# What Is Feature Engineering:

You might perform feature engineering to:

* improve a model's predictive performance
* reduce computational or data needs
* improve interpretability of the results

example:

## Step1 : establish a base line score

import pandas as pd

from sklearn.ensemble import RandomForestRegressor

from sklearn.model\_selection import cross\_val\_score

df = pd.read\_csv("../input/fe-course-data/concrete.csv")

df.head()

X = df.copy()

y = X.pop("CompressiveStrength")

*# Train and score baseline model*

baseline = RandomForestRegressor(criterion="absolute\_error", random\_state=0)

baseline\_score = cross\_val\_score(

baseline, X, y, cv=5, scoring="neg\_mean\_absolute\_error"

)

baseline\_score = -1 \* baseline\_score.mean()

print(f"MAE Baseline Score: **{**baseline\_score**:**.4**}**")

MAE Baseline Score: 8.232

## Step 2: create new feature and test the result

X = df.copy()

y = X.pop("CompressiveStrength")

*# Create synthetic features*

X["FCRatio"] = X["FineAggregate"] / X["CoarseAggregate"]

X["AggCmtRatio"] = (X["CoarseAggregate"] + X["FineAggregate"]) / X["Cement"]

X["WtrCmtRatio"] = X["Water"] / X["Cement"]

*# Train and score model on dataset with additional ratio features*

model = RandomForestRegressor(criterion="absolute\_error", random\_state=0)

score = cross\_val\_score(

model, X, y, cv=5, scoring="neg\_mean\_absolute\_error"

)

score = -1 \* score.mean()

print(f"MAE Score with Ratio Features: **{**score**:**.4**}**")

MAE Score with Ratio Features: 7.948

# Mutual Information:

Mutual information describes relationships in terms of uncertainty.

mutual information (MI) between two quantities is a measure of the extent to which knowledge of one quantity reduces uncertainty about the other.

A great first step is to construct a ranking with a feature utility metric, a function measuring associations between a feature and the target.

The metric we'll use is called "mutual information"

* easy to use and interpret,
* computationally efficient,
* theoretically well-founded,
* resistant to overfitting, and,
* able to detect any kind of relationship

**Technical note:** What we're calling uncertainty is measured using a quantity from information theory known as "**entropy**". The entropy of a variable means roughly: "how many yes-or-no questions you would need to describe an occurance of that variable, on average." The more questions you have to ask, the more uncertain you must be about the variable. Mutual information is how many questions you expect the feature to answer about the target

Categoricals (object or categorial dtype) can be treated as discrete by giving them a label encoding.

X = df.copy()

y = X.pop("price")

*# Label encoding for categoricals*

for colname **in** X.select\_dtypes("object"):

X[colname], \_ = X[colname].factorize()

*# All discrete features should now have integer dtypes (double-check this before using MI!)*

discrete\_features = X.dtypes == int

Scikit-learn has two mutual information metrics in its feature\_selection module: one for real-valued targets (mutual\_info\_regression) and one for categorical targets (mutual\_info\_classif). Our target, price, is real-valued. The next cell computes the MI scores for our features and wraps them up in a nice dataframe.

from sklearn.feature\_selection import mutual\_info\_regression

def make\_mi\_scores(X, y, discrete\_features):

mi\_scores = mutual\_info\_regression(X, y, discrete\_features=discrete\_features)

mi\_scores = pd.Series(mi\_scores, name="MI Scores", index=X.columns)

mi\_scores = mi\_scores.sort\_values(ascending=False)

return mi\_scores

mi\_scores = make\_mi\_scores(X, y, discrete\_features)

mi\_scores[::3] *# show a few features with their MI scores*

def plot\_mi\_scores(scores):

scores = scores.sort\_values(ascending=True)

width = np.arange(len(scores))

ticks = list(scores.index)

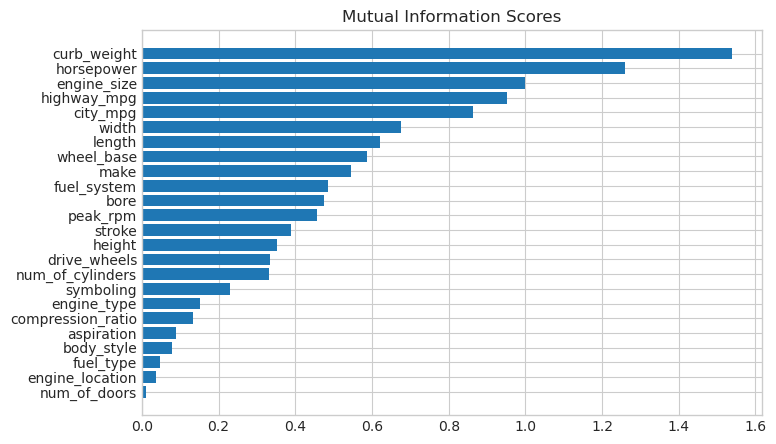
plt.barh(width, scores)

plt.yticks(width, ticks)

plt.title("Mutual Information Scores")

plt.figure(dpi=100, figsize=(8, 5))

plot\_mi\_scores(mi\_scores)



# Creating feature:

Tips on Discovering New Features

* Understand the features. Refer to your dataset's data documentation, if available.
* Research the problem domain to acquire domain knowledge. If your problem is predicting house prices, do some research on real-estate for instance. Wikipedia can be a good starting point, but books and journal articles will often have the best information.
* Study previous work. Solution write-ups from past Kaggle competitions are a great resource.
* Use data visualization. Visualization can reveal pathologies in the distribution of a feature or complicated relationships that could be simplified. Be sure to visualize your dataset as you work through the feature engineering process.

## Mathematical Transforms:

autos["stroke\_ratio"] = autos.stroke / autos.bore

autos[["stroke", "bore", "stroke\_ratio"]].head()

| stroke | bore | stroke\_ratio |
| --- | --- | --- |
| 0 | 2.68 | 3.47 | 0.772334 |
| 1 | 2.68 | 3.47 | 0.772334 |
| 2 | 3.47 | 2.68 | 1.294776 |
| 3 | 3.40 | 3.19 | 1.065831 |
| 4 | 3.40 | 3.19 | 1.065831 |

The more complicated a combination is, the more difficult it will be for a model to learn, like this formula for an engine's "displacement", a measure of its power:

autos["displacement"] = (

np.pi \* ((0.5 \* autos.bore) \*\* 2) \* autos.stroke \* autos.num\_of\_cylinders

)

Data visualization can suggest transformations, often a "reshaping" of a feature through powers or logarithms. The distribution of WindSpeed in US Accidents is highly skewed, for instance. In this case the logarithm is effective at normalizing it:

*# If the feature has 0.0 values, use np.log1p (log(1+x)) instead of np.log*

accidents["LogWindSpeed"] = accidents.WindSpeed.apply(np.log1p)

*# Plot a comparison*

fig, axs = plt.subplots(1, 2, figsize=(8, 4))

sns.kdeplot(accidents.WindSpeed, shade=True, ax=axs[0])

sns.kdeplot(accidents.LogWindSpeed, shade=True, ax=axs[1]);

A graph of a graph of a graph

Description automatically generated with medium confidence

## Counts

Features describing the presence or absence of something often come in sets, the set of risk factors for a disease, say. You can aggregate such features by creating a count.

roadway\_features = ["Amenity", "Bump", "Crossing", "GiveWay",

"Junction", "NoExit", "Railway", "Roundabout", "Station", "Stop",

"TrafficCalming", "TrafficSignal"]

accidents["RoadwayFeatures"] = accidents[roadway\_features].sum(axis=1)

accidents[roadway\_features + ["RoadwayFeatures"]].head(10)

You could also use a dataframe's built-in methods to create boolean values.

Many formulations lack one or more components (that is, the component has a value of 0). This will count how many components are in a formulation with the dataframe's built-in greater-than gt method:

components = [ "Cement", "BlastFurnaceSlag", "FlyAsh", "Water",

"Superplasticizer", "CoarseAggregate", "FineAggregate"]

concrete["Components"] = concrete[components].gt(0).sum(axis=1)

concrete[components + ["Components"]].head(10)

## Building-Up and Breaking-Down Features:

Often you'll have complex strings that can usefully be broken into simpler pieces. Some common examples:

* ID numbers: '123-45-6789'
* Phone numbers: '(999) 555-0123'
* Street addresses: '8241 Kaggle Ln., Goose City, NV'
* Internet addresses: 'http://www.kaggle.com
* Product codes: '0 36000 29145 2'
* Dates and times: 'Mon Sep 30 07:06:05 2013'

Features like these will often have some kind of structure that you can make use of. US phone numbers, for instance, have an area code (the '(999)' part) that tells you the location of the caller. As always, some research can pay off here.

The str accessor lets you apply string methods like split directly to columns. The Customer Lifetime Value dataset contains features describing customers of an insurance company. From the Policy feature, we could separate the Type from the Level of coverage:

customer[["Type", "Level"]] = ( *# Create two new features*

customer["Policy"] *# from the Policy feature*

.str *# through the string accessor*

.split(" ", expand=True) *# by splitting on " "*

*# and expanding the result into separate columns*

)

customer[["Policy", "Type", "Level"]].head(10)

You could also join simple features into a composed feature if you had reason to believe there was some interaction in the combination:

autos["make\_and\_style"] = autos["make"] + "\_" + autos["body\_style"]

autos[["make", "body\_style", "make\_and\_style"]].head()

## Group Transforms

Finally we have Group transforms, which aggregate information across multiple rows grouped by some category.

customer["AverageIncome"] = (

customer.groupby("State") *# for each state*

["Income"] *# select the income*

.transform("mean") *# and compute its mean*

)

customer[["State", "Income", "AverageIncome"]].head(10)

The mean function is a built-in dataframe method, which means we can pass it as a string to transform. Other handy methods include max, min, median, var, std, and count. Here's how you could calculate the frequency with which each state occurs in the dataset:

customer["StateFreq"] = (

customer.groupby("State")

["State"]

.transform("count")

/ customer.State.count()

)

customer[["State", "StateFreq"]].head(10)

You could use a transform like this to create a "frequency encoding" for a categorical feature.

If you're using training and validation splits, to preserve their independence, it's best to create a grouped feature using only the training set and then join it to the validation set. We can use the validation set's merge method after creating a unique set of values with drop\_duplicates on the training set:

Tips on Creating Features

It's good to keep in mind your model's own strengths and weaknesses when creating features. Here are some guidelines:

* Linear models learn sums and differences naturally, but can't learn anything more complex.
* Ratios seem to be difficult for most models to learn. Ratio combinations often lead to some easy performance gains.
* Linear models and neural nets generally do better with normalized features. Neural nets especially need features scaled to values not too far from 0. Tree-based models (like random forests and XGBoost) can sometimes benefit from normalization, but usually much less so.
* Tree models can learn to approximate almost any combination of features, but when a combination is especially important they can still benefit from having it explicitly created, especially when data is limited.
* Counts are especially helpful for tree models, since these models don't have a natural way of aggregating information across many features at once

If you've discovered an interaction effect between a numeric feature and a **categorical** feature, you might want to model it explicitly using a one-hot encoding, like so:

# One-hot encode Categorical feature, adding a column prefix "Cat"

X\_new = pd.get\_dummies(df.Categorical, prefix="Cat")

# Multiply row-by-row

X\_new = X\_new.mul(df.Continuous, axis=0)

# Join the new features to the feature set

X = X.join(X\_new)

# Clustering With K-Means:

Clustering simply means the assigning of data points to groups based upon how similar the points are to each other. A clustering algorithm makes "birds of a feather flock together," so to speak.

## Cluster Labels as a Feature:

Applied to a single real-valued feature, clustering acts like a traditional "binning" or "discretization" transform. On multiple features, it's like "multi-dimensional binning" (sometimes called vector quantization).

It's important to remember that this Cluster feature is categorical. Here, it's shown with a label encoding (that is, as a sequence of integers) as a typical clustering algorithm would produce; depending on your model, a one-hot encoding may be more appropriate.

## k-Means Clustering:

There are a great many clustering algorithms.

K-means clustering measures similarity using ordinary straight-line distance (Euclidean distance, in other words). It creates clusters by placing a number of points, called centroids, inside the feature-space. The "k" in "k-means" is how many centroids (that is, clusters) it creates. You define the k yourself.

The algorithm starts by randomly initializing some predefined number (n\_clusters) of centroids. It then iterates over these two operations:

1. assign points to the nearest cluster centroid
2. move each centroid to minimize the distance to its points

It iterates over these two steps until the centroids aren't moving anymore, or until some maximum number of iterations has passed (max\_iter).

It often happens that the initial random position of the centroids ends in a poor clustering. For this reason the algorithm repeats a number of times (n\_init) and returns the clustering that has the least total distance between each point and its centroid, the optimal clustering.

kmeans = KMeans(n\_clusters=6)

X["Cluster"] = kmeans.fit\_predict(X)

X["Cluster"] = X["Cluster"].astype("category")

X.head()

sns.relplot(

x="Longitude", y="Latitude", hue="Cluster", data=X, height=6,

);

check out the preprocessing module in scikit-learn for some of the rescaling methods it offers

The k-means algorithm is sensitive to scale. This means we need to be thoughtful about how and whether we rescale our features since we might get very different results depending on our choices. As a rule of thumb, if the features are already directly comparable (like a test result at different times), then you would not want to rescale. On the other hand, features that aren't on comparable scales (like height and weight) will usually benefit from rescaling.

# Principal Component Analysis:

Just like clustering is a partitioning of the dataset based on proximity, you could think of PCA as a partitioning of the variation in the data

Technical note: PCA is typically applied to standardized data. With standardized data "variation" means "correlation". With unstandardized data "variation" means "covariance". All data in this course will be standardized before applying PCA.

A graph of a diagram

Description automatically generated with medium confidence

The new features PCA constructs are actually just linear combinations (weighted sums) of the original features:

df["Size"] = 0.707 \* X["Height"] + 0.707 \* X["Diameter"]

df["Shape"] = 0.707 \* X["Height"] - 0.707 \* X["Diameter"]

These new features are called the principal components of the data

The weights themselves are called loadings.

There will be as many principal components as there are features in the original dataset: if we had used ten features instead of two, we would have ended up with ten components.

A component's loadings tell us what variation it expresses through signs and magnitudes:

| Features \ Components | Size (PC1) | Shape (PC2) |
| --- | --- | --- |
| Height | 0.707 | 0.707 |
| Diameter | 0.707 | -0.707 |
|  |  |  |

This table of loadings is telling us that in the Size component, Height and Diameter vary in the same direction (same sign), but in the Shape component they vary in opposite directions (opposite sign). In each component, the loadings are all of the same magnitude and so the features contribute equally in both.

## PCA for Feature Engineering

There are two ways you could use PCA for feature engineering.

1. The first way is to use it as a descriptive technique, Since the components tell you about the variation, you could compute the MI scores for the components and see what kind of variation is most predictive of your target. That could give you ideas for kinds of features to create -- a product of 'Height' and 'Diameter' if 'Size' is important
2. The second way is to use the components themselves as features, Because the components expose the variational structure of the data directly, they can often be more informative than the original features.

Here are some use-cases:

* **Dimensionality reduction**: When your features are highly redundant (multicollinear, specifically), PCA will partition out the redundancy into one or more near-zero variance components, which you can then drop since they will contain little or no information.
* **Anomaly detection**: Unusual variation, not apparent from the original features, will often show up in the low-variance components. These components could be highly informative in an anomaly or outlier detection task.
* **Noise reduction**: A collection of sensor readings will often share some common background noise. PCA can sometimes collect the (informative) signal into a smaller number of features while leaving the noise alone, thus boosting the signal-to-noise ratio.
* **Decorrelation**: Some ML algorithms struggle with highly-correlated features. PCA transforms correlated features into uncorrelated components, which could be easier for your algorithm to work with.

## PCA Best Practices

There are a few things to keep in mind when applying PCA:

* PCA only works with numeric features, like continuous quantities or counts.
* PCA is sensitive to scale. It's good practice to standardize your data before applying PCA, unless you know you have good reason not to.
* Consider removing or constraining outliers, since they can have an undue influence on the results.

EX:

# Each of these features also has a high MI score with the target

features = ["highway\_mpg", "engine\_size", "horsepower", "curb\_weight"]

X = df.copy()

y = X.pop('price')

X = X.loc[:, features]

*# Standardize*

X\_scaled = (X - X.mean(axis=0)) / X.std(axis=0)

Now we can fit scikit-learn's PCA estimator and create the principal components. You can see here the first few rows of the transformed dataset.

from sklearn.decomposition import PCA

*# Create principal components*

pca = PCA()

X\_pca = pca.fit\_transform(X\_scaled)

*# Convert to dataframe*

component\_names = [f"PC**{**i+1**}**" for i **in** range(X\_pca.shape[1])]

X\_pca = pd.DataFrame(X\_pca, columns=component\_names)

X\_pca.head()

After fitting, the PCA instance contains the loadings in its components\_ attribute. We'll wrap the loadings up in a dataframe.

loadings = pd.DataFrame(

pca.components\_.T, *# transpose the matrix of loadings*

columns=component\_names, *# so the columns are the principal components*

index=X.columns, *# and the rows are the original features*

)

loadings

*# Look at explained variance*

plot\_variance(pca);

mi\_scores = make\_mi\_scores(X\_pca, y, discrete\_features=False)

mi\_scores

PC1 1.013264

PC2 0.379156

PC3 0.306703

PC4 0.203329

Name: MI Scores, dtype: float64

*# Show dataframe sorted by PC3*

idx = X\_pca["PC3"].sort\_values(ascending=False).index

cols = ["make", "body\_style", "horsepower", "curb\_weight"]

df.loc[idx, cols]

df["sports\_or\_wagon"] = X.curb\_weight / X.horsepower

sns.regplot(x="sports\_or\_wagon", y='price', data=df, order=2);

**from** learntools.core **import** binder

binder.bind(globals())

**from** learntools.feature\_engineering\_new.ex5 **import** **\***

​

**import** matplotlib.pyplot **as** plt

**import** numpy **as** np

**import** pandas **as** pd

**import** seaborn **as** sns

**from** sklearn.decomposition **import** PCA

**from** sklearn.feature\_selection **import** mutual\_info\_regression

**from** sklearn.model\_selection **import** cross\_val\_score

**from** xgboost **import** XGBRegressor

​

*# Set Matplotlib defaults*

plt.style.use("seaborn-whitegrid")

plt.rc("figure", autolayout**=True**)

plt.rc(

"axes",

labelweight**=**"bold",

labelsize**=**"large",

titleweight**=**"bold",

titlesize**=**14,

titlepad**=**10,

)

​

​

**def** apply\_pca(X, standardize**=True**):

*# Standardize*

**if** standardize:

X **=** (X **-** X.mean(axis**=**0)) **/** X.std(axis**=**0)

*# Create principal components*

pca **=** PCA()

X\_pca **=** pca.fit\_transform(X)

*# Convert to dataframe*

component\_names **=** [f"PC{i**+**1}" **for** i **in** range(X\_pca.shape[1])]

X\_pca **=** pd.DataFrame(X\_pca, columns**=**component\_names)

*# Create loadings*

loadings **=** pd.DataFrame(

pca.components\_.T, *# transpose the matrix of loadings*

columns**=**component\_names, *# so the columns are the principal components*

index**=**X.columns, *# and the rows are the original features*

)

**return** pca, X\_pca, loadings

​

​

**def** plot\_variance(pca, width**=**8, dpi**=**100):

*# Create figure*

fig, axs **=** plt.subplots(1, 2)

n **=** pca.n\_components\_

grid **=** np.arange(1, n **+** 1)

*# Explained variance*

evr **=** pca.explained\_variance\_ratio\_

axs[0].bar(grid, evr)

axs[0].set(

xlabel**=**"Component", title**=**"% Explained Variance", ylim**=**(0.0, 1.0)

)

*# Cumulative Variance*

cv **=** np.cumsum(evr)

axs[1].plot(np.r\_[0, grid], np.r\_[0, cv], "o-")

axs[1].set(

xlabel**=**"Component", title**=**"% Cumulative Variance", ylim**=**(0.0, 1.0)

)

*# Set up figure*

fig.set(figwidth**=**8, dpi**=**100)

**return** axs

​

​

**def** make\_mi\_scores(X, y):

X **=** X.copy()

**for** colname **in** X.select\_dtypes(["object", "category"]):

X[colname], \_ **=** X[colname].factorize()

*# All discrete features should now have integer dtypes*

discrete\_features **=** [pd.api.types.is\_integer\_dtype(t) **for** t **in** X.dtypes]

mi\_scores **=** mutual\_info\_regression(X, y, discrete\_features**=**discrete\_features, random\_state**=**0)

mi\_scores **=** pd.Series(mi\_scores, name**=**"MI Scores", index**=**X.columns)

mi\_scores **=** mi\_scores.sort\_values(ascending**=False**)

**return** mi\_scores

​

​

**def** score\_dataset(X, y, model**=**XGBRegressor()):

*# Label encoding for categoricals*

**for** colname **in** X.select\_dtypes(["category", "object"]):

X[colname], \_ **=** X[colname].factorize()

*# Metric for Housing competition is RMSLE (Root Mean Squared Log Error)*

score **=** cross\_val\_score(

model, X, y, cv**=**5, scoring**=**"neg\_mean\_squared\_log\_error",

)

score **=** **-**1 **\*** score.mean()

score **=** np.sqrt(score)

**return** score

​

​

df **=** pd.read\_csv("../input/fe-course-data/ames.csv")

# Target Encoding:

A target encoding is any kind of encoding that replaces a feature's categories with some number derived from the target.

target encoding, is meant for categorical features, It's a method of encoding categories as numbers, like one-hot or label encoding, with the difference that it also uses the target to create the encoding. This makes it what we call a **supervised feature engineering** technique.

autos["make\_encoded"] = autos.groupby("make")["price"].transform("mean")

autos[["make", "price", "make\_encoded"]].head(10)

## Smoothing:

An encoding like this presents a couple of problems;

First are unknown categories. Target encodings create a special risk of overfitting, which means they need to be trained on an independent "encoding" split.

Second are rare categories. When a category only occurs a few times in the dataset, any statistics calculated on its group are unlikely to be very accurate.

A solution to these problems is to add smoothing.

The idea is to blend the in-category average with the overall average. Rare categories get less weight on their category average, while missing categories just get the overall average

In pseudocode:

*encoding = weight \* in\_category + (1 - weight) \* overall*

where weight is a value between 0 and 1 calculated from the category frequency.

An easy way to determine the value for weight is to compute an m-estimate:

*weight = n / (n + m)*

where n is the total number of times that category occurs in the data. The parameter m determines the "smoothing factor". Larger values of m put more weight on the overall estimate

When choosing a value for m, consider how noisy you expect the categories to be. Does the price of a vehicle vary a great deal within each make? Would you need a lot of data to get good estimates? If so, it could be better to choose a larger value for m; if the average price for each make were relatively stable, a smaller value could be okay.

### Use Cases for Target Encoding

Target encoding is great for:

* High-cardinality features: A feature with a large number of categories can be troublesome to encode: a one-hot encoding would generate too many features and alternatives, like a label encoding, might not be appropriate for that feature. A target encoding derives numbers for the categories using the feature's most important property: its relationship with the target.
* Domain-motivated features: From prior experience, you might suspect that a categorical feature should be important even if it scored poorly with a feature metric. A target encoding can help reveal a feature's true informativeness.

We'll start by creating a 25% split to train the target encoder.

X = df.copy()

y = X.pop('Rating')

X\_encode = X.sample(frac=0.25)

y\_encode = y[X\_encode.index]

X\_pretrain = X.drop(X\_encode.index)

y\_train = y[X\_pretrain.index]

from category\_encoders import MEstimateEncoder

*# Create the encoder instance. Choose m to control noise.*

encoder = MEstimateEncoder(cols=["Zipcode"], m=5.0)

*# Fit the encoder on the encoding split.*

encoder.fit(X\_encode, y\_encode)

*# Encode the Zipcode column to create the final training data*

X\_train = encoder.transform(X\_pretrain)

plt.figure(dpi=90)

ax = sns.distplot(y, kde=False, norm\_hist=True)

ax = sns.kdeplot(X\_train.Zipcode, color='r', ax=ax)

ax.set\_xlabel("Rating")

ax.legend(labels=['Zipcode', 'Rating']);

A graph with a red line

Description automatically generated

The distribution of the encoded Zipcode feature roughly follows the distribution of the actual ratings, meaning that movie-watchers differed enough in their ratings from zipcode to zipcode that our target encoding was able to capture useful information.

df.select\_dtypes(["object"]).nunique()

df["SaleType"].value\_counts()