1. What is Feature Engineering

1. Learning What? :

- determine which features are the most important with mutual information.
- invent new features in several real-world problem domains.
- encode high-cardinalir categoriew with a target encoding.
- create segmentation features with k-means clustering
- decompose a dataset's variation into feature with principal component analysis.

2. The Goal of Feature Engineering :

The goal of feature engineering is simply to make your data better suited to the problem at hand.

Consider "apparent temperature" measures like the heat index and the wind chill. These quantities attempt to measure the perceived temperature to humans based on air temperature, humidty, and wind speed, things which we can measure directly. You could think of an apparent temperature as the result of a kind of feature engieerning, an attemp to make the observed data more relevant to what we actually care about.

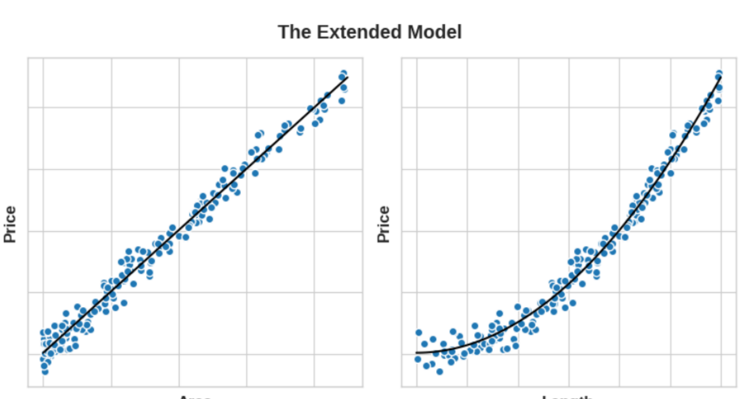
- improve a model's predictive performance
- reduce computational or data needs
- improve interpretability of the results

3. A Guiding Principle of Feature Engineering :

For a feature to be useful, it must have a relationship to the target that your model is able to learn. Linear model, for instance, are only able to learn linear relationship. So, when using a linear model, your goal is to transform the features to make their relationship to the target linear.

The key idea here is that a transformation you apply to a features becomes in essence a part of the model itself. Say you were trying to predict the Price of square plots of land from the Length of one side. Fitting a linear model directly at Length gives poor results.

If we square the Length features to get "Area", however, we create a linear relationship. Adding Area to the feature set means this linear model can now fit a parabola. Squaring a feature, in other words, gave the linear model the ability to fit squared features.



Left: *The fit to Area is much better.* ***Right:*** *Which makes the fit to Length better as well.*

4. Example - Concrete Formulations

To illustrate these ideas we'll see how adding a few synthetic features to a dataset can improve the predictive performance of a randomforest model.

The Concrete datasets contains a variety of concrete formulations and resulting products's compressive strength, which is a measure of how much load that kind of concrete can bear. The task for dataset is to predict a concrete's compressive strength given its formulation.

	Cement	BlastFurnaceSlag	FlyAsh	Water	Superplasticizer	CoarseAggregate	FineAggregate	Age	CompressiveStrength
0	540.0	0.0	0.0	162.0	2.5	1040.0	676.0	28	79.99
1	540.0	0.0	0.0	162.0	2.5	1055.0	676.0	28	61.89
2	332.5	142.5	0.0	228.0	0.0	932.0	594.0	270	40.27
3	332.5	142.5	0.0	228.0	0.0	932.0	594.0	365	41.05
4	198.6	132.4	0.0	192.0	0.0	978.4	825.5	360	44.30

You can see here the various ingredients going into each variety of concrete. We'll see in a moment how adding some additional synthetic features derived from these can help a model to learn important relationships among them.

We'll first establish a baseline by training the model on the un-augmented dataset. This will help us determine whether our new features are actually useful.

Establishing baselines like this is good practice at the start of the feature engineering process. A baseline score can help you decide wheter your new features are worth keeping, or whether you should discard them and possibly try something else.

```
X = df.copy()
y = X.pop("CompressiveStrength")

# Train and score baseline model
baseline = RandomForestRegressor(criterion="mae", 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}")
```

If you ever cook at home, you might know that the ratio of ingredients in a recipe is usually a better predictor of how the recipe turns out than thier absolute amounts. We might reason then that ratios of the features above would be a good predictor of CompressiveStrenght.

```
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="mae", 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}")
```

2. Mutual Infromation

First encountering a new datasets can sometimes feel overwhelming. You might be preseneted with hundreds or thousands of features without even a description to go by. Where do you even begin?

A great first step is to construct a ranking with a feature utility metric, a function measuring associations between a feature and the target. Then you can choose a smaller set of the most useful feature and the target. Then you can choose a smaller set of the most useful features to develop initally and have more confidence that your time will be well spent.

The metric we'll use is called "mutual information". Mutual information is a lot like correlation in that it mesures a relationship between two quantities. The advantage of mututal information is that it can detect any kind of relationship, while correlation only detects linear relationships.

Mutual information is a great general-purpose metric and especially useful at the start of feature development when you might not know what model you'd like to use yet. It is :

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

1. Mutual information and what it measures

Mutual information descirbes relationships in terms of uncertainty. The mutual information between two quantities is a meausre of the extent to which knowledge of one qualitiy reduces uncertainty about the other. If you knew the value of a feature, how much more confident would you be about the target?

2. Interpreting Mutual information Scores

The least possible mutual information between quantities is 0.0. When MI is zero, the quantities are independent : neither can tell you anything about the other. Coversly, in theroy there's no upper bound to what MI can be. In practice though values above 2.0 or so are uncommon.

Here are some things to remember when applying mutual information :

- MI can help you to understand the relative potential of a features as a predictor of the target, considered by itself.
- It's possible for a feature to be very informative when interacting with other features, but not so informative all alone. MI can't detect interactiosn between features. It is a univariate metric.
- The actural usefulness of a feature depends on the model you use it with. A feature is only useful to the extent that its relationship with the target is one your model can learn. Jsut becuase a feature has high MI score doesn't mean you model will be able to do anything with that information. you may need to transform the feature first to expose the association.

3. What need to focus

1. The scikit-learn algorithm for MI treats discrete features differently from continuous features. Consequently, you need to tell it which are which. As a rule of thumb, anything that must have a flaot dtypes is not discrete. Categorical can be treated by giving them a label encoding(or ordinaly encoding).
2. Scikit-learn has two mutual information metrics in its feature_selection module: one for real-valued targets(mutual_info_regresion) and one for categorical targets(mutual_info_classif).

In [3]:
```
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

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")
```

Data visulaization is a great addition to your feature-engineering toolbox. Along with utility metrics like mutual information, visualization like these can help you discover important relationships in your data.

3. Creating Features

Once you've identified a set of features with some potential, it's time to start developing them. In this lesson, you'll learn a number of common transformation you can do entirely in Pandas.

1. Tips on Discovering New Features

- Understand the features. Refer to your dataset's data documentation, if available.
- Reserach 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 competiotions are a greeat resource.
- Use data visualization. Visualization can reveal pathologies in the distribution of a feature or complicated relationships that colud be simplified. Be sure to visualize your dataset as you work through the feature engineering process.

2. Mathematical Transforms

Relationships among numerical features are often expressed through mathmatical formulas, which you'll frequently come across as part of your domain researhc. In pandas, you can apply arithmetic operations to columns just as if they were ordinary numbers. Data visualization can suggest transformations, often a "reshaping" of a feature trhough powers or logarithms.

3. Counts

Features describing the presence or absence of something oftten come in sets, the set of risk factor for disease, say. You can aggregate such features by creating a count. These features will be binary(1 for Present, 0 for Absent) or boolen (True or False). In Python, booleans can be added up jus as if they were integers.

In traffic accidents are several features indicating wheter some roadway object was near the accident. This will create a count of the total number of roadway features nearby using the sum method :

You could also use a dataframe's built-in methods to create boolean values. In the Concrete dataset are the amounts of components in a concrete formulation. Many formulations lack one or more componets. This wil count how many components are in a formulation with the dataframe's built-in greater-than gt method :

```
# 2) Mathematical Tranform

autos["displacement"] = (
    np.pi * ((0.5 * autos.bore) ** 2) * autos.stroke * autos.num_of_cylinders
)

# 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]);

# 3) Counts

roadway_features = ["Amenity", "Bump", "Crossing", "GiveWay",
    "Junction", "NoExit", "Railway", "Roundabout", "Station", "Stop",
    "TrafficCalming", "TrafficSignal"]
accidents["RoadwayFeatures"] = accidents[roadway_features].sum(axis=1)
```

4. 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 : '123-456-789'
- Phone numbers : '(999) 555-0123'
- Stret address : '8241 Kaggle Ln., Goose City, NV'

Features like these will often have soem kind of structure that you cna make use of. US phone numbers, for instance that tells you the location of the caller, As always, some research can pay off here.

The str accesor lets you apply string method like split directly to columns.

5. Group Transforms

Finally we have group transforms, which aggregate information across multiple rows grouped by some category. With a group transform you can create features like : "the average income of a person's state of residence", or "the proportion of movies released on a weekday, by gnere." If you had discovered a categroy interaction, a group transform over that category could be something good to investigate.

Using an aggregation fucntion, a group transform combines two features : a categorical features that provides the grouping and another feature whose value you wish to aggregate. For an "average income by state", you would choose State for grouping feature mean for the aggregation function and Income for the aggregated feature. To compute this in Pandas, we use the groupby and transform method :

```
customer["AverageIncome"] = (
customer["AverageIncome"] = (
    customer.groupby("State")  # for each state
    ["Income"]                 # select the income
    .transform("mean")         # and compute its mean
)

# Frequency of categorical value

customer["StateFreq"] = customer.groupby("State")["State"].transform("count") / customer.State.count()
```

6. Tips on Creating Features

- Linear models learn sums and differneces naturally, but can't learn anything more complex.
- Ratios seem to be difficult for most model 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 valud not too far from 0. Tree-based models (like random forest 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 expecially 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.

3. Clustering with K-Means

This lesson and the next make use of what are known unsupervised learning algorithms. Unsupervised algorithms don't make use of a target; instead, thier purpose is to learn some property of the data, to represent the structure of the features in a certain way. In the context of feature engineering for predictions, you could think of an unsupervised algorightms as a "feature discovery" technique.

Cluertering simply means the assigning of data points to group based upon how similar the points are to each other. A clustering algorithm makes "birds of feature flock together", so to speak.

When used for feature engineering, we could attempt to discover groups of customers representing a market segment, for instance, or geographic areas that share similar weather patteerns. Adding a feature of clustering labels can help machine learning models untangle complicated relationships of space or proximity.

1. 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".

It's important to remember that this Cluster feature is categorical. Here, it's shonw with a lbel encoding as a typical clustering algorithm would produce; depending on your model, a ohe-hot encoding may be more appropriate.

The motivating idea of adding cluster labels is that the clusters will break up complicated relationships across features into simpler chunks. Our model can then just learn the simpler chunks one-by-one instead having to learn the complicated whole all at once. It's a "divide and conquer" strategy.

2. K-Means Clustering

There are a great many clustering algorithms. They differ primarily in how they measure "similarity" or "proximity" and in what kinds of features they work with. The algorithm we'll use, k-means, is intuitive and easy to apply in a feature engineering context. Depending on your application another algorithm might be more appropriate.

K-means clustering measures similarity using ordinary stright-line distance (Euclidean distance, in other words). It creates clusters by placing a number of points, called centroids, inside the feature-space. Each point in the dataset is assigned to the cluster of whichever centroid it's closest to. The "k" in "k-means" is how many centroids (that is, clusters) it creates. You define k your self.

You could imagine each centroid capturing points through a sequence of radiating circles. When sets of circles from competing centroids overlap they form a line. The result is what's called a voronoi tessallation. The tessalation shows you to what clusters future data will be assigned; the tessallation is essentially what k-means learns from its training data.

Let's review how the k-means algorithm learns the clusters and what that means for feature engineering. We'll focus on three parameters from scikit-learn's implementation : n_cluster, max_iter, and n_init.

It's a simple two-step process. The algorithm starts by randomly initailizing some predefined number (n_clusters) of centroids. It then iterates over these two operation :

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 initail random position of the centroids ends in a poor clustering. For this reason the algorithm repeats a number of times(n_init) and return the clustering that has the least total distance between each point and its centroid, the optimal clustering.

You many need to increase the max_iter for a large number of clusters or n_init for a complex dataset. Ordinarily though the only parameter you'll need to choose your self is n_cluster (k, that is). The best partitioning for a set of features depends on the model you're using and what you're trying to predict, so it's best to tune it like any hyperparametr (through cross-validation, say).

```
# Example of making new feature using K-means

kmeans = KMeans(n_clusters = 6)

X['Cluster'] = kmeans.fit_predict(X)
X['Clutser'] = X['Cluster'].astype('category')
```

4. Principal Component Analysis

1. Introudcution

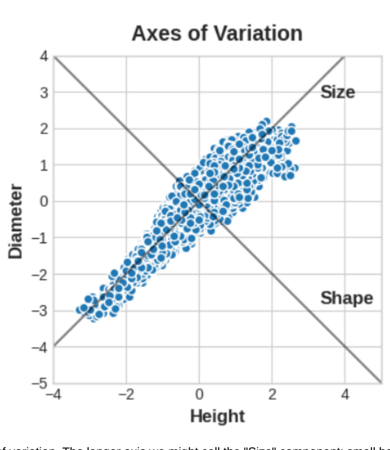
In the previous lesson we looked at our first model-based method for feature engineering : clustering. In this lesson we look at our next: principal component analysis(PCA). Just like clustering is a partitioning of the dataset based on proximity, you could think of PCA as partitioning of the variation in the data. PCA is a great tool to help you discover important relationships in the data and can also be used to creat more informative features.

PCA is typically applied to standardized data. With standarized data "variation" means "correlation". With unstandarzed data "variation" means "covariance". All data in this course will be standardized before applying PCA.

2. Principal Component Analysis

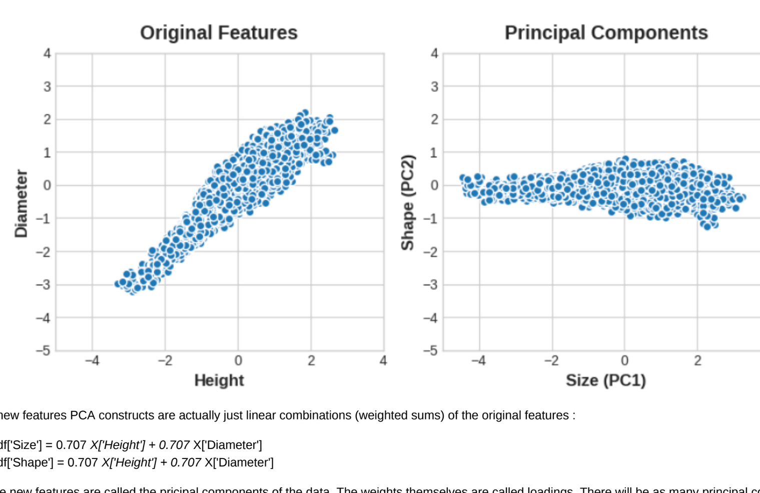
In the Abalone dataset are physical measurements taken from several thousand Tasmanian abalon. (An abalone is a sea creature much like a clam or an oyster.) We'll just look at a couple feature for now: the 'Height' and 'Diameter' of their shells.

You could imagine that within this data are "axes of variation" that describe the way the ablaon tend to differ from one another. Pictorially, these axes appear as perpendicular lines running along the natural dimensions of the data, one axis for each original features.



Often, we can gives names to these axes of variation. The longer axis we might call the "Size" component: small height and small diameter (loer left) contrasted with large height and large diameter (upper right). The shorter axis we might call the "Shape" component: small height and larget diameter (flat shape) contrasted with large height and small diameter (round shape).

Notice that instead of describing abalones by thier 'Height' and 'Diameter', we could just as well describe them by their 'Size' and 'Shape'. This, in fact, is the whole idea of PCA: instead of describing the data with the original features, we describes it with its axes of variation. The axes of variation become the new features.



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

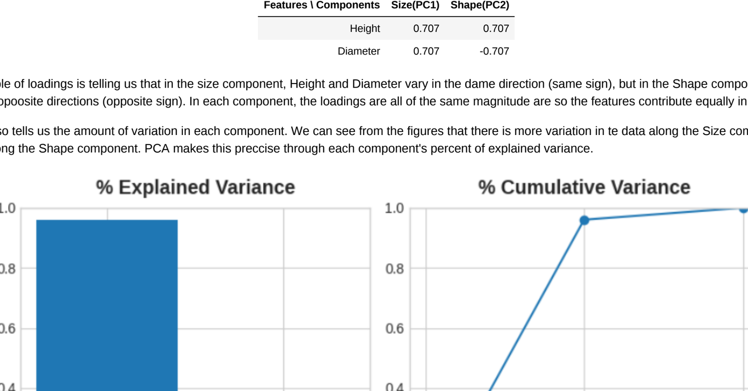
- df['Size'] = 0.707 *X['Height'] + 0.707* X['Diameter']
- df['Shape'] = 0.707 *X['Height'] + 0.707* X['Diameter']

These new features are called the pricipal 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.

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 dame direction (same sign), but in the Shape component the vary in opposite directions (opposite sign). In each component, the loadings are all of the same magnitude are so the features contribute equally in both.

PCA also tells us the amount of variation in each component. We can see from the figures that there is more variation in te data along the Size component than along the Shape component. PCA makes this preccise through each component's percent of explained variance.



The size component captures the majority of the variation between Height and Diameter. It's important to remeber, however, that the amount of variance in a component doesn't necessarily correspond to how good it is a predictor: it depends on what you're trying to predict.

3. PCA for Feature Engineering

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

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 feature to create -- a product of 'Height' and 'Diameter' if 'Size' is important, say, or a ratio of 'Heigh' and 'Diameter' if Shape is important. You could even try clustering on one or more of the high-scoring components.

The second way is to use the components themselves as features. Because the components expose the variation 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 (multicolinear, 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 sometinmes collect the (informative) signal into a small 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 you algorithm to work with.

PCA basically gives you direct access to the correlational structure of you data. You'll no doubt come up with application of you own!

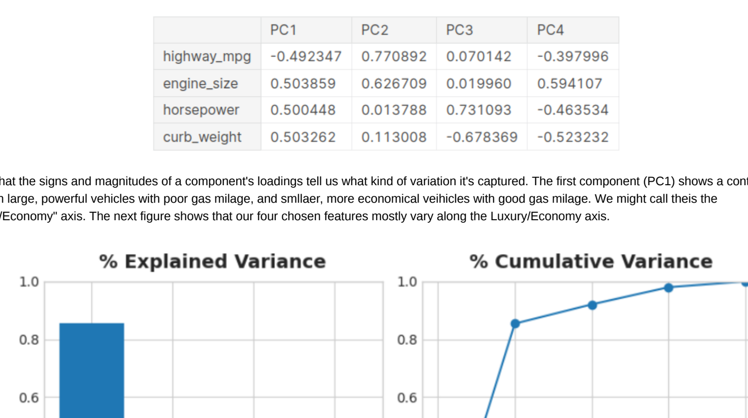
- PCA Best Practices
  - PCA only works with numeric features, like continuous quantities or counts.
  - PCA is sensitive to scale. it's good practice to standardize your data before applying PCA, unless you know you have good reason not to
  - Consider removing or constraining outlier, since they can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

```
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)

# Create principal components

from sklearn.decomposition import PCA

pca = PCA()
X_pca = pca.fit_transform(X_scaled)

# Convert to dataframe

component_names = [f"PC{i+1}" for i in range(X_pcc.shape[1])]
X_pca = pd.DataFrame(X_pca, columns = component_names)
```

After fitting, the PCA instance contains the loadings in its components_ attribute.

	PC1	PC2	PC3	PC4
highway_mpg	-0.492347	0.770892	0.070142	-0.397996
engine_size	0.503859	0.626709	0.019960	0.594107
horsepower	0.500448	0.013788	0.731093	-0.463534
curb_weight	0.503262	0.113008	-0.678369	-0.523232

Recall that the signs and magnitudes of a component's loadings tell us what kind of variation it's captured. The first component (PC1) shows a contrast between large, powerful vehicles with poor gas milage, and smllaer, more economical veihicles with good gas milage. We might call theis the "Luxury/Economy" axis. The next figure shows that our four chosen features mostly vary along the Luxury/Economy axis.



Let's also look at the MI scores of the components. Not surprisingly, PC1 is highly informattive, though the remaining comonents, despite thier small variance, still have significant relationship with price. Examining those components could be worthwile to find relationships not captured by the main Luxury/Economy axis.

```
PC1    1.013800
PC2    0.379440
PC3    0.306502
PC4    0.204447
Name: MI Scores, dtype: float64
```

5. Target Encoding

1. Introduction

Most of the techniques we've seen in this course have been for numerical features. The technique we'll look at this lesson, target encoding, is instead meant for categorical features. It's a method of encoding categories as numbers, like one-hot or label eocnding, with the dicfference that it also uses the target to creat the encoding. This makes it what we call a supervised features engineering technique.

2. Target Encoding

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

A simple and effective version is to apply a group aggregation from Lesson 3, like the mean. Using the Automobiles dataset, this computes the average price of each vehicle's make :

```
autos['make_encoded'] = autos.groupby('make')['price'].transform('mean')

autos[['make', 'price', 'make_encoded']].head(10)
```

This kind of target encoding is sometimes called a mean encoding. Applied to a binary target, it's also called bin counting. (Other names you might come across include : likelihood encoding, impact encoding, and leave-one-out encoding.)

3. Smoothing

An encoding like this presents a couple of problems, hoever. First are unknown categories. Target encodings create a special risk of overfitting, which means they need to be trained on an independent "encoding" split. When you join the encoding to future splits, Pandas will fill in missing values for any categories not present in the encoding split. These missing values you would have to impute somehow.

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 unaccurate. In the Automobiles dataset, the mercurcy make only occurs once. The "mean" price we calculated is just the price of that one vehicle, which might not be very representative of any Mercuries we might see in the future. Target encoding rare categories can make overfitting more likely.

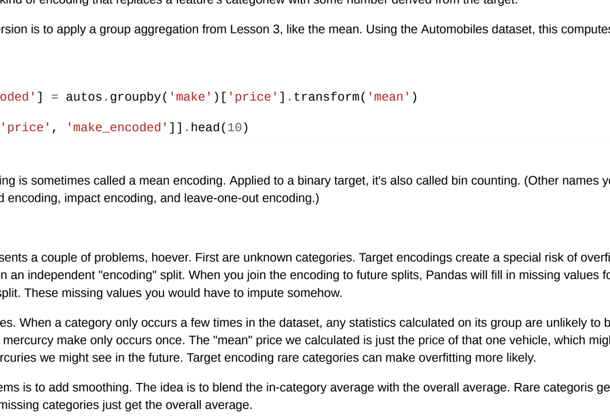
A solution to these problems is to add smoothing. The idea is to blend the in-category average with the overall average. Rare categoris get less weight on thier category average, while missing categories just get the overall average.

- 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 "smooting factor". Larger value of m put more weight on the overall estimate.



In the Automobiles dataset there are three cars with the make vhevorolet. If you chose m = 2.0, then the chevrolet category would be encoded with 60% of the average Chevrolet price plus 40% of the overall average price. When choosing a value of 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 colud be okay

- Use cases for Target Encoding
  - 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 label encoding, might not be appropriate for that features. A target encoding derives numbers for the categories using the feature's most imortant property: its relationship with the target.
  - Domain-motivaed features : From prior experience, you might suspect that a categorical features should be important even if it scored poorly with a feature metric. A target encoding can help reveal a feature's true informativeness

```
# Split datasets

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)
```