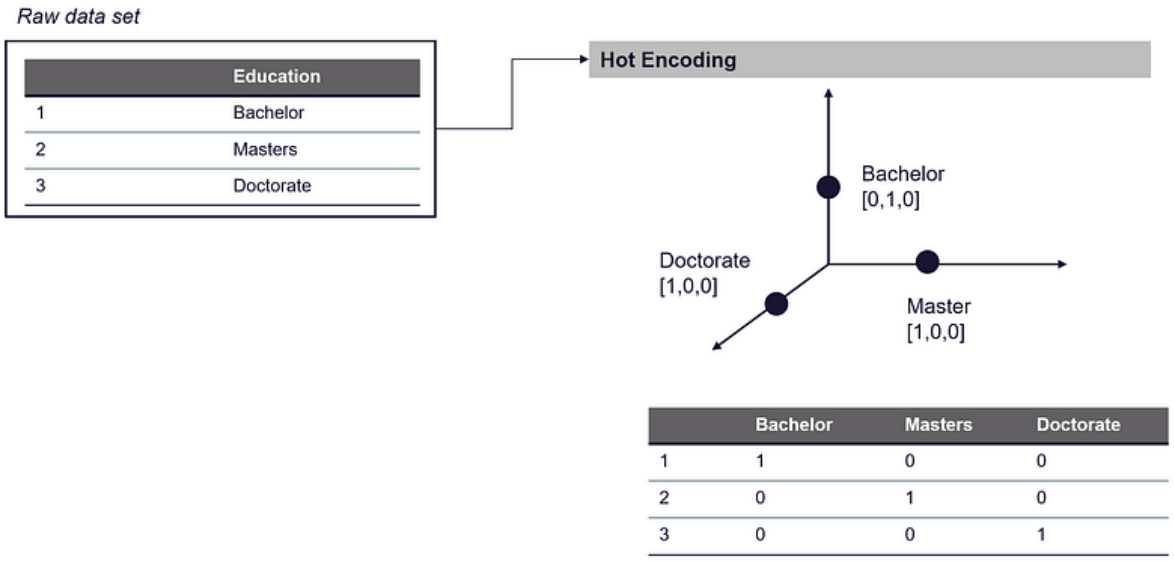
> What do we do when values have no inherent order or we don’t have enough information to map it?

If the categorical values do not have an inherent order at all, it may be better to use a method of encoding that converts the categories into a set of numeric variables ***without introducing any bias***. One-hot encoding is one suitable method.

1.2 One-Hot Encoding using Scikit-learn, Pandas and Tensorflow

One-hot encoding is a technique for converting categorical data into numerical data. It does this by creating a new binary column for each unique category in the data set and assigning a value of 1 to rows that belong to that category and a value of 0 to rows that do not.

→ This process helps to avoid introducing bias into the data by not assuming any inherent order between the categories.



One-Hot Encoding—Image by the author

The process of One-Hot Encoding is pretty straightforward. We could simply implement it by ourselves or use one of the existing functions. Scikit-learn has the **.preprocessing.OneHotEncoder()** function**, Tensorflow the .one_hot()** function, and Pandas the **.get_dummies()** function.

- **Pandas.get_dummies()**

```
import pandas as pd

education_one_hot_pandas = pd.get_dummies(census_income["education"],
prefix='education')
education_one_hot_pandas.head(2)
```

- **Sklearn.preprocessing.LabelBinarizer()**

```
from sklearn import preprocessing

lb = preprocessing.LabelBinarizer()
lb.fit(census_income["education"])

education_one_hot_sklearn_binar =
pd.DataFrame(lb.transform(census_income["education"]), columns=lb.classes_)
education_one_hot_sklearn_binar.head(2)
```

- **Sklearn.preprocessing.OneHotEncoder()**

```
from sklearn.preprocessing import OneHotEncoder

# define and fit the OneHotEncoder
ohe = OneHotEncoder()
ohe.fit(census_income[['education']])
```

```
# transform the data
education_one_hot_sklearn =
pd.DataFrame(ohe.transform(census_income[["education"]]).toarray(),
columns=ohe.categories_[0])
education_one_hot_sklearn.head(3)
```

The problem with one-hot encoding is that it can lead to large and sparse datasets with high dimensionality.

Using one-hot encoding to convert a categorical feature with 10,000 unique values into numerical data would result in the creation of 10,000 new columns in the data set, each representing a different category. This can be a problem when working with large data sets, as it can quickly consume a lot of memory and computational resources.

If memory and computer power are limited, it may be ***necessary to reduce the number of features*** *in the data set to avoid running into memory or performance issues.*

How can we reduce dimensionality to save memory?

When doing this, it is important to try to minimize the loss of information as much as possible. This can be achieved by carefully selecting which features to retain or remove, by using techniques such as ***feature selection or dimensionality reduction*** to identify and remove redundant or irrelevant features. [Sklearn.org][Wikipedia, 2022]

In this article I am describing two possible ways how to reduce the dimensionality of your data set:

1. **Feature Hashing**—see section 2. Feature Hashing
2. **Principal Component Analysis (PCA)**—see section 7. PCA

Loss of information vs. speed vs. memory

There is probably not one “perfect” solution for reducing the number of dimensions in your data set. One method may be faster but may result in the loss of a lot of information, while the other method preserves more information but requires a lot of computing resources (which may also lead to memory issues).

---

2. Feature Hashing

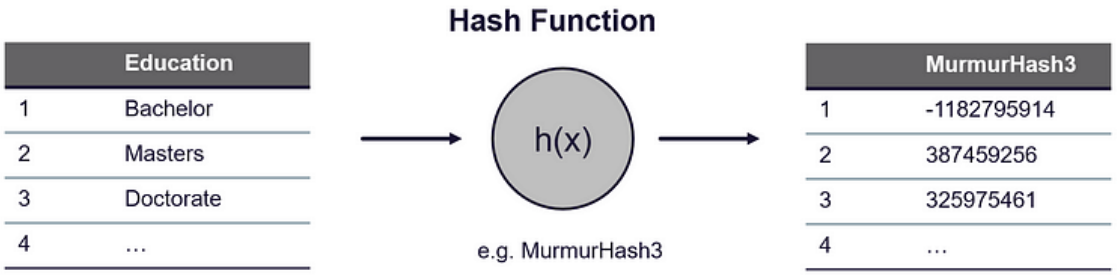
Feature hashing is primarily a dimensionality reduction technique and is often used in Natural Language Processing. However, **hashing can also be useful when we want to vectorize categorical features with several hundred and thousand unique categories**. With hashing, we can limit the increase of dimensionality by assigning several unique values to the same hash value.

→ ***Hashing is thus a low-memory alternative to OneHotEncoding and other feature vectorizing methods.***

Hashing works by applying a hash function to the features and using the hash values directly as indices, rather than building a hash table and looking up indices in it individually. The implementation in Sklearn is based on Weinberger [Weinberger et al., 2009].

We can break it down into 2 steps:

- **Step 1:** the categorical values are first converted to a hash value using a hash function. The implementation in Scikit-learn uses the 32-bit variant of MurmurHash3 for this.



Hashing Trick—Image by the author

```
import sklearn
import pandas as pd

# load data set
census_income = pd.read_csv(r'../input/income/train.csv')
education_feature =
census_income.groupby(by=["education"]).count().reset_index()["education
"].to_frame()

#############################################################################
# Apply the hash function, here MurmurHash3
#############################################################################
def hash_function(row):
   return(sklearn.utils.murmurhash3_32(row.education))

education_feature["education_hash"] =
education_feature.apply(hash_function, axis=1)
education_feature
```

	education	education_hash
0	10th	611946777
1	11th	1907886797
2	12th	960033297
3	1st-4th	-918906142
4	5th-6th	211780192
5	7th-8th	-1858991020
6	9th	1219543683
7	Assoc-acdm	-1621657589
8	Assoc-voc	334383773
9	Bachelors	-1182795914
10	Doctorate	325975461
11	HS-grad	-221332533
12	Masters	387459256
13	Preschool	556701269
14	Prof-school	-1597813725
15	Some-college	1210481717

- **Step 2:** Next, we reduce the dimensionality by applying a ***mod function*** to the feature values. Using the mod function, we calculate the remainder after dividing the hash value by ***n_features*** (the number of features of the output vector).

*It is advisable to use a power of two as the **n_features** parameter; otherwise, the features will not map evenly to the columns.*

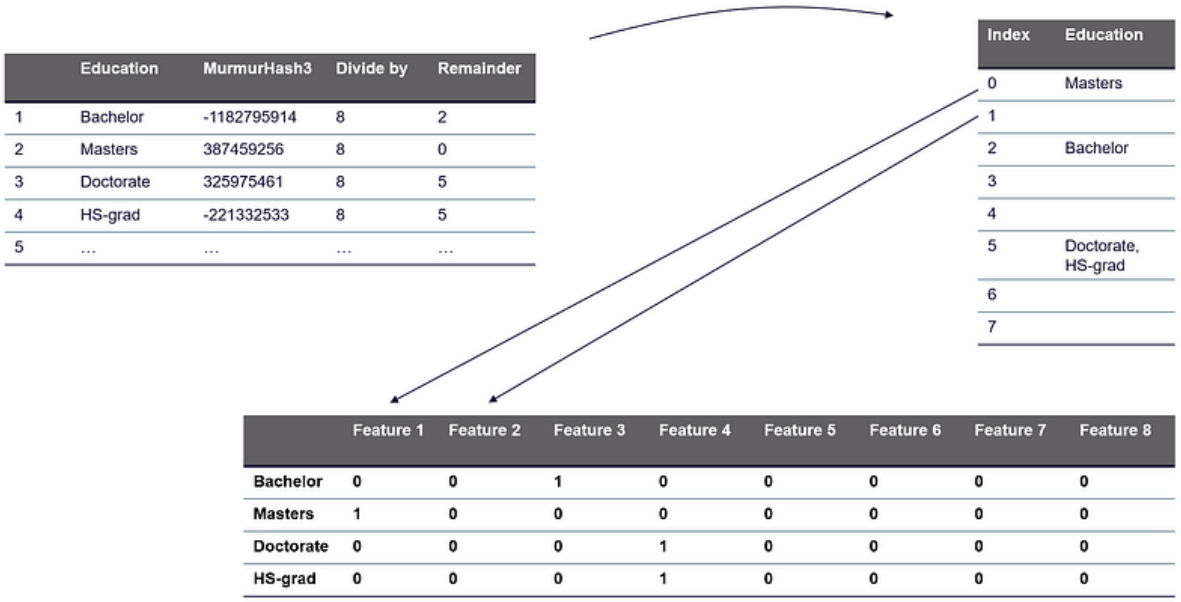
Example: Census Income data set →Encoding the Education column

The “Education” column of the census income dataset contains 16 unique values:

- With one-hot encoding, 16 columns will be added to the dataset, each representing one of the 16 unique values.

→ To reduce the dimensions, we set ***n_features*** to 8.

This inevitably leads to “collisions”, i.e. that different categorical values are mapped to the same hash columns. So in other words, we are assigning different values to the same bucket. Similar to what we are doing in ***Binning/Bucketizing*** (see section 3. Binning/Bucketizing). In feature hashing, we deal with collisions by simply chaining values that are assigned to the same bucket and storing them in a list (known as “separate chaining”).



Feature Hashing: Reducing the dimensionality by calculating the remainder as a new column index—Image by the author

```
##############################################################################
# Apply mod function
##############################################################################
n_features = 8

def mod_function(row):
   return(abs(row.education_hash) % n_features)

education_feature["education_hash_mod"] =
education_feature.apply(mod_function, axis=1)
education_feature.head(5)
```

	education	education_hash	education_hash_mod
0	10th	611946777	1
1	11th	1907886797	5
2	12th	960033297	1
3	1st-4th	-918906142	6
4	5th-6th	211780192	0
5	7th-8th	-1858991020	4
6	9th	1219543683	3
7	Assoc-acdm	-1621657589	5
8	Assoc-voc	334383773	5
9	Bachelors	-1182795914	2
10	Doctorate	325975461	5
11	HS-grad	-221332533	5
12	Masters	387459256	0
13	Preschool	556701269	5
14	Prof-school	-1597813725	5
15	Some-college	1210481717	5

A function that is doing the just described steps for us is the ***HashingVectorizer*** function from ***Scikit-learn***.

2.1 Feature Hashing using Scikit-learn

sklearn.feature_extraction.text.HashingVectorizer

from sklearn.feature_extraction.text import HashingVectorizer

```
# define Feature Hashing Vectorizer
vectorizer = HashingVectorizer(n_features=8, norm=None,
alternate_sign=False, ngram_range=(1,1), binary=True)

# fit the hashing vectorizer and transform the education column
X = vectorizer.fit_transform(education_feature["education"])

# transformed and raw column to data frame
df = pd.DataFrame(X.toarray()).assign(education =
education_feature["education"])
display(df)
```

	0	1	2	3	4	5	6	7	education
0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	10th
1	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	11th
2	0.0	1.0	0.0	0.0	0.0	0.0	0.0	0.0	12th
3	0.0	0.0	0.0	1.0	0.0	0.0	1.0	0.0	1st-4th
4	0.0	1.0	0.0	0.0	1.0	0.0	0.0	0.0	5th-6th
5	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	7th-8th
6	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	9th
7	0.0	1.0	0.0	0.0	0.0	0.0	1.0	0.0	Assoc-acdm
8	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	Assoc-voc
9	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	Bachelors
10	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	Doctorate
11	0.0	0.0	1.0	0.0	0.0	0.0	1.0	0.0	HS-grad
12	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	Masters
13	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	Preschool
14	0.0	0.0	0.0	1.0	1.0	0.0	0.0	0.0	Prof-school
15	0.0	0.0	0.0	0.0	1.0	0.0	0.0	1.0	Some-college

---

3. Binning / Bucketizing

Binning is used for both categorical and numerical data. As the name suggests, the goal is to map the values of the features to “bins” and replace the original value with the value that represents the bin.

For example, if we had a dataset with values ranging from 0 to 100 and we wanted to group those values into bins of size 10, we might create bins for values 0–9, 10–19, 20–29 and so on.

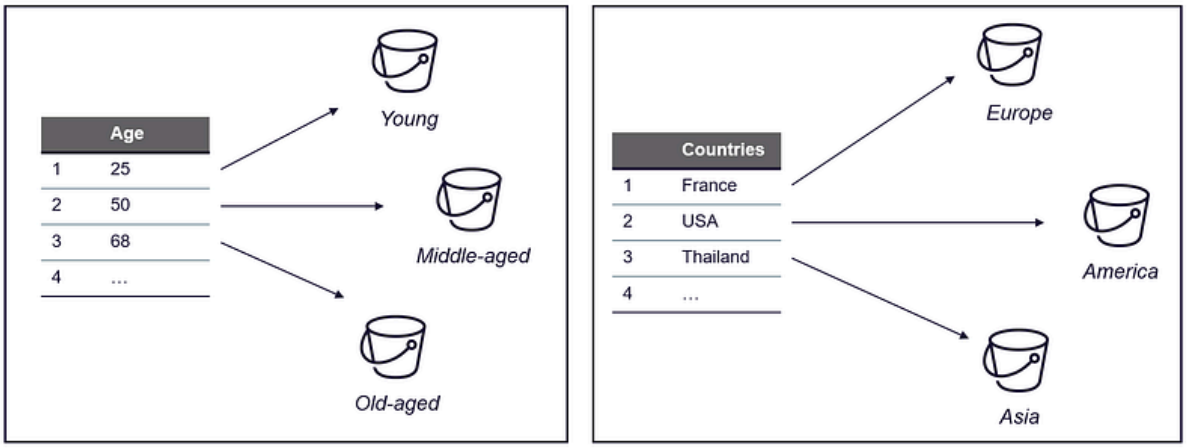
→ In this case, the original values would be replaced with the value that represents the bin to which they belong, such as 10, 20, 30, etc. This can help visualize and analyze the data.

Since we are reducing the number of unique values in the data set, it can help to:

- prevent overfitting
- increase the robustness of the model and mitigate the influence of anomalies
- reduce the model complexity and the required resources to train the model

Systematic binning can help the algorithm to detect underlying patterns more easily and efficiently. It is especially helpful if we can already form a hypothesis before we are defining the bins.

Binning can be used for both numeric and categorical values, e.g.:



Bucketizing for categorical and numerical features—Image by the author

In the following, I describe how this might look for numeric and categorical attributes using three examples:

1. **Numeric—**How binning can be used when building a streaming recommender— a use case I found in the book ***Feature Engineering for Machine Learning*** by [Zheng, Alice, and Amanda Casari. 2018]
2. **Numeric—**Census Income Data set: Binning applied to the “Age” column
3. **Categorical** —Binning in Supply Chain: Assign countries to bins, depending on the target variable

---

Example 1: Streaming Recommender System—How popular is a song or video?
If you want to develop a recommender system, it is important to assign numerical values to the relative popularity of songs. One of the most significant attributes is the number of clicks, which is how often a user has listened to a song.
However, it is not necessarily true that a user who listens to a song 1000 times likes it 20 times as much as someone who has heard it 50 times. Binning can help prevent overfitting. Therefore, it can be beneficial to divide the number of clicks of songs into categories.: [Zheng, Alice and Amanda Casari. 2018]
Click count >=10: Very popular
Click count >=2: popular
Click count <2: neutral, no statement possible

Example 2: Age-Income Relation

For the second example, I am again using the census income data set. The goal is to group individuals into age buckets.

Hypothesis—The idea is that the specific age of a person may not have a significant impact on their income, but rather the stage of life they are in. For example, a person who is 20 and still in school may have a different income compared to a 30-year-old young professional. Similarly, a person who is still working full-time at age 60 may have a different income

compared to someone who has retired at age 70, while there is probably not much difference in income if the person is 40 or 50.

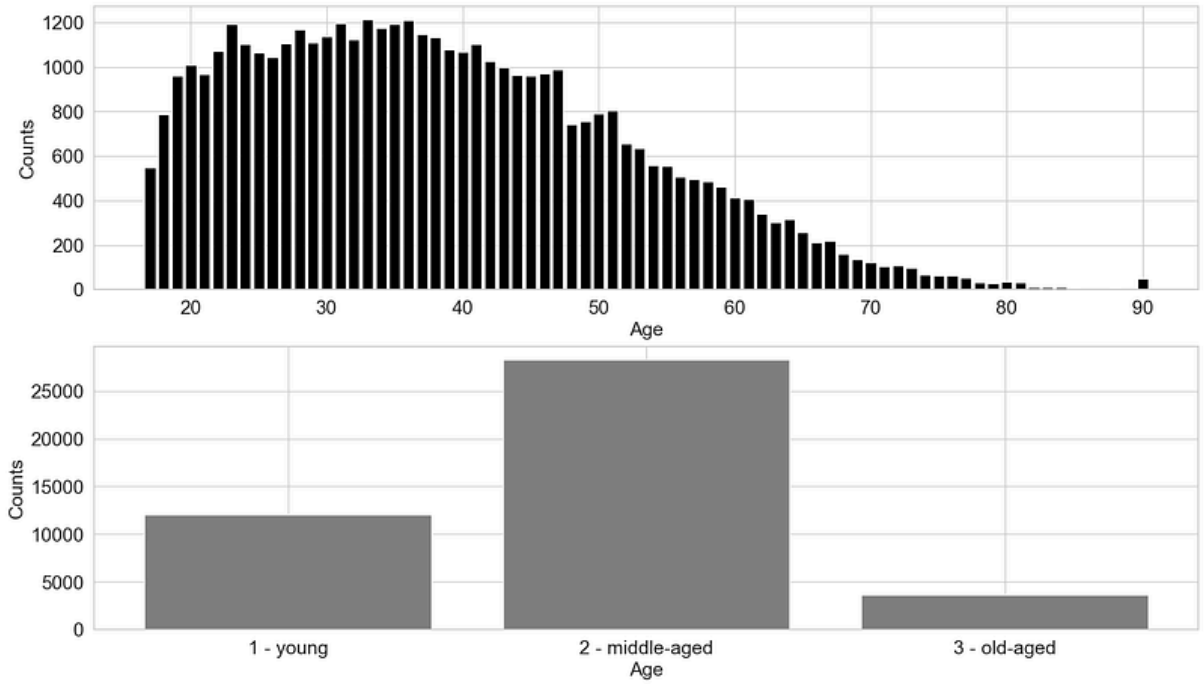
3.1 Bucketizing using Pandas

To bucketize the data by age, I am defining three “buckets”:

- **young**—28 and younger
- **middle-aged**—29 to 59
- **old-aged**—60 and older

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

# creating a dictionary
sns.set_style("whitegrid")
plt.rc('font', size=16) #controls default text size
plt.rc('axes', titlesize=16) #fontsize of the title
plt.rc('axes', labelsize=16) #fontsize of the x and y labels
plt.rc('xtick', labelsize=16) #fontsize of the x tick labels
plt.rc('ytick', labelsize=16) #fontsize of the y tick labels
plt.rc('legend', fontsize=16) #fontsize of the legend

# load dataset - census income
census_income = pd.read_csv(r'../input/income/train.csv')

# define figure
fig, (ax1, ax2) = plt.subplots(2)
fig.set_size_inches(18.5, 10.5)

# plot age histogram
age_count = census_income.groupby(by=["age"])["age"].count()
ax1.bar(age_count.index, age_count, color='black')
ax1.set_ylabel("Counts")
ax1.set_xlabel("Age")

# binning age
def age_bins(age):
    if age < 29:
        return "1 - young"
    if age < 60 and age >= 29:
        return "2 - middle-aged"
    else:
        return "3 - old-aged"

# apply trans. function
```

```
census_income["age_bins"] = census_income["age"].apply(age_bins)

# group and count all entries in the same bin
age_bins_df = census_income.groupby(by=["age_bins"])["age_bins"].count()

ax2.bar(age_bins_df.index, age_bins_df, color='grey')
ax2.set_ylabel("Counts")
ax2.set_xlabel("Age")
```



3.2 Bucketizing using Tensorflow

Tensorflow provides a module called ***feature columns*** that contains a range of functions designed to help with the pre-processing of raw data. Feature Columns are functions that organize and interpret raw data so that a machine learning algorithm can interpret it and use it to learn. [Google Developers, 2017]

TensorFlow's ***feature_column.bucketized_column*** API provides a way to bucketize numeric data. This API takes in a numeric column and creates a bucketized column based on the specified boundaries. The input numeric column can be either a continuous or discrete value.

```
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow import feature_column
from tensorflow.keras import layers
```

```
# load dataset - census income
census_income = pd.read_csv(r'../input/income/train.csv')
data = census_income.to_dict('list')

# A utility method to show the transformation from feature column
def demo(feature_column):
   feature_layer = layers.DenseFeatures(feature_column)
   return feature_layer(data).numpy()

age = feature_column.numeric_column("age")
age_buckets = feature_column.bucketized_column(age, boundaries=[30,50])
buckets = demo(age_buckets)

# add buckets to the data set
buckets_tensorflow = pd.DataFrame(buckets)

# define boundaries for buckets
boundary1 = 30
boundary2 = 50

# define column names for buckets
bucket1=f"age<{boundary1}"
bucket2=f"{boundary1}<age<{boundary2}"
bucket3=f"age>{boundary2}"

buckets_tensorflow_renamed = buckets_tensorflow.rename(columns =
{0:bucket1,

1:bucket2,

2:bucket3})

buckets_tensorflow_renamed.assign(age=census_income["age"]).head(7)
```

	age<30	30<age<50	age>50	age
0	0.0	0.0	1.0	67
1	1.0	0.0	0.0	17
2	0.0	1.0	0.0	31
3	0.0	0.0	1.0	58
4	1.0	0.0	0.0	25
5	0.0	0.0	1.0	59
6	0.0	0.0	1.0	70

We can do the same by using the ***KBinsDiscretizer*** from ***Scikit-learn.***

3.3 Bucketizing using Scikit-learn

```
from sklearn.preprocessing import KBinsDiscretizer

# define bucketizer
est = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='uniform')
est.fit(census_income[["age"]])

# transform data
buckets = est.transform(census_income[["age"]])

# add buckets column to data frame
census_income_bucketized = census_income.assign(buckets=buckets)[["age",
"buckets"]]
Census_income_bucketized
```

...

	age	buckets
0	67	2.0
1	17	0.0
2	31	0.0
3	58	1.0
4	25	0.0
...	...	...
43952	52	1.0
43953	19	0.0
43954	30	0.0
43955	46	1.0
43956	30	0.0

43957 rows × 2 columns

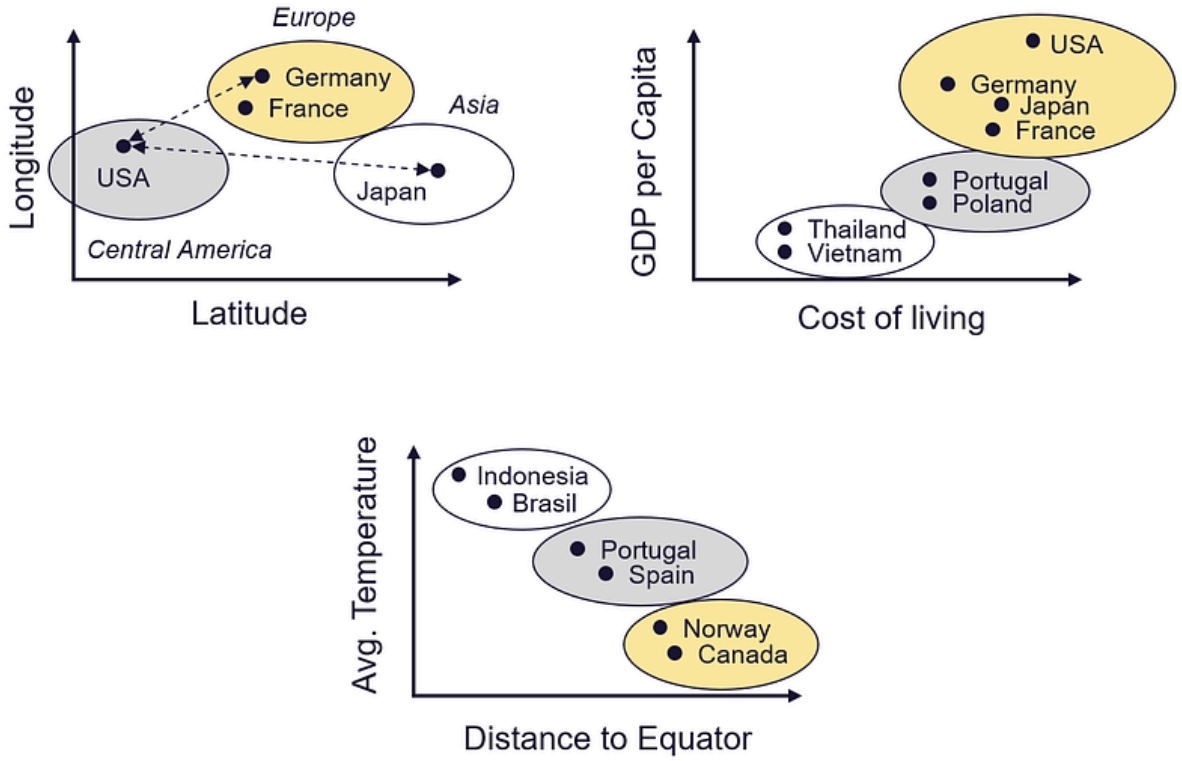
One last example for categorical values …

There is not one universally best way to represent categorical variables numerically. The appropriate method to use, depends on the specific use case and the information that you want to convey to the model. Different methods of encoding can highlight different aspects of the data and it is important to choose the method that best suits the needs of your particular use case.

Example 3: Bucketizing categorical values—e.g. Countries

Supply Chain—When assessing the reliability of a supplier/subcontractor, things like region, climate and type of transport can influence the accuracy of the delivery time. In our resource planning system, we usually store the supplier's country and city. Since some countries are similar in some aspects and not at all in others, it may make sense to highlight aspects that are relevant to the use case.

- If we want to highlight the distance between countries and group countries in the same region, we could define bins for continents
- If we are more interested in pricing or possible revenues, the GDP per Capita is probably more important → Here we could group "high-cost" countries like USA, Germany and Japan in one bin
- If we want to highlight general weather conditions, we could put countries in the north like Canada and Norway in the same bin

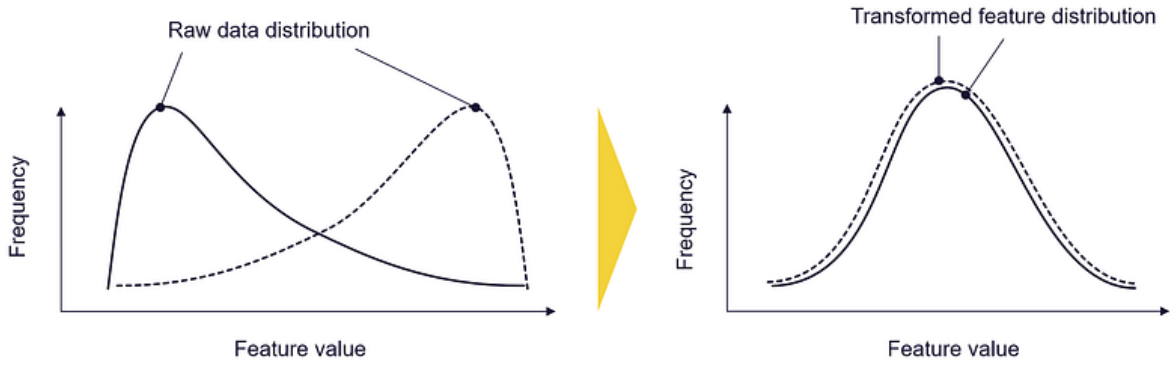


Example: Bucketizing of countries using different attributes—Image by the author

4. Transformer

Transformation techniques are methods used to change the form or distribution of a data set. Some common transformation techniques, such as the **log** or **Box-Cox functions**, are used to convert data that is not normally distributed into a form that is more symmetrical and follows a bell curve shape. These techniques can be useful when the data needs to meet certain assumptions that are required by certain statistical models or techniques. The

specific transformation method that is used may depend on the desired outcome and the characteristics of the data.



Transformers: Transform the value distribution of your feature—Image by the author

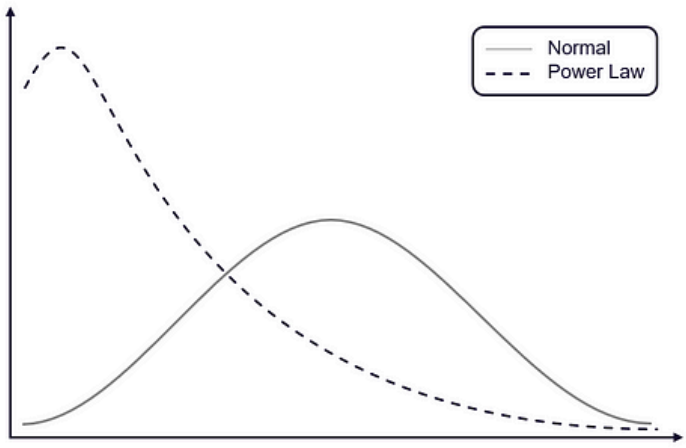
4.1 Log-Transformer using Numpy

Log transformers are used to change the scale of the data by applying a logarithmic function to each value. This transformation is often used to convert highly skewed data into data that more closely resembles a normal distribution.

Skewed data is by no means something unusual, there are various situations where data is naturally or artificially skewed, e.g.:

- The frequency with which words are used in a language follows a pattern known as ***Zipf's law***
- How humans perceive different stimuli follows a pattern described by ***Stevens' power function.***

The distribution of the data is not symmetrical and may have a long tail of values that are much larger or smaller than the majority of the data.



Normal vs. Power Law Distribution—Image by the author (Inspired by [Geeksforgeeks, 2022])

Certain types of algorithms may struggle to handle this type of data and may produce less accurate or reliable results. Applying a power transformation, such as the Box-Cox or log transformation, can help to adjust the scale of the data and make it more suitable for these algorithms to work with. [Geeksforgeeks, 2020]

Let's have a look at how the transformers affect real-world data. Therefore I am using the World Population, Online News Popularity and Boston Housing data sets:

> **Data Set 1: World Population** [License: CC BY 3.0 IGO]
> https://population.un.org/wpp/Download/Standard/CSV/
> The dataset contains the population of every country in the world. [United Nations, 2022]
> **Data Set 2: Online News Popularity** [License: CC0: Public Domain]
> https://archive.ics.uci.edu/ml/datasets/online+news+popularity
> This dataset summarizes a heterogeneous set of features about articles published by Mashable in a period of two years. The goal is to predict the number of shares in social networks (popularity).
> **Data set 3: Boston Housing Dataset** [License: CC0: Public Domain]
> https://www.cs.toronto.edu/~delve/data/boston/bostonDetail.html
> The Boston Housing data set was collected by the U.S. Census Bureau in the late 1970s and early 1980s. It contains information on the median values of homes in the Boston, Massachusetts area. The set includes 13 attributes, such as the average number of rooms per house, average distance to employment centers and property tax rate, as well as the median value of homes in the area.

The following function helps to plot the distributions before and after transforming the data sets:

```
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from scipy import stats
import seaborn as sns

def plot_transformer(chosen_dataset, chosen_transformation,
chosen_feature = None, box_cox_lambda = 0):
   plt.rcParams['font.size'] = '16'
   sns.set_style("darkgrid", {"axes.facecolor": ".9"})


##########################################################################
   # choose dataset
##########################################################################
   if chosen_dataset == "Online News Popularity":
       df =
pd.read_csv("../input/uci-online-news-popularity-data-set/OnlineNewsPopu
```

```
larity.csv")
        X_feature = " n_tokens_content"
    elif chosen_dataset == "World Population":
        df =
pd.read_csv("../input/world-population-dataset/WPP2022_TotalPopulationBy
Sex.csv")
        df = df[df["Time"]==2020]
        df["Area"] = df["PopTotal"] / df["PopDensity"]
        X_feature = "Area"
    elif chosen_dataset == "Housing Data":
        df = pd.read_csv("../input/housing-data-set/HousingData.csv")
        X_feature = "AGE"

    # in case you want to plot the histogram for another feature
    if chosen_feature != None:
        X_feature = chosen_feature


#############################################################################
    # choose type of transformation
#############################################################################
    #chosen_transformation = "box-cox" #"log", "box-cox"

    if chosen_transformation == "log":
        def transform_feature(df, X_feature):
            # We add 1 to number_of_words to make sure we don't have a
null value in the column to be transformed (0-> -inf)
            return (np.log10(1+ df[[X_feature]]))

    elif chosen_transformation == "box-cox":
        def transform_feature(df, X_feature):
            return stats.boxcox(df[[X_feature]]+1, lmbda=box_cox_lambda)
            #return stats.boxcox(df[X_feature]+1)


#############################################################################
    # plot histogram to chosen dataset and X_feature
#############################################################################
    # figure settings
    fig, (ax1, ax2) = plt.subplots(2)
    fig.set_size_inches(18.5, 10.5)

    ax1.set_title(chosen_dataset)
    ax1.hist(df[[X_feature]], 100, facecolor='black', ec="white")
    ax1.set_ylabel(f"Count {X_feature}")
    ax1.set_xlabel(f"{X_feature}")
```

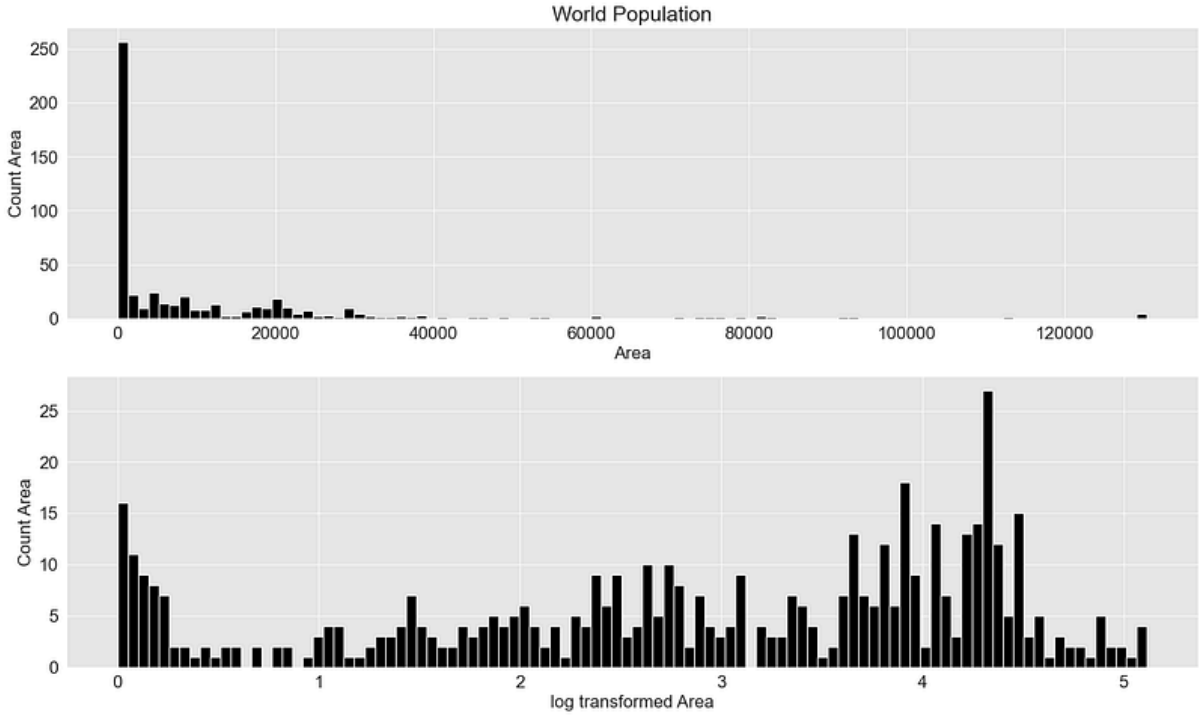
```
   ax2.hist(transform_feature(df, X_feature), 100, facecolor='black',
ec="white")
   ax2.set_ylabel(f"Count {X_feature}")
   ax2.set_xlabel(f"{chosen_transformation} transformed {X_feature}")

   fig.show()
```

In the following, I am not only testing how the data looks like, after the transformation, I am also interested in how this could affect the model-building process. I am using Polynomial Regression to build simple regression models and compare the performance of the models. The input data for the models is just 2-dimensional:

- for the ***Only News Popularity data set*** we try to build a model which is predicting the “shares” based on the “n_tokens” of the online article—so we try to build a model which predicts the popularity of an article based on the number of tokens/length of the article
- for the ***World Population data set,*** we build a simple model which is predicting the population based on only one input feature, the area size of the country

Example 1: World Population data set—Area size of countries

```
plot_transformer(chosen_dataset = "World Population", chosen_transformation = "log")
```

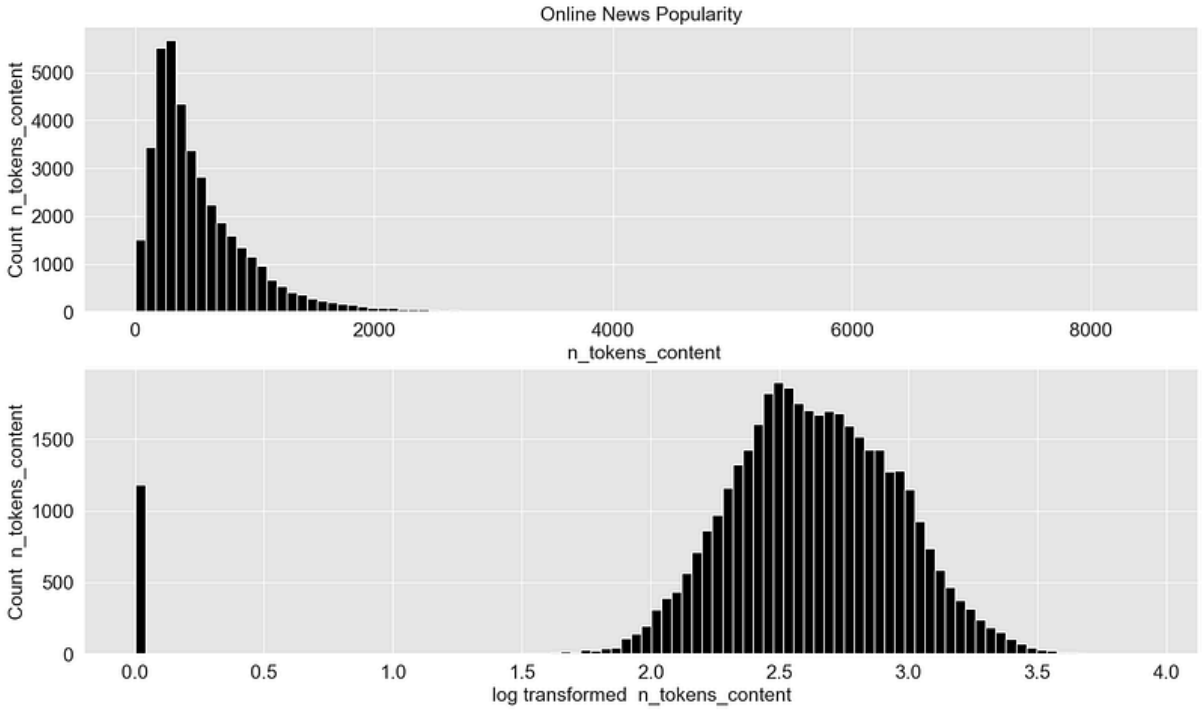


In the graph, you can see that the original distribution, which was skewed to the left, has been transformed into a distribution that is more symmetrical.

Let's try the same for the Online news popularity data set:

Example 2: Online news popularity data set—article popularity based on article length

```
plot_transformer(chosen_dataset = "Online News Popularity", chosen_transformation =
"log")
```



This example shows the effect even better. The skewed data is transformed into almost “perfect” normally distributed data.

But can this have a positive impact on the model-building process? For this purpose, I create polynomial models for the raw data and the transformed data and compare their performance:

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
import plotly.graph_objects as go
import pandas as pd
from scipy import stats
import plotly.express as px
import seaborn as sns

def plot_cross_val_score_comparison(chosen_dataset):
   scores_raw = []
   scores_transformed = []
   degrees = []
```

```
    if chosen_dataset == "Online News Popularity":
        df =
pd.read_csv("../input/uci-online-news-popularity-data-set/OnlineNewsPopu
larity.csv")
        X = df[[" n_tokens_content"]]
        y = df[" shares"]
    elif chosen_dataset == "World Population":
        df =
pd.read_csv("../input/world-population-dataset/WPP2022_TotalPopulationBy
Sex.csv")
        df = df[df["Time"]==2020]
        df["Area"] = df["PopTotal"] / df["PopDensity"]
        X = df[["Area"]]
        y = df["PopTotal"]

    for i in range(1,10):
        degree = i
        polynomial_features = PolynomialFeatures(degree=degree,
include_bias=False)

        linear_regression = LinearRegression()
        pipeline = Pipeline(
            [
                ("polynomial_features", polynomial_features),
                ("linear_regression", linear_regression),
            ]
        )

        # Evaluate the models using crossvalidation
        scores = cross_val_score(
            pipeline, X, y, scoring="neg_mean_absolute_error", cv=5
        )


        scores_raw.append(scores.mean())
        degrees.append(i)


###########################################################################
        # Fit model with transformed data
###########################################################################
        def transform_feature(X):
            # We add 1 to number_of_words to make sure we don't have a
null value in the column to be transformed (0-> -inf)
            #return (np.log10(1+ X))
```

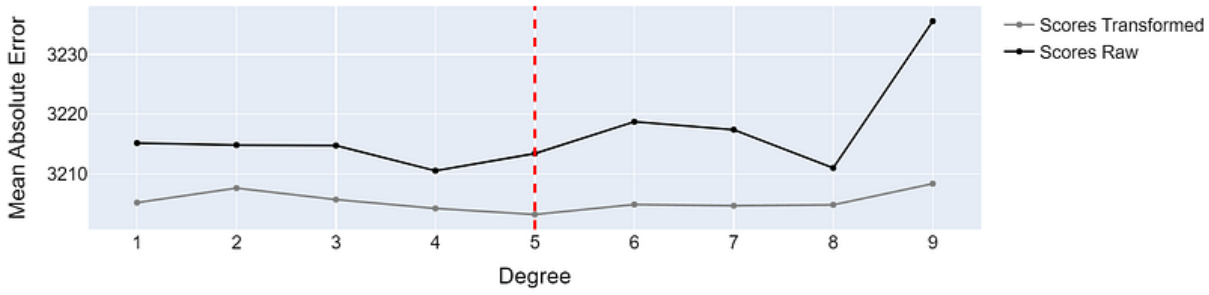
```
            return stats.boxcox(X+1, lmbda=0)

        X_trans = transform_feature(X)

        pipeline.fit(X_trans, y)

        # Evaluate the models using crossvalidation
        scores = cross_val_score(
            pipeline, X_trans, y, scoring="neg_mean_absolute_error", cv=5
        )

        scores_transformed.append(scores.mean())

    plot_df = pd.DataFrame(degrees, columns=["degrees"])
    plot_df = plot_df.assign(scores_raw = np.abs(scores_raw))
    plot_df = plot_df.assign(scores_transformed =
np.abs(scores_transformed))

    fig = go.Figure()
    fig.add_scatter(x=plot_df["degrees"],
y=plot_df["scores_transformed"], name="Scores Transformed",
line=dict(color="#0000ff"))

    # Only thing I figured is - I could do this
    #fig.add_scatter(x=plot_df['degrees'],
y=plot_df['scores_transformed'], title='Scores Raw')
    fig.add_scatter(x=plot_df['degrees'], y=plot_df['scores_raw'],
name='Scores Raw')

    # write scores for raw and transformed data in one data frame and
find degree that shows the mininmal error score
    scores_df = pd.DataFrame(np.array([degrees,scores_raw,
scores_transformed]).transpose(), columns=["degrees", "scores_raw",
"scores_transformed"]).abs()
    scores_df_merged =
scores_df[["degrees","scores_raw"]].rename(columns={"scores_raw":"scores
"}).append(scores_df[["degrees",
"scores_transformed"]].rename(columns={"scores_transformed":"scores"}))
    degree_best_performance =
scores_df_merged[scores_df_merged["scores"]==scores_df_merged["scores"].
min()]["degrees"]

    # plot a vertical line
    fig.add_vline(x=int(degree_best_performance), line_width=3,
line_dash="dash", line_color="red", name="Best Performance")
```

```
    # update plot layout
    fig.update_layout(
        font=dict(
            family="Arial",
            size=18,  # Set the font size here
            color="Black"
        ),
        xaxis_title="Degree",
        yaxis_title="Mean Absolute Error",
        showlegend=True,
        width=1200,
        height=400,
    )

    fig['data'][0]['line']['color']="grey"
    fig['data'][1]['line']['color']="black"

    fig.show()
    return degree_best_performance
```

I am aware this simple 2-dimensional example isn't a representative example, as it is not possible to accurately predict the target variable using only one attribute. However, let's see if the transformation is changing anything at all.

Let's try it first with the Online News Popularity data set:

```
degree_best_performance_online_news =
plot_cross_val_score_comparison(chosen_dataset="Online News Popularity")
```



Performance of a Polynomial Regression Model trained on raw and transformed data—Image by the author

The model trained on the transformed data is doing slightly better. Even though the difference in Absolute Error from 3213 to 3202 is not particularly large, it does indicate that transforming the data can have a positive effect on the training process.

By plotting the transformed data and building the model, we can see, that the data is shifted to the right. This gives the model a little more "room" to fit the data:

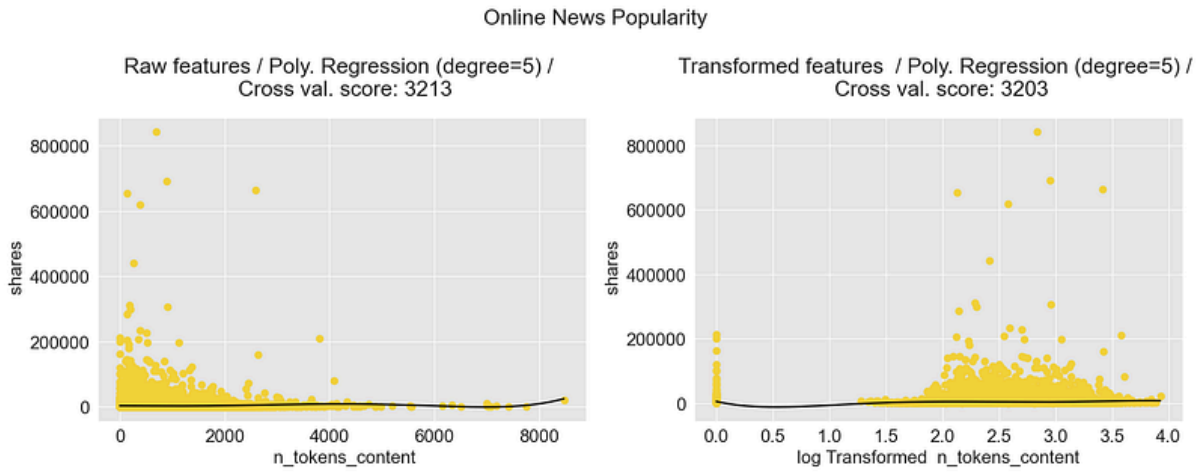
```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline
import numpy as np

def plot_polynomial_regression_model(chosen_dataset,
chosen_transformation, degree_best_performance):
   # fig settings
   plt.rcParams.update({'font.size': 16})
   fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(15, 6))


   if chosen_dataset == "Online News Popularity":
       df =
pd.read_csv("../input/uci-online-news-popularity-data-set/OnlineNewsPopu
larity.csv")
       y_column = " shares"
       X_column = " n_tokens_content"
       X = df[[X_column]]
       y = df[y_column]
   elif chosen_dataset == "World Population":
       df =
pd.read_csv("../input/world-population-dataset/WPP2022_TotalPopulationBy
Sex.csv")
       df = df[df["Time"]==2020]
       df["Area"] = df["PopTotal"] / df["PopDensity"]
       y_column = "PopTotal"
       X_column = "Area"
       X = df[[X_column]]
       y = df[y_column]


##########################################################################
#
   # Define model

##########################################################################
#
   # degree_best_performance was calculated in the cell above
   degree = int(degree_best_performance)
   polynomial_features = PolynomialFeatures(degree=degree,
include_bias=False)
```

```
    linear_regression = LinearRegression()
    pipeline = Pipeline(
        [
            ("polynomial_features", polynomial_features),
            ("linear_regression", linear_regression),
        ]
    )


##########################################################################
    # Fit model
##########################################################################
pipeline.fit(X, y)

##########################################################################
    # Fit model and plot raw features
##########################################################################
    reg = pipeline.fit(X, y)
    X_pred = np.linspace(min(X.iloc[:,0]),
max(X.iloc[:,0]),1000).reshape(-1,1)
    X_pred = pd.DataFrame(X_pred)
    X_pred.columns = [X.columns[0]]
    y_pred_1 = reg.predict(X_pred)

    # plot model and transformed data
    ax[0].scatter(X, y, color='#f3d23aff')
    ax[0].plot(X_pred, y_pred_1, color='black')
    ax[0].set_xlabel(f"{X_column}")
    ax[0].set_ylabel(f"{y_column}")
    ax[0].set_title(f"Raw features / Poly. Regression (degree={degree})",
pad=20)


##########################################################################
    # Fit model with transformed data
##########################################################################
    def transform_feature(X, chosen_transformation):
        # We add 1 to number_of_words to make sure we don't have a null
value in the column to be transformed (0-> -inf)
        if chosen_transformation == "log":
            return (np.log10(1+ X))
        if chosen_transformation == "box-cox":
            return stats.boxcox(X+1, lmbda=0)

    X_trans = transform_feature(X, chosen_transformation)
```

```
    # fit model with transformed data
    reg = pipeline.fit(X_trans, y)

    # define X_pred
    X_pred = np.linspace(min(X_trans.iloc[:,0]),
max(X_trans.iloc[:,0]),1000).reshape(-1,1)
    X_pred = pd.DataFrame(X_pred)
    X_pred.columns = [X.columns[0]]

    # predict
    y_pred_2 = reg.predict(X_pred)

    # plot model and transformed data
    ax[1].scatter(X_trans, y, color='#f3d23aff')
    ax[1].plot(X_pred, y_pred_2, color='black')
    ax[1].set_xlabel(f"{chosen_transformation} Transformed {X_column}")
    ax[1].set_ylabel(f"{y_column}")
    # calc cross val score and add to title
    ax[1].set_title(f"Transformed features  / Poly. Regression
(degree={degree})", pad=20)

    fig.suptitle(chosen_dataset)
    fig.tight_layout()
```

We use the just defined function to plot the polynomial model that showed the best performance:

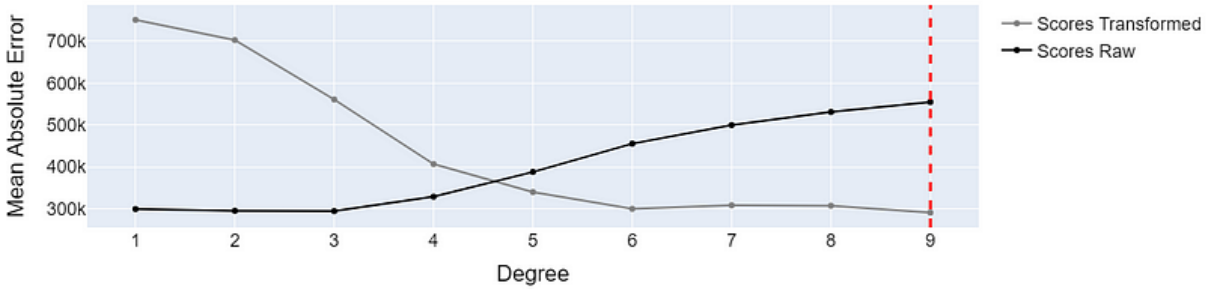
```
plot_polynomial_regression_model(chosen_dataset = "Online News
Popularity", chosen_transformation = "log",
degree_best_performance=degree_best_performance_online_news)
```



Polynomial Regression Model: Shares of online news— Image by the author

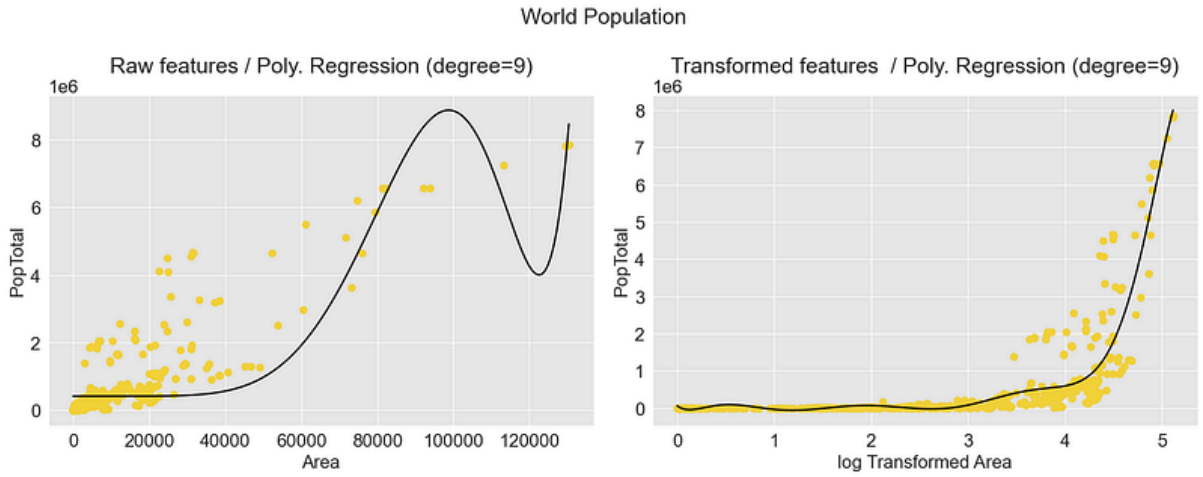
Let's try the same with the World Population Data set:

```
degree_best_performance_world_pop =
plot_cross_val_score_comparison(chosen_dataset="World Population")
```



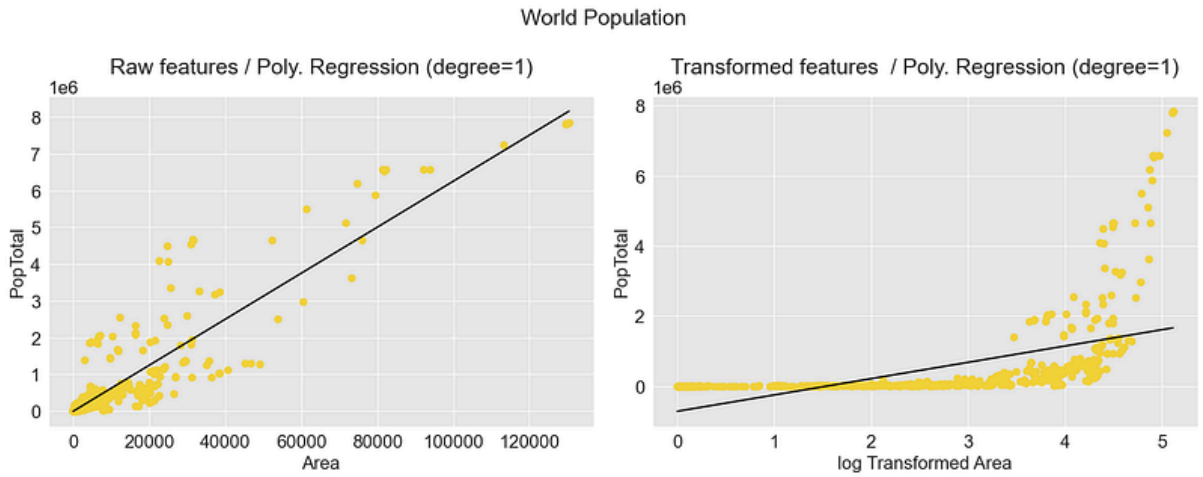
Performance of a Polynomial Regression Model trained on raw and transformed data—Image by the author

We can see that the model performance is quite different between the raw and transformed data sets. While the models based on the raw data perform significantly better at low polynomial degrees, the models trained on the transformed data perform better when we build models with a higher polynomial degree.



Polynomial Regression Model: Country Population—Image by the author

Just for comparison, this is the linear model, that showed the best performance for the raw data:



Polynomial Regression Model: Country Population—Image by the author

---

4.2 Box-Cox Function using Scipy

Another very popular transformer function is the Box-Cox function which belongs to the group of power transformers.

Power transformers are a family of parametric transformations that aim to transform any distribution of data into a data set that is normally distributed and minimize variance and skewness. [Sklearn.org]

To achieve this flexible transformation, the Box-Cox function is defined as follows:

$$x_i^{(\lambda)} = \begin{cases} \dfrac{x_i^{\lambda} - 1}{\lambda} & \text{if } \lambda \neq 0, \\ \ln(x_i) & \text{if } \lambda = 0, \end{cases}$$

To use the Box-Cox function, we must determine the transformation parameter lambda. If you do not manually specify a lambda, the transformer tries to find the best lambda by maximizing the likelihood function. [Rdocumentation.org] A suitable implementation can be found with the function ***scipy.stats.boxcox***.

> NOTE: The Box-Cox function must be used on data that is greater than zero.

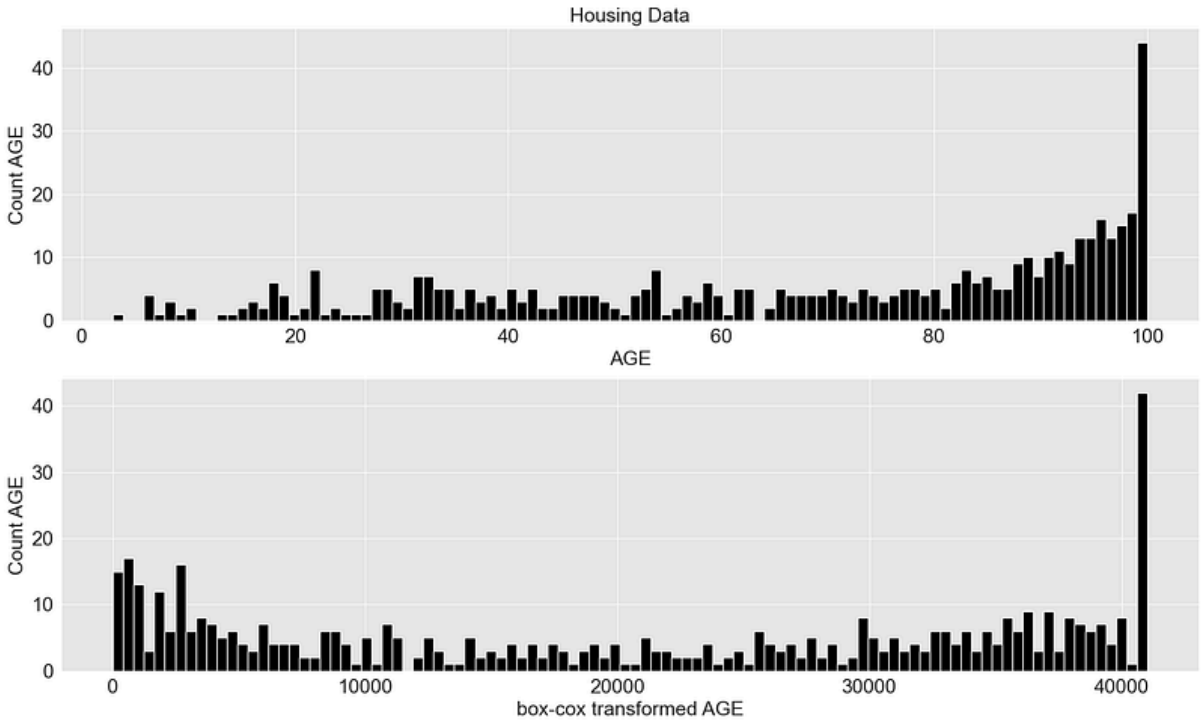
Box-Cox transformation is more flexible than a log transformation because it can produce a variety of transformation shapes, including linear, quadratic and exponential, which may better fit the data. Additionally, Box-Cox transformations can be used to transform data that is both positively and negatively skewed, whereas a log transformation can only be used on positively skewed data.

> What is the lambda parameter used for?

The parameter lambda (λ) can be adjusted to tailor the transformation to the characteristics of the data being analyzed. A value of 0 for lambda produces a log transformation, while values between 0 and 1 create increasingly "*strong*" transformations. If lambda is set to 1, the function does not perform any transformation on the data.

To show that the box-cox function can convert data that is not only skewed to the right, I am also using the ***AGE*** ("proportion of owner-occupied units built before 1940") attribute from the Boston Housing data set.

```
import seaborn as sns

plot_transformer(chosen_dataset = "Housing Data", chosen_transformation
= "box-cox", chosen_feature = None, box_cox_lambda = 2.5)
```



---

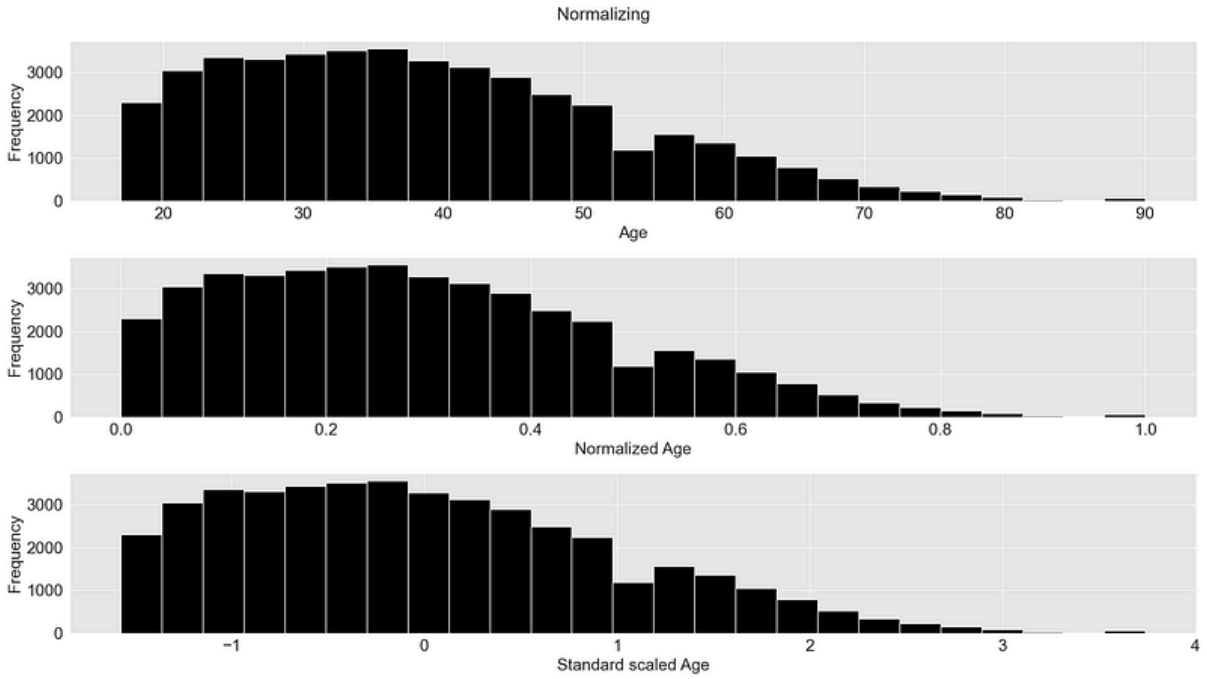
5. Normalize / Standardize

Normalizing and Standardizing are important preprocessing steps in Machine Learning. They can help algorithms to converge faster and can even increase the model accuracy.

5.1 Normalize and Standardize using Scikit-learn

- ***Scikit-learn's MinMaxScaler*** scales features to a given range. It transforms features by scaling each feature to a given range between 0 and 1
- ***Scikit-learn's StandardScaler*** transforms data to have a mean of 0 and a standard deviation of 1

```
# data normalization with sklearn
from sklearn.preprocessing import MinMaxScaler, StandardScaler
import matplotlib.pyplot as plt
import pandas as pd
import seaborn as sns

plt.rcParams['font.size'] = '16'
sns.set_style("darkgrid", {"axes.facecolor": ".9"})

# load data set
census_income = pd.read_csv(r'../input/income/train.csv')

X = census_income[["age"]]

# fit scaler and transform data
X_norm = MinMaxScaler().fit_transform(X)
X_scaled = StandardScaler().fit_transform(X)

# plots
fig, (ax1, ax2, ax3) = plt.subplots(3)
fig.suptitle('Normalizing')
fig.set_size_inches(18.5, 10.5)

# subplot 1 - raw data
ax1.hist(X, 25, facecolor='black', ec="white")
ax1.set_xlabel("Age")
ax1.set_ylabel("Frequency")

# subplot 2 - normalizer
ax2.hist(X_norm, 25, facecolor='black', ec="white")
ax2.set_xlabel("Normalized Age")
ax2.set_ylabel("Frequency")

# subplot 3 - standard scaler
ax3.hist(X_scaled, 25, facecolor='black', ec="white")
ax3.set_xlabel("Normalized Age")
ax3.set_ylabel("Frequency")

fig.tight_layout()
```



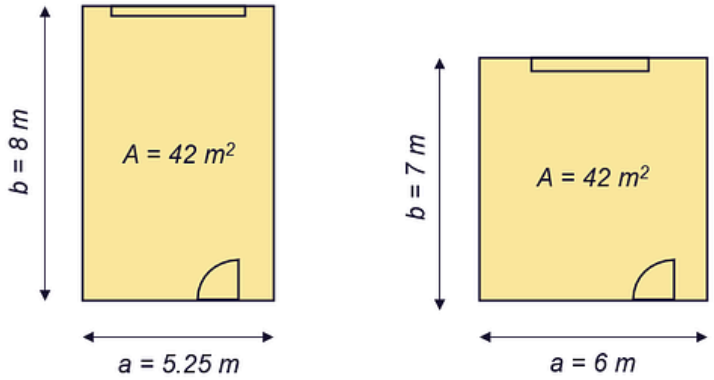
---

6. Feature Crossing

Feature crossing is the process of connecting multiple features from a dataset to create a new feature. This can include combining data from other sources to emphasize existing correlations.

There are no limits to creativity. First of all, it makes sense to make known correlations apparent by combining the attributes correctly, e.g.:

> ***Example 1: Predicting the price of an apartment***
> Let's say we have a series of apartments we want to sell and have the technical blueprints and thus the dimensions of the apartments available.
> To determine a reasonable price for the apartments, the specific dimensions are probably not as important as the total area: whether a room is 6 m * 7 m or 5.25 m * 8 m is not as important as the fact that the room has 42 m². If I only have the dimensions a and b, it makes sense to add the area as a feature as well.



Predicting the price of an apartment—Image by the author

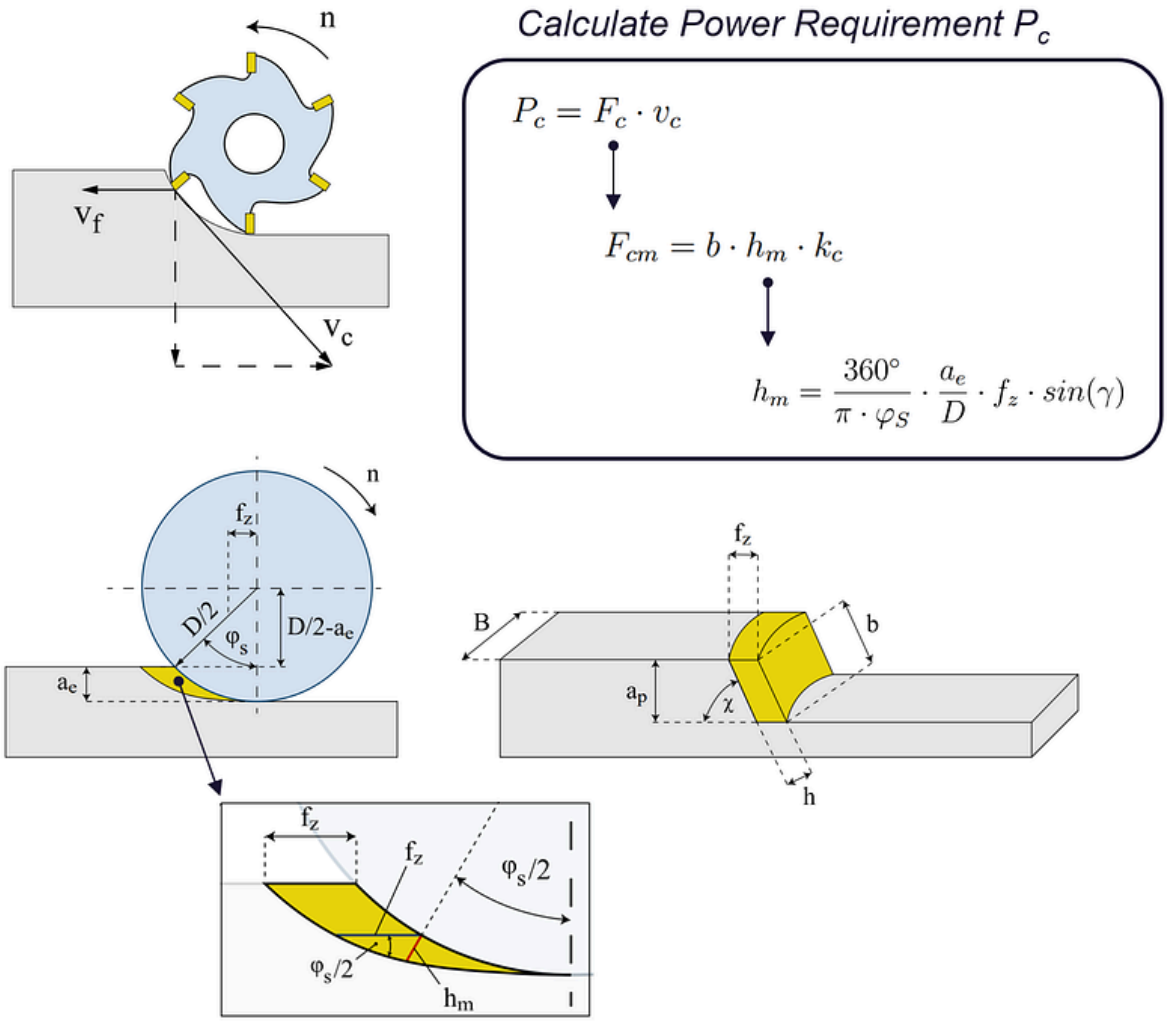
Example 2: Use technical know-how —Predicting the energy consumption of milling machines

A few years back, I was working on a regression model that would allow me to predict the energy consumption of a milling machine.

When implementing a solution for a specific domain, it is always worthwhile to look at the available literature. Since people were interested in being able to calculate the power requirements of a cutting machine for minimum 50 years, there are 50 years of research we can use.

Even if the existing formulas are only suitable for a rough calculation, we can use the know-how about identified correlations between attributes and the target variable, the energy consumption. In the figure below you can see some known relationships between the power demand and the variables.

→ We can make it easier for our model by highlighting these known relationships as features. We could for example cross the **width of cut b** and the **cutting depth h** (to calculate the **cross-sectional area A**) and define it as a new feature. This could help the training process, especially if we are using less complex models.



Calculation of the power requirement of a milling machine—Image by the author

But we don't just use it to prepare our dataset, some algorithms use Feature Crossing as a fundamental part of how they work.

> Some ML methods and algorithms, already use feature crossing by default

Two well-known ML techniques that use feature crossing are **polynomial regression** and the **kernel trick (e.g., in combination with support vector regression)**.

6.1 Feature Crossing in Polynomial Regression

Scikit-learn does not contain a function for polynomial regression, we create a pipeline out of:

1. a feature transformation step and
2. a linear regression model-building step.

By combining and exponentiating features we generate several new features, which makes it possible to represent also non-linear relationships between the output variables.

Let's say we want to build a regression model with a 2-dimensional input matrix X and the target variable y. Unless specifically defined, the feature transformation function (*sklearn.PolynomialFeatures*) transforms the matrix as follows:

$$X = \begin{pmatrix} x_{11} & x_{12} \\ x_{11} & x_{12} \end{pmatrix} \rightarrow X = \begin{pmatrix} 1 & x_{11} & x_{12} & x_{11}^2 & x_{11}x_{12} & x_{12}^2 \\ 1 & x_{21} & x_{22} & x_{21}^2 & x_{21}x_{22} & x_{22}^2 \end{pmatrix}$$

You can see that the new matrix contains four new columns. The attributes are not only potentiated but also mutually crossed:

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

# Raw input features
X = np.arange(6).reshape(3, 2)
print("Raw input matrix:")
display(X)

# Crossed features
poly = PolynomialFeatures(2)
print("Transformed feature matrix:")
poly.fit_transform(X)
```

```
Raw input matrix:

array([[0, 1],
       [2, 3],
       [4, 5]])

Transformed feature matrix:

array([[ 0.,  1.,  0.,  0.,  1.],
       [ 2.,  3.,  4.,  6.,  9.],
       [ 4.,  5., 16., 20., 25.]])
```

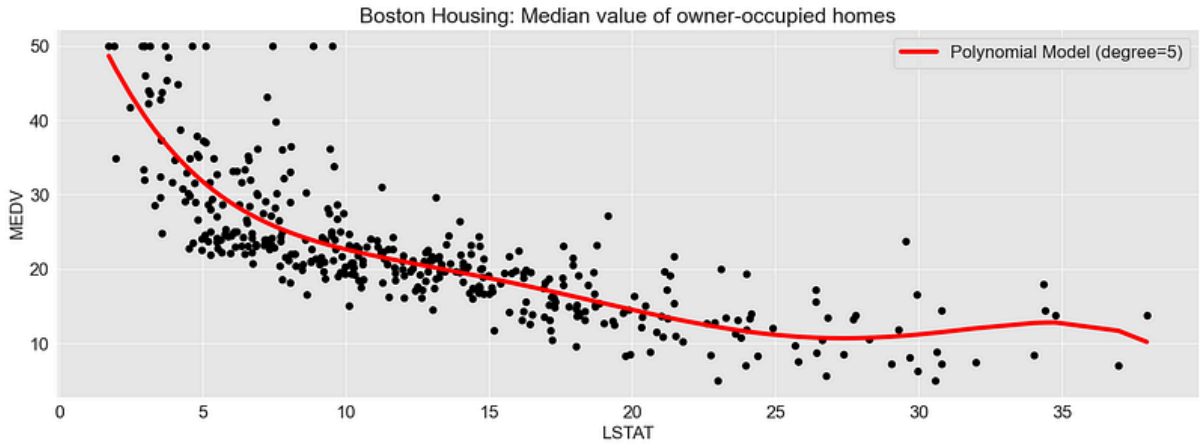
This way we can model any imaginable relationship, we just have to choose the right polynomial degree.

The following example shows a polynomial regression model (polynomial degree=5) which is predicting the median value of owner-occupied homes in Boston based on the attribute LSTAT (*percentage of the population that has lower socio-economic status*).

```
from sklearn.preprocessing import PolynomialFeatures
```

```
from sklearn.linear_model import LinearRegression
from sklearn.pipeline import Pipeline

import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

plt.rcParams['font.size'] = '16'
sns.set_style("darkgrid", {"axes.facecolor": ".9"})

# load data set: Boston Housing
boston_housing_df =
pd.read_csv("../input/housing-data-set/HousingData.csv")
boston_housing_df = boston_housing_df.dropna()
boston_housing_df = boston_housing_df.sort_values(by=["LSTAT"])

# define x and target variable y
X = boston_housing_df[["LSTAT"]]
y = boston_housing_df["MEDV"]

# fit model and create predictions
degree = 5
polynomial_features = PolynomialFeatures(degree=degree)
linear_regression = LinearRegression()
pipeline = Pipeline(
   [
       ("polynomial_features", polynomial_features),
       ("linear_regression", linear_regression),
   ]
)
pipeline.fit(X, y)

y_pred = pipeline.predict(X)

# train linear model
regr = LinearRegression()
regr.fit(X,y)

# figure settings
fig, (ax1) = plt.subplots(1)
fig.set_size_inches(18.5, 6)

ax1.set_title("Boston Housing: Median value of owner-occupied homes")
ax1.scatter(X, y, c="black")
ax1.plot(X, y_pred, c="red", linewidth=4, label=f"Polynomial Model
(degree={degree})")
```

```
ax1.set_ylabel(f"MEDV")
ax1.set_xlabel(f"LSTAT")
ax1.legend()

fig.show()
```



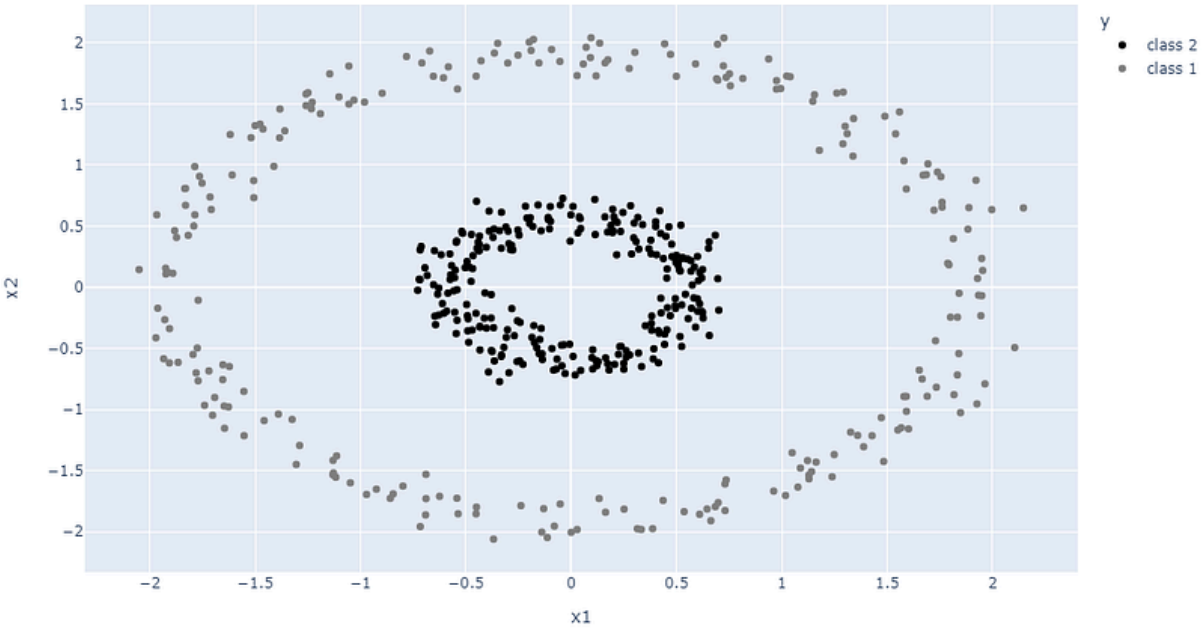
6.2 Feature Crossing and the Kernel-Trick

Another example where feature crossing is already inherently used is the kernel trick, for example in Support Vector Regression. Through a kernel function, we transform the data set into a higher dimensional space, which makes it possible to easily separate classes from each other.

For the following example, I am generating a 2-dimensional data set, which is pretty hard to separate in a 2-dimensional space, at least for linear models.

```
from sklearn.datasets import make_circles
from sklearn import preprocessing
import plotly_express as px
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd


sns.set_style("darkgrid", {"axes.facecolor": ".9"})
plt.rcParams['font.size'] = '30'

# generate a data set
X, y = make_circles(n_samples=1_000, factor=0.3, noise=0.05,
random_state=0)
X = preprocessing.scale(X)
```

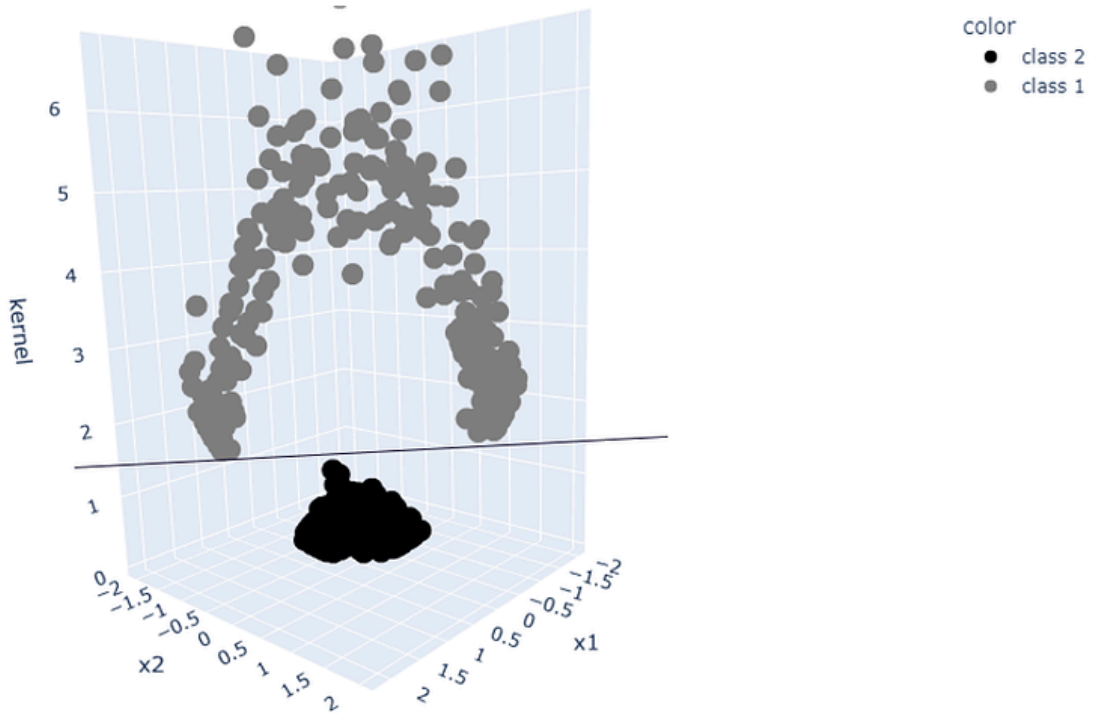
```
X=X[500:]
y=y[500:]

# define target value, here: binary classification, class 1 or class 2
y=np.where(y==0,"class 1","class 2")

# define x1 and x2 of a 2-dimensional data set
x1 = X[:,0]
x2 = X[:,1]

import plotly_express as px
# define the kernel function
kernel = x1*x2 + x1**2 + x2**2

circle_df = pd.DataFrame(X).rename(columns={0:"x1", 1:"x2"})
circle_df = circle_df.assign(y=y)

color_discrete_map = {circle_df.y.unique()[0]: "black",
circle_df.y.unique()[1]: "#f3d23a"}
px.scatter(circle_df, x="x1", y="x2", color="y", color_discrete_map =
color_discrete_map, width=1000, height=800)

# plot the data set together with the kernel value in a 3-dimensional
space
color_discrete_map = {circle_df.y.unique()[0]: "black",
circle_df.y.unique()[1]: "grey"}
fig = px.scatter(circle_df, x="x1", y="x2", color="y",
color_discrete_map = color_discrete_map, width=1000, height=600)
fig.update_layout(
   font=dict(
       family="Arial",
       size=24,  # Set the font size here
       color="Black"
   ),
   showlegend=True,
   width=1200,
   height=800,
)
```



By using the kernel trick, we generate a 3rd dimension. For this example, the kernel function is defined as:

$$kernel = x_1 * x_2 + x_1^2 + x_2^2$$

```
import plotly_express as px

# define the kernel function
kernel = x1*x2 + x1**2 + x2**2

kernel_df = pd.DataFrame(X).rename(columns={0:"x1", 1:"x2"})
kernel_df = kernel_df.assign(kernel=kernel)
kernel_df = kernel_df.assign(y=y)

# plot the data set together with the kernel value in a 3-dimensional
space
color_discrete_map = {kernel_df.y.unique()[0]: "black",
kernel_df.y.unique()[1]: "grey"}
px.scatter_3d(kernel_df, x="x1", y="x2", z="kernel", color="y",
width=1000, height=600)
```



Kernel-trick: A 2-dimensional dataset transferred to a 3-dimensional space—Image by the author

The three-dimensional space allows the data to be separated using a simple linear classifier.

---

7. Principal Component Analysis (PCA)

Principal Component Analysis, or PCA, reduces the dimension of a dataset by creating a set of new features derived from the raw features. Thus, similar to hashing, we reduce the complexity of the dataset and thus the computational effort required.

In addition, it can help us visualize the data set. Data sets with multiple dimensions can be visualized in a lower-dimensional representation, while still preserving as much of the original variation as possible.

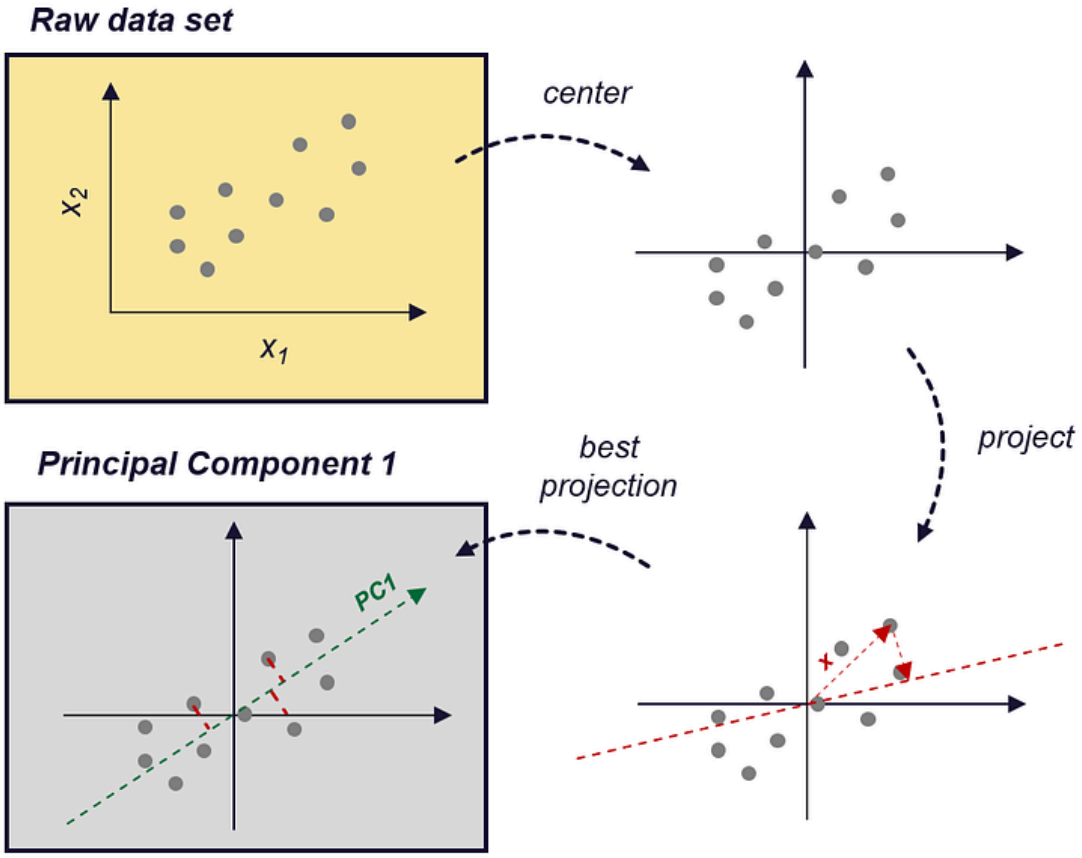
I'll leave out a more in-depth explanation of how it works in this section, as there are already some excellent sources on the subject, e.g.:

- Josh Starmer: StatQuest: Principal Component Analysis (PCA), Step-by-Step
- Sebastian Raschka: Principal Component Analysis in 3 simple steps

As a quick summary, here are the 4 most important steps:

1. **Standardize the data:** Subtract the mean from each feature and scale the features to unit variance.

2. **Calculate the covariance matrix of the standardized data:** This matrix will contain the pairwise covariances between all of the features.
3. **Compute the eigenvectors and eigenvalues of the covariance matrix:** The eigenvectors determine the directions of the new feature space and represent the principal components and the eigenvalues determine their magnitude.
4. **Select the principal components:** Select the first few principal components (usually the ones with the highest eigenvalues) and use them to project the data onto a lower-dimensional space. You can choose the number of components to keep, based on the amount of variance you want to preserve or the number of dimensions you want to reduce the data to.



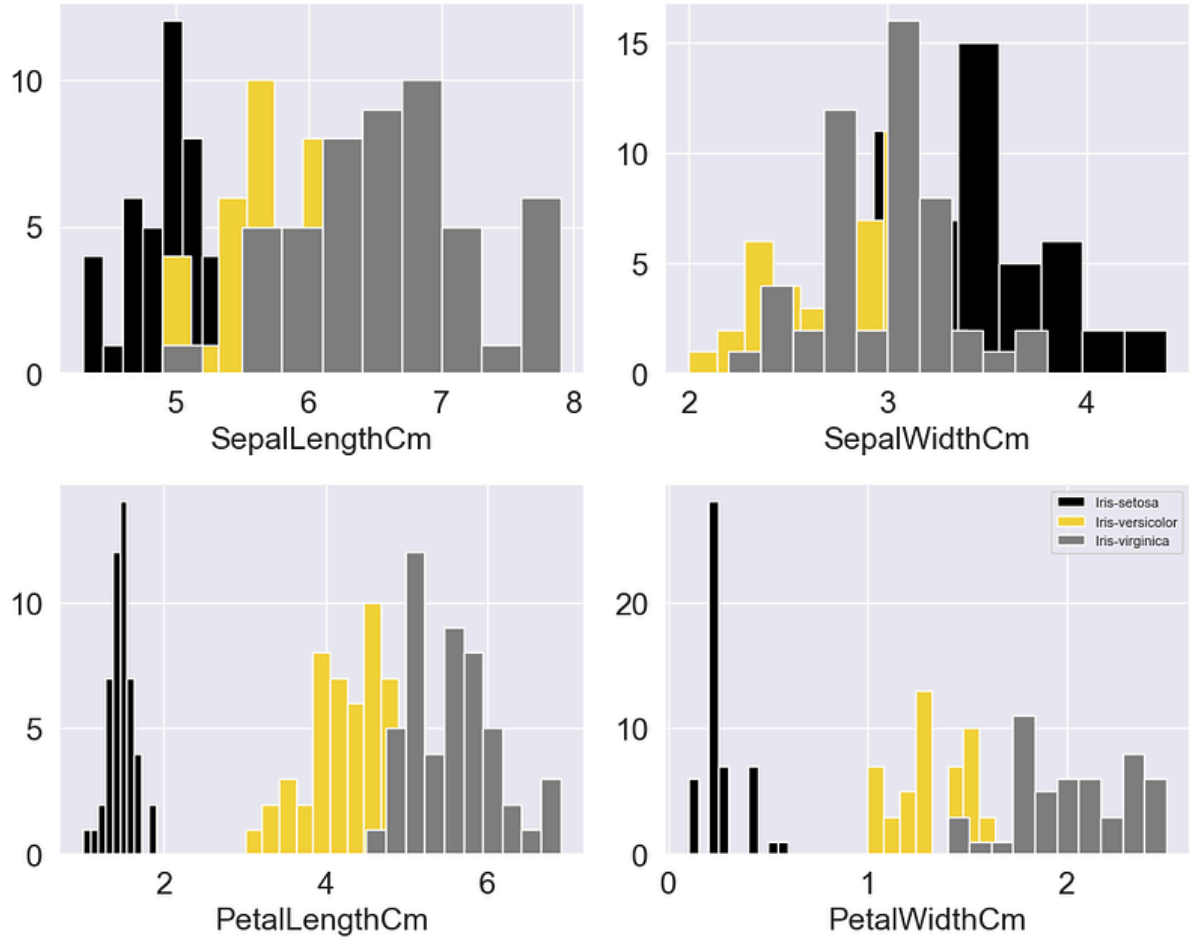
How PCA works—Image by the author (inspired by [Zheng, Alice, and Amanda Casari, 2018])

I'll show you how you can use PCA to create new features using the ***iris data set.***

> **Data Set: Iris Data Set** [License: CC0: Public Domain]
> https://archive.ics.uci.edu/ml/datasets/iris
> https://www.kaggle.com/datasets/uciml/iris
> The Iris dataset is a dataset containing information about three species of Iris flowers (Iris setosa, Iris virginica and Iris versicolor). It includes data on the length and width of the sepals and petals of the flowers in centimeters and contains 150 observations.

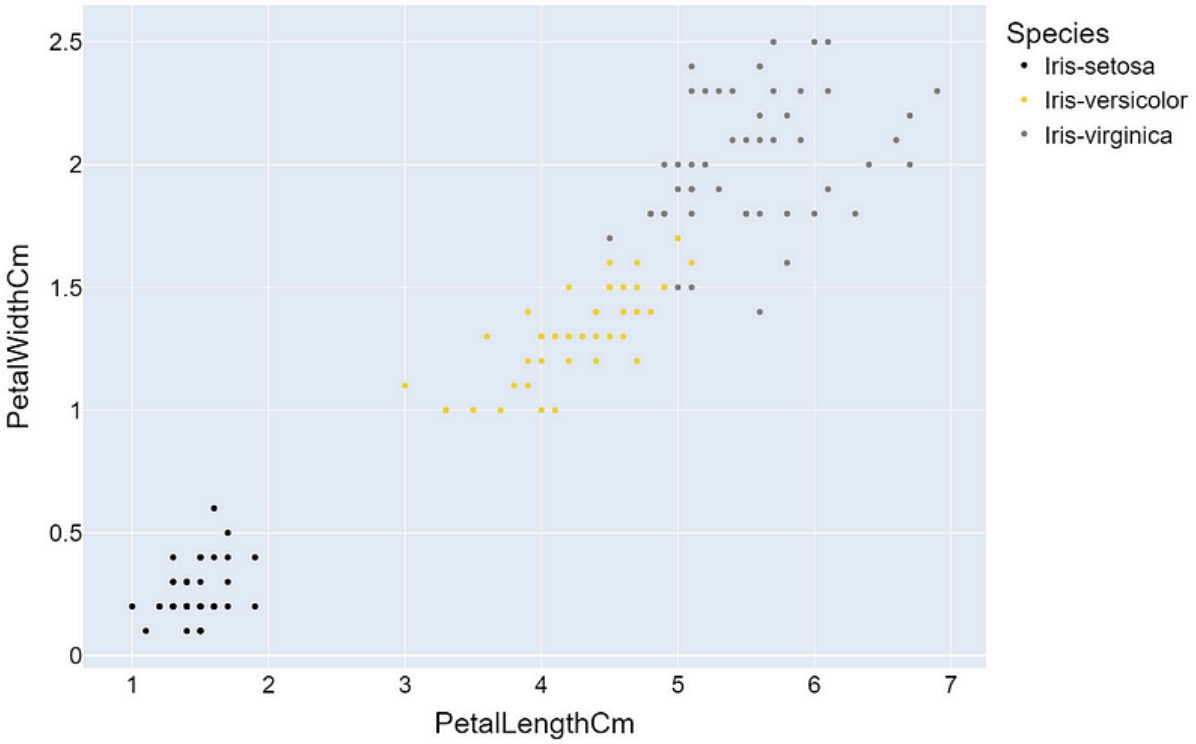
So first, let's load the data set and have a look at the distribution of the data in the four dimensions (Sepal Length, Sepal Width, Petal Length, Petal Width)

```
from matplotlib import pyplot as plt
import numpy as np
import seaborn as sns
import math

import pandas as pd

# Load the Iris dataset
iris_data = pd.read_csv(r'../input/iris-data/Iris.csv')
iris_data.dropna(how="all", inplace=True) # drops the empty line at
file-end

sns.set_style("whitegrid")
colors = ["black", "#f3d23aff", "grey"]
plt.figure(figsize=(10, 8))

with sns.axes_style("darkgrid"):
   for cnt, column in enumerate(iris_data.columns[1:5]):
       plt.subplot(2, 2, cnt+1)
       for species_cnt, species in
enumerate(iris_data.Species.unique()):
           plt.hist(iris_data[iris_data.Species == species][column],
label=species, color=colors[species_cnt])

       plt.xlabel(column)
   plt.legend(loc='upper right', fancybox=True, fontsize=8)

   plt.tight_layout()
   plt.show()
```



You can already see from the distribution that the individual classes can be distinguished relatively well based on the PetalWidth and PetalLength. In the other two dimensions, the distributions overlap strongly.

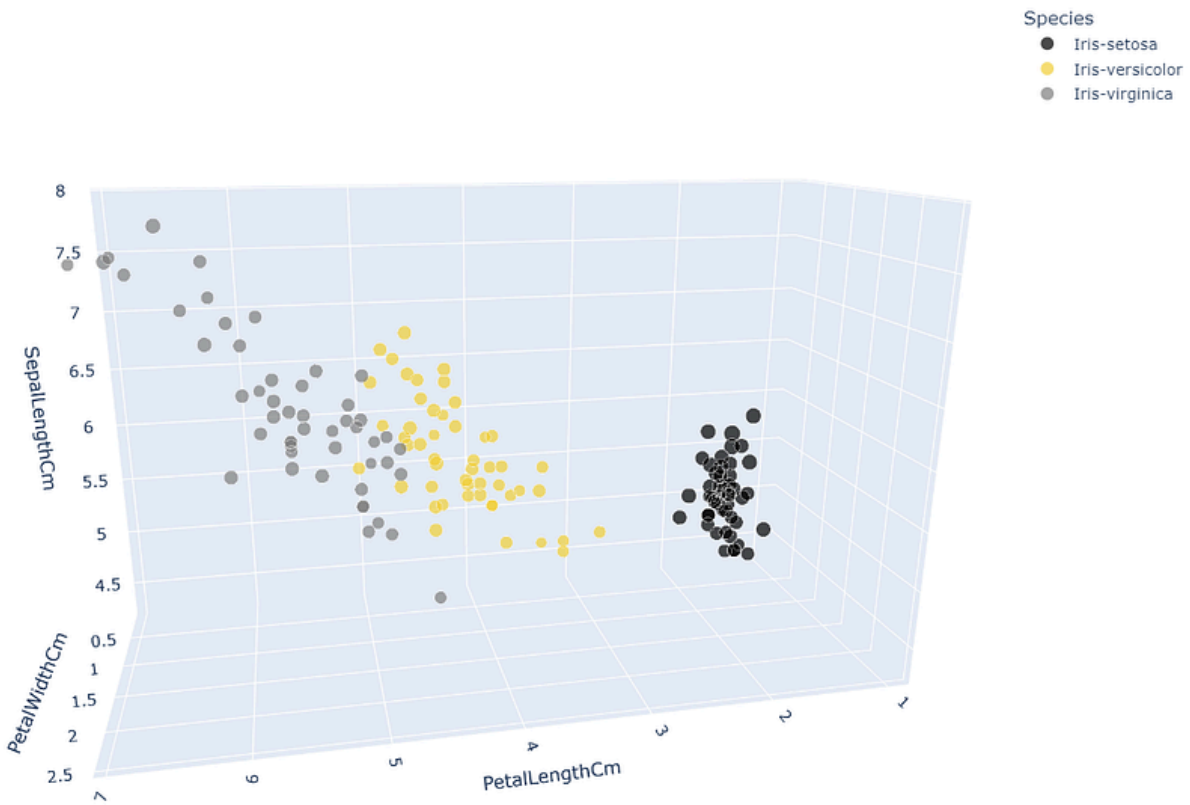
This suggests that the PetalWidth and PetalLength are probably more important for our model than the other two dimensions. So if I were only allowed to use 2 dimensions as features for my model, I would take these two. If we plot these two dimensions, we get the following figure:



Not too bad, we see that by choosing the “most important” attributes we can reduce the dimension and lose as little information as possible.

PCA goes one step further, it centers and scales the data and projects the data onto newly generated dimensions. This way we are not bound to the existing dimensions. Similar to a 3D representation of a dataset that we rotate in space until we find an orientation that allows us to separate the classes as easily as possible:

```
import plotly_express as px

color_discrete_map = {iris_data.Species.unique()[0]: "black",
iris_data.Species.unique()[1]: "#f3d23a",
iris_data.Species.unique()[2]:"grey"}

px.scatter_3d(iris_data, x="PetalLengthCm", y="PetalWidthCm",
z="SepalLengthCm", size="SepalWidthCm", color="Species",
color_discrete_map = color_discrete_map, width=1000, height=800)
```



Iris Data set visualized in a 3d plot—Image by the Author

Overall, then, PCA transforms, scales and rotates the data until a new set of dimensions (the principal components) is found that capture the most important information of the original data.

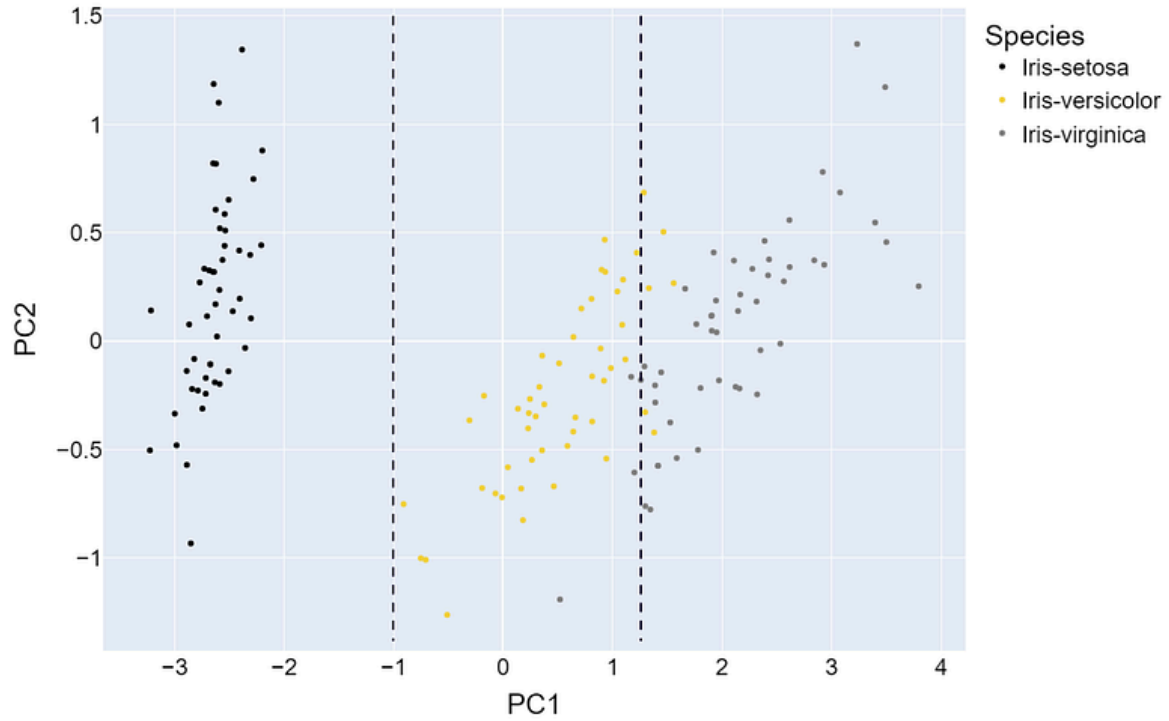
A suitable function to apply PCA to your data set can be found in Scikit-learn, ***sklearn.decomposition.PCA***.

7.1 PCA using Scikit-learn

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
import matplotlib
import plotly.graph_objects as go

# Load the Iris dataset
iris_data = pd.read_csv(r'../input/iris-data/Iris.csv')
iris_data.dropna(how="all", inplace=True) # drops the empty lines

# define X
X = iris_data[["PetalLengthCm", "PetalWidthCm", "SepalLengthCm",
"SepalWidthCm"]]

# fit PCA and transform X
```

```
pca = PCA(n_components=2).fit(X)
X_transform = pca.transform(X)
iris_data_trans = pd.DataFrame(X_transform).assign(Species =
iris_data.Species).rename(columns={0:"PCA1", 1:"PCA2"})

# plot 2d plot
color_discrete_map = {iris_data.Species.unique()[0]: "black",
iris_data.Species.unique()[1]: "#f3d23a",
iris_data.Species.unique()[2]:"grey"}
fig = px.scatter(iris_data_trans, x="PCA1", y="PCA2", color="Species",
color_discrete_map = color_discrete_map)

fig.update_layout(
   font=dict(
       family="Arial",
       size=18,  # Set the font size here
       color="Black"
   ),
   showlegend=True,
   width=1200,
   height=800,
)
```

If you compare the plot of the first two principal components (PC1 and PC2) with the two-dimensional plot of PetalWidth and PetalLength, you can see that the most important information in the data set is preserved. In addition, the data is centered and scaled.



PCA1 and PCA2 of the transformed Iris data—Image by the author

---

Summary

The techniques described in the article can be applied to any type of data, but they can not replace methods that are specific to the field you are working in.

A good example is the field of acoustics. Let's assume you have no idea about acoustics and signal processing ….

- How would you process the airborne sound signal and what features would you extract from it?

Would you think of decomposing the signal into individual spectral components to understand the composition of the signal? Probably not. Fortunately, someone before you has asked the same question and defined some suitable transformation methods, such as the Fast Fourier Transform (FFT).

> Standing on the shoulders of giants

Even though the technology is advanced, that doesn't mean we can't use insights that were made decades ago. We can learn from these past experiences and use them to make our model-building process more efficient. So it's always a good idea to spend some time to understand the field your data is from if you want to create models that are effective at solving real-world problems.

> **Continue reading…**

If you enjoyed reading and want to continue reading about Machine Learning concepts, algorithms and applications, you can find a list of my related articles here:

Machine Learning: Concepts, Techniques and Applications
dmnkplzr.medium.com

If you are interested in signing up with Medium to get unlimited access to all stories, you can support me by using my referral link. This will earn me a small commission at no additional cost to you.

Thanks for reading!

References

Box, G E P, and D R Cox. "An Analysis of Transformations." : 43.

Brown, Sara. 2022. "Why It's Time for 'Data-Centric Artificial Intelligence.'" *MIT Sloan*. https://mitsloan.mit.edu/ideas-made-to-matter/why-its-time-data-centric-artificial-intelligence (October 30, 2022).

educative.io. "Feature Selection and Feature Engineering—Machine Learning System Design." *Educative: Interactive Courses for Software Developers*. https://www.educative.io/courses/machine-learning-system-design/q2AwDN4nZ73 (November 25, 2022).

Feature Engineering with H2O—Dmitry Larko, Senior Data Scientist, H2O.Ai. 2017. https://www.youtube.com/watch?v=irkV4sYExX4 (September 5, 2022).

featureranking.com. "Case Study: Predicting Income Status." *[www.featureranking.com*.](http://www.featureranking.com./) https://www.featureranking.com/tutorials/machine-learning-tutorials/case-study-predicting-income-status/ (November 26, 2022).

Geeksforgeeks. 2020. "Python | Box-Cox Transformation." *GeeksforGeeks*. https://www.geeksforgeeks.org/box-cox-transformation-using-python/ (December 25, 2022).

Google Developers. 2017. "Intro to Feature Engineering with TensorFlow—Machine Learning Recipes #9." https://www.youtube.com/watch?v=d12ra3b_M-0 (September 5, 2022).

Google Developers. "Introducing TensorFlow Feature Columns." https://developers.googleblog.com/2017/11/introducing-tensorflow-feature-columns.html (September 21, 2022).

Heavy.ai. 2022. "What Is Feature Engineering? Definition and FAQs | HEAVY.AI." https://www.heavy.ai/technical-glossary/feature-engineering (September 19, 2022).

heavy.ai. "Feature Engineering." https://www.heavy.ai/technical-glossary/feature-engineering.

Koehrsten, Will. “Introduction to Manual Feature Engineering.” https://kaggle.com/code/willkoehrsen/introduction-to-manual-feature-engineering (September 5, 2022).

Kousar, Summer. “EDA on Cyber Security Salary.” https://kaggle.com/code/summerakousar/eda-on-cyber-security-salary (September 7, 2022).

Moody, John. 1988. “Fast Learning in Multi-Resolution Hierarchies.” In *Advances in Neural Information Processing Systems*, Morgan-Kaufmann. https://proceedings.neurips.cc/paper/1988/hash/82161242827b703e6acf9c726942a1e4-Abstract.html (November 28, 2022).

Poon, Wing. 2022. “Feature Engineering for Machine Learning (1/3).” *Medium*. https://towardsdatascience.com/feature-engineering-for-machine-learning-a80d3cdfede6 (September 19, 2022).

Poon, Wing. “Feature Engineering for Machine Learning (2/3) Part 2: Feature Generation.” : 10.

pydata.org. “Pandas.Get_dummies—Pandas 1.5.2 Documentation.” https://pandas.pydata.org/docs/reference/api/pandas.get_dummies.html (November 25, 2022).

Raschka, Sebastian. “Principal Component Analysis.” *Sebastian Raschka, PhD*. https://sebastianraschka.com/Articles/2015_pca_in_3_steps.html (December 17, 2022).

Rdocumentation.org. “Boxcox Function—RDocumentation.” https://www.rdocumentation.org/packages/EnvStats/versions/2.7.0/topics/boxcox (December 25, 2022).

Sarkar, Dipanjan. 2019a. “Categorical Data.” *Medium*. https://towardsdatascience.com/understanding-feature-engineering-part-2-categorical-data-f54324193e63 (September 19, 2022).

Sarkar, Dipanjan. 2019b. “Continuous Numeric Data.” *Medium*. https://towardsdatascience.com/understanding-feature-engineering-part-1-continuous-numeric-data-da4e47099a7b (September 19, 2022).

Sarkar, Dipanjan, Raghav Bali, and Tushar Sharma. 2018. *Practical Machine Learning with Python*. Berkeley, CA: Apress. http://link.springer.com/10.1007/978-1-4842-3207-1 (November 25, 2022).

Siddhartha. 2020. “Demonstration of TensorFlow Feature Columns (Tf.Feature_column).” *ML Book*. https://medium.com/ml-book/demonstration-of-tensorflow-feature-columns-tf-feature-column-3bfcca4ca5c4 (September 21, 2022).

Sklearn.org. “6.2. Feature Extraction.” *scikit-learn*. https://scikit-learn/stable/modules/feature_extraction.html (November 28, 2022a).

spark.apache.org. “Extracting, Transforming and Selecting Features—Spark 3.3.1 Documentation.” https://spark.apache.org/docs/latest/ml-features (December 1, 2022).

Tensorflow.org. “Tf.One_hot | TensorFlow v2.11.0.” *TensorFlow*. https://www.tensorflow.org/api_docs/python/tf/one_hot (November 25, 2022).

United Nations, Department of Economic and Social Affairs, Population Division (2022). *World Population Prospects 2022, Online Edition*.

Valohai.com. 2022. “What Is a Machine Learning Pipeline?” https://valohai.com/machine-learning-pipeline/ (September 19, 2022).

Votava, Adam. 2022. “Keeping Up With Data #105.” *Medium*. https://adamvotava.medium.com/keeping-up-with-data-105-6a2a8a41f4b6 (October 21, 2022).

Weinberger, Kilian et al. 2009. “Feature Hashing for Large Scale Multitask Learning.” In *Proceedings of the 26th Annual International Conference on Machine Learning—ICML ’09*, Montreal, Quebec, Canada: ACM Press, 1–8. http://portal.acm.org/citation.cfm?doid=1553374.1553516 (December 25, 2022).

What Makes a Good Feature?—Machine Learning Recipes #3. 2016. https://www.youtube.com/watch?v=N9fDIAflCMY (September 5, 2022).

Wikimedia. “Average yearly temperature per country.png.” https://commons.wikimedia.org/wiki/File:Average_yearly_temperature_per_country.png (December 25, 2022).

Wikipedia. 2022. “Feature Hashing.” *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Feature_hashing&oldid=1114513799 (November 28, 2022).

Zheng, Alice and Amanda Casari. 2018. *Feature Engineering for Machine Learning*.