**Kaggle: Intermediate Machine Learning**

**General Approach of ML**

1. Understand ML object & data description

2. Read training/test data

3. Clean data

- Drop missing values or null values or

- Replace missing values or null values with mean of the column/meaningful values

4. Separate target from predictors

5 Divide data into training and validation subsets

6. Preprocess data

7. Generate ML model & fit the ML model to training data

8. Evaluate the ML model with error measure

**Categorical Variables**

approach1: drop columns with categorical data

approach2: ordinal encoding

approach3: one-hot encoding

**Pipelines**

|  |
| --- |
| Step1: Define Preprocessing Steps  from sklearn.compose import ColumnTransformer  from sklearn.pipeline import Pipeline  from sklearn.impute import SimpleImputer  from sklearn.preprocessing import OneHotEncoder  *# Preprocessing for numerical data*  numerical\_transformer = SimpleImputer(strategy='constant')  *# Preprocessing for categorical data*  categorical\_transformer = Pipeline(steps=[  ('imputer', SimpleImputer(strategy='most\_frequent')),  ('onehot', OneHotEncoder(handle\_unknown='ignore'))  ])  *# Bundle preprocessing for numerical and categorical data*  preprocessor = ColumnTransformer(  transformers=[  ('num', numerical\_transformer, numerical\_cols),  ('cat', categorical\_transformer, categorical\_cols)  ])  Step2: Define model  from sklearn.ensemble import RandomForestRegressor  model = RandomForestRegressor(n\_estimators=100, random\_state=0)  Step3: Create and Evaluate the Pipeline  from sklearn.metrics import mean\_absolute\_error  *# Bundle preprocessing and modeling code in a pipeline*  my\_pipeline = Pipeline(steps=[('preprocessor', preprocessor),  ('model', model)  ])  *# Preprocessing of training data, fit model*  my\_pipeline.fit(X\_train, y\_train)  *# Preprocessing of validation data, get predictions*  preds = my\_pipeline.predict(X\_valid)  *# Evaluate the model*  score = mean\_absolute\_error(y\_valid, preds)  print('MAE:', score) |

**Cross-Validation**

the drawbacks of data-driven way measuring model quality with a validation set

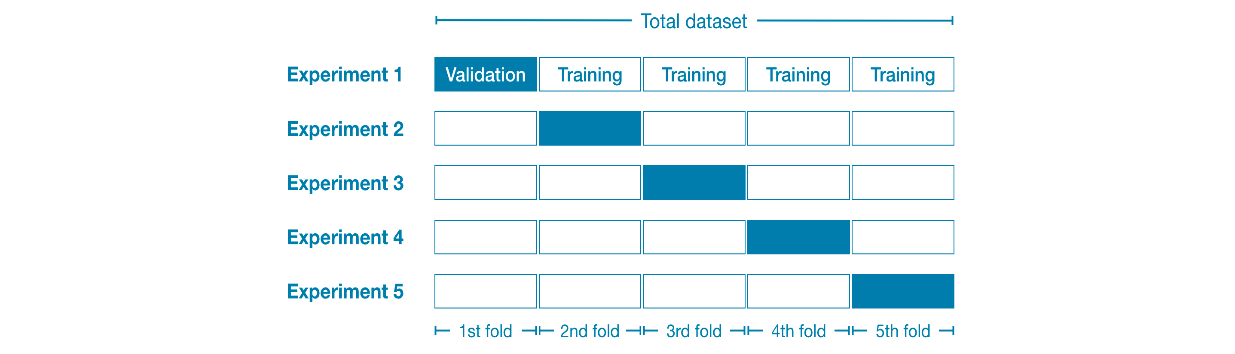
* In general, the larger the validation set, the less randomness (aka "noise") there is in our measure of model quality, and the more reliable it will be. Unfortunately, we can only get a large validation set by removing rows from our training data, and smaller training datasets mean worse models!

Solution for the drawbacks of data-driven way measuring model quality with a validation set

Cross-Validation for better measurements of model performance

What is cross-validation?

* run our modeling process on different subsets of the data to get multiple measures of model quality
* For example, we could begin by dividing the data into 5 pieces, each 20% of the full dataset. In this case, we say that we have broken the data into 5 "**folds**".



Then, we run one experiment for each fold:

* In **Experiment 1**, we use the first fold as a validation (or holdout) set and everything else as training data. This gives us a measure of model quality based on a 20% holdout set.
* In **Experiment 2**, we hold out data from the second fold (and use everything except the second fold for training the model). The holdout set is then used to get a second estimate of model quality.
* We repeat this process, using every fold once as the holdout set. Putting this together, 100% of the data is used as holdout at some point, and we end up with a measure of model quality that is based on all of the rows in the dataset (even if we don't use all rows simultaneously).

When should you use cross-validation?

Cross-validation gives a more accurate measure of model quality, which is especially important if you are making a lot of modeling decisions. However, it can take longer to run, because it estimates multiple models (one for each fold).

So, given these tradeoffs, when should you use each approach?

* *For small datasets*, where extra computational burden isn't a big deal, you should run cross-validation.
* *For larger datasets*, a single validation set is sufficient. Your code will run faster, and you may have enough data that there's little need to re-use some of it for holdout.

There's no simple threshold for what constitutes a large vs. small dataset. But if your model takes a couple minutes or less to run, it's probably worth switching to cross-validation.

Alternatively, you can run cross-validation and see if the scores for each experiment seem close. If each experiment yields the same results, a single validation set is probably sufficient.

\* While it's possible to do cross-validation without pipelines, it is quite difficult! Using a pipeline will make the code remarkably straightforward

|  |
| --- |
| from sklearn.model\_selection import cross\_val\_score  *# Multiply by -1 since sklearn calculates \*negative\* MAE*  scores = -1 \* cross\_val\_score(my\_pipeline, X, y,  cv=5,  scoring='neg\_mean\_absolute\_error') |

If you'd like to learn more about [hyperparameter optimization](https://en.wikipedia.org/wiki/Hyperparameter_optimization), you're encouraged to start with **grid search**, which is a straightforward method for determining the best combination of parameters for a machine learning model. Thankfully, scikit-learn also contains a built-in function [GridSearchCV()](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html" \t "_blank) that can make your grid search code very efficient!

**XGBoost(Extreme Gradient Boosting)**

The most accurate modeling technique for structured data.

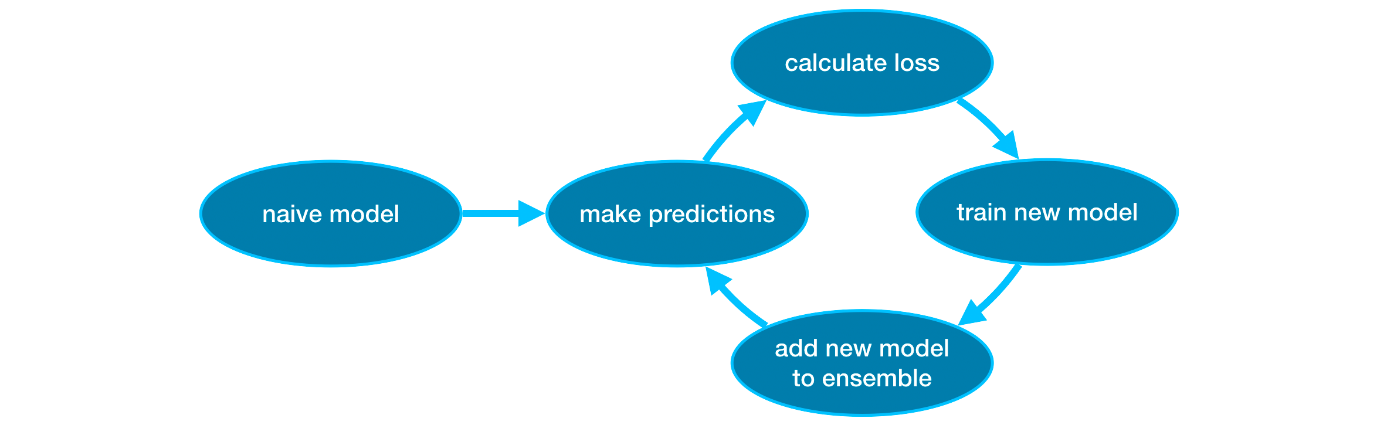
Leading software library for working with standard tabular data (the type of data you store in Pandas DataFrames, as opposed to more exotic types of data like images and videos). With careful parameter tuning, you can train highly accurate models.

Ensemble method that goes through cycles to iteratively add models into an ensemble

It begins by initializing the ensemble with a single model, whose predictions can be pretty naive. (Even if its predictions are wildly inaccurate, subsequent additions to the ensemble will address those errors.)

Then, we start the cycle:

* First, we use the current ensemble to generate predictions for each observation in the dataset. To make a prediction, we add the predictions from all models in the ensemble.
* These predictions are used to calculate a loss function (like [mean squared error](https://en.wikipedia.org/wiki/Mean_squared_error), for instance).
* Then, we use the loss function to fit a new model that will be added to the ensemble. Specifically, we determine model parameters so that adding this new model to the ensemble will reduce the loss. (*Side note: The "gradient" in "gradient boosting" refers to the fact that we'll use*[*gradient descent*](https://en.wikipedia.org/wiki/Gradient_descent)*on the loss function to determine the parameters in this new model.*)
* Finally, we add the new model to ensemble, and ...
* ... repeat!



Parameter tuning

the parameters that can dramatically affect accuracy and training speed

* n\_estimators

n\_estimators specifies how many times to go through the modeling cycle described above. It is equal to the number of models that we include in the ensemble.

* Too *low* a value causes *underfitting*, which leads to inaccurate predictions on both training data and test data.
* Too *high* a value causes *overfitting*, which causes accurate predictions on training data, but inaccurate predictions on test data (*which is what we care about*).

Typical values range from 100-1000, though this depends a lot on the learning\_rate parameter discussed below.

* early\_stopping\_rounds

early\_stopping\_rounds offers a way to automatically find the ideal value for n\_estimators. Early stopping causes the model to stop iterating when the validation score stops improving, even if we aren't at the hard stop for n\_estimators. It's smart to set a high value for n\_estimators and then use early\_stopping\_rounds to find the optimal time to stop iterating.

When using early\_stopping\_rounds, you also need to set aside some data for calculating the validation scores - this is done by setting the eval\_set parameter.

If you later want to fit a model with all of your data, set n\_estimators to whatever value you found to be optimal when run with early stopping.

* learning\_rate

Instead of getting predictions by simply adding up the predictions from each component model, we can multiply the predictions from each model by a small number (known as the **learning rate**) before adding them in.

This means each tree we add to the ensemble helps us less. So, we can set a higher value for n\_estimators without overfitting. If we use early stopping, the appropriate number of trees will be determined automatically.

In general, a small learning rate and large number of estimators will yield more accurate XGBoost models, though it will also take the model longer to train since it does more iterations through the cycle. As default, XGBoost sets learning\_rate=0.1.

* n\_jobs

On larger datasets where runtime is a consideration, you can use parallelism to build your models faster. It's common to set the parameter n\_jobs equal to the number of cores on your machine. On smaller datasets, this won't help.

The resulting model won't be any better, so micro-optimizing for fitting time is typically nothing but a distraction. But, it's useful in large datasets where you would otherwise spend a long time waiting during the fit command.

**Data Leakage**

**Data leakage** (or **leakage**) happens when your training data contains information about the target, but similar data will not be available when the model is used for prediction. This leads to high performance on the training set (and possibly even the validation data), but the model will perform poorly in production.

In other words, leakage causes a model to look accurate until you start making decisions with the model, and then the model becomes very inaccurate.

There are two main types of leakage: **target leakage** and **train-test contamination.**

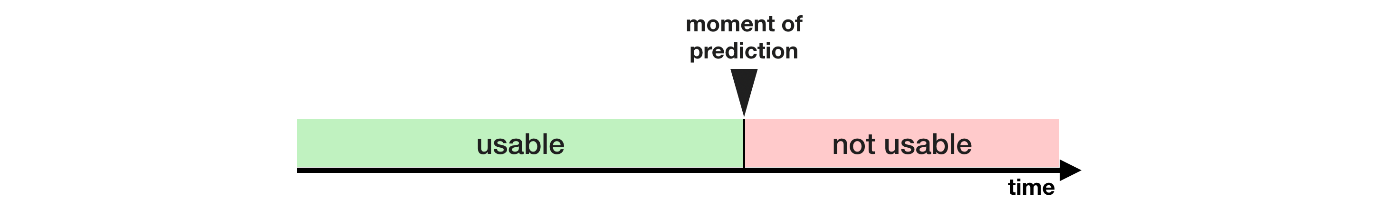
### Target leakage

* **Target leakage** occurs when your predictors include data that will not be available at the time you make predictions. It is important to think about target leakage in terms of the timing or chronological order that data becomes available, not merely whether a feature helps make good predictions.
* **Example:** Imagine you want to predict who will get sick with pneumonia. The top few rows of your raw data look like this

| got\_pneumonia | age | weight | male | took\_antibiotic\_medicine | ... |
| --- | --- | --- | --- | --- | --- |
| False | 65 | 100 | False | False | ... |
| False | 72 | 130 | True | False | ... |
| True | 58 | 100 | False | True | ... |

People take antibiotic medicines after getting pneumonia in order to recover. The raw data shows a strong relationship between those columns, but took\_antibiotic\_medicine is frequently changed after the value for got\_pneumonia is determined. This is target leakage.

* To prevent this type of data leakage, any variable updated (or created) after the target value is realized should be excluded.



Train-Test Contamination

* A different type of leak occurs when you aren't careful to distinguish training data from validation data.
* Recall that validation is meant to be a measure of how the model does on data that it hasn't considered before. You can corrupt this process in subtle ways if the validation data affects the preprocessing behavior. This is sometimes called **train-test contamination**.
* If your validation is based on a simple train-test split, exclude the validation data from any type of fitting, including the fitting of preprocessing steps. This is easier if you use scikit-learn pipelines. When using cross-validation, it's even more critical that you do your preprocessing inside the pipeline!

Data leakage can occur at different stages of the machine learning process:

1. **Training Phase:** If information from the test set or future data is inadvertently included in the training data, the model may learn patterns that do not generalize well. For example, using features that would not be available at the time of prediction can lead to overfitting.
2. **Feature Selection:** Leakage can occur when features are selected based on information from the entire dataset, including the test set. This can result in the model learning patterns that do not exist in new, unseen data.
3. **Preprocessing:** Inappropriate data preprocessing steps that involve information from the test set or that rely on knowledge of the target variable can introduce leakage. For instance, scaling features based on statistics computed from the entire dataset can lead to data leakage.

To avoid data leakage in machine learning, it's important to follow best practices:

1. **Split Data Properly:** Divide the dataset into training and testing sets before any preprocessing or feature engineering. Ensure that the test set is not used in any way during model training.
2. **Cross-Validation:** Use techniques like k-fold cross-validation to assess model performance without leakage. This involves splitting the data into multiple folds and training the model on different subsets while evaluating on the remaining folds.
3. **Feature Engineering:** Be cautious when creating new features or transforming existing ones. Ensure that these processes are based solely on information available in the training set.
4. **Temporal Data Handling:** If dealing with time-series data, ensure that the training set includes only information available up to a certain point in time, and the test set includes data from later periods.
5. **Strict Validation Procedures:** When tuning hyperparameters or selecting models, use a separate validation set that is not part of the training set.

Data leakage in machine learning can lead to overly optimistic performance estimates during development but result in poor performance on real-world, unseen data. Therefore, it is crucial to be vigilant about preventing leakage at every stage of the machine learning pipeline.

**Kaggle: Machine Learning Explainability**

Machine Learning Explainability

* the ability to understand, interpret, and make sense of the predictions or decisions made by machine learning models. While many modern machine learning algorithms, especially complex models like deep neural networks, can achieve impressive predictive performance, their internal workings are often seen as "black boxes," making it challenging for humans to comprehend why a specific prediction was made.
* an evolving field, and researchers continue to develop new methods to make machine learning models more interpretable. Striking a balance between model complexity and interpretability is essential, as overly complex models may sacrifice interpretability for predictive power. The choice of explainability method depends on the specific requirements of the application and the stakeholders involved.

Explainability is crucial for various reasons:

1. **Trust and Transparency:**
   * Understanding how a model makes predictions fosters trust in its results. Users, stakeholders, and regulatory bodies often require transparency to trust and adopt machine learning solutions, especially in sensitive domains like healthcare, finance, and legal systems.
2. **Model Debugging:**
   * Explainability aids in identifying and rectifying issues with the model. If a model produces unexpected or incorrect results, explainability tools can help diagnose the problem and improve model performance.
3. **Fairness and Bias Detection:**
   * Explainability is essential for identifying and mitigating biases in machine learning models. It helps ensure that models are fair and do not discriminate against certain groups or individuals.
4. **Compliance with Regulations:**
   * In regulated industries, there are often legal requirements to provide explanations for algorithmic decisions. Explainability helps organizations comply with regulations such as the General Data Protection Regulation (GDPR) and Fair Lending laws.
5. **Human-in-the-Loop Systems:**
   * In applications where human experts are involved in decision-making alongside machine learning models (e.g., medical diagnoses), explainability is crucial for collaboration and effective decision-making.

Several approaches and techniques are used for achieving explainability in machine learning models:

1. **Feature Importance:**
   * Identifying which features or variables have the most significant impact on model predictions. This can be determined through techniques like permutation importance, tree-based feature importance, and LASSO regression.
2. **Local Explanations:**
   * Providing explanations for individual predictions. Techniques like LIME (Local Interpretable Model-agnostic Explanations) generate simple, interpretable models that approximate the behavior of the black-box model for a specific instance.
3. **Global Explanations:**
   * Understanding the overall behavior of the model. SHAP (SHapley Additive exPlanations) values and Partial Dependence Plots fall into this category, providing insights into the impact of features across the entire dataset.
4. **Rule-based Models:**
   * Translating the black-box model into a more interpretable form, such as decision rules. This can be done using techniques like rule extraction.
5. **Model-Agnostic Approaches:**
   * Techniques that can be applied to any model, regardless of the underlying algorithm. This includes methods like SHAP values, LIME, and Anchors.
6. **Sensitivity Analysis:**
   * Evaluating how changes in the input features affect the model's predictions. Sensitivity analysis helps understand the model's robustness and its sensitivity to variations in input data.

**Use Cases for Feature Importance**

techniques to extract the following insights from sophisticated machine learning models.

* What features in the data did the model think are most important?
* For any single prediction from a model, how did each feature in the data affect that particular prediction?
* How does each feature affect the model's predictions in a big-picture sense (what is its typical effect when considered over a large number of possible predictions)?

# Why Are These Insights Valuable

These insights have many uses, including

* Debugging
* Informing feature engineering
* Directing future data collection
* Informing human decision-making
* Building Trust

**Methods handling Feature Importance**

One of the most basic questions we might ask of a model is: What features have the biggest impact on predictions? This concept is called **feature importance**.

Handling feature importance is a crucial aspect of building and interpreting machine learning models. Feature importance helps understand the contribution of different features to the model's predictions.

The choice of method depends on the specific characteristics of your data, the type of model used, and the interpretability requirements. It's common to use a combination of these methods to get a comprehensive understanding of feature importance in a given context. Always remember to validate the results and consider the domain knowledge while interpreting feature importance.

1. **Inherent Feature Importance Methods:**
   * Some machine learning models provide inherent methods for assessing feature importance. For example:
     + **Decision Trees and Ensembles (Random Forest, Gradient Boosting):** These models often have built-in methods to calculate feature importance based on how frequently a feature is used to make decisions across multiple trees.
     + **Linear Models:** Coefficients in linear models provide information about the impact of each feature on the predicted outcome.
2. **Permutation Importance:**
   * **Method:** Involves randomly shuffling the values of a single feature and measuring the change in model performance.
   * **Idea:** If a feature is important, shuffling its values should significantly degrade model performance. And it is consistent with feature importance measure
   * **Implementation:** Permutation importance is calculated by comparing the model's performance metric (e.g., accuracy) before and after permuting each feature. Permutation importance is calculated after a machine learning model has been fitted
3. **Recursive Feature Elimination (RFE):**
   * **Method:** Iteratively removes the least important features until a specified number of features is reached.
   * **Idea:** Features that are consistently ranked as less important during the elimination process are considered less crucial to the model.
4. **LASSO Regression (L1 Regularization):**
   * **Method:** LASSO introduces a penalty term in the linear regression model that encourages sparsity in the coefficients.
   * **Idea:** LASSO tends to drive some coefficients to exactly zero, effectively eliminating the corresponding features.
5. **Tree-based Feature Importance:**
   * **Method:** Involves using tree-based models (e.g., Random Forest, XGBoost) and extracting feature importance scores from the model.
   * **Idea:** Features with higher importance scores are more influential in making decisions within the ensemble of trees.
6. **Correlation and Mutual Information:**
   * **Method:** Measures the statistical relationships between features and the target variable.
   * **Idea:** Highly correlated or informative features are likely to be more important.
7. **Shapley Values:**
   * **Method:** Originating from cooperative game theory, Shapley values allocate a value to each player (feature) based on their marginal contributions to all possible coalitions.
   * **Idea:** Shapley values provide a fair way to distribute the model's prediction among individual features.
8. **Partial Dependence Plots (PDP) and Individual Conditional Expectation (ICE):**
   * **Method:** PDP shows the average effect of a feature on the predicted outcome by varying that feature while keeping other features constant. ICE plots show the effect for individual instances.
   * **Idea:** Reveals the relationship between a feature and the predicted outcome.
   * **Implementation:**

**Permutation Importance**

|  |
| --- |
| import eli5  from eli5.sklearn import PermutationImportance  perm = PermutationImportance(my\_model, random\_state=1).fit(val\_X, val\_y)  eli5.show\_weights(perm, feature\_names = val\_X.columns.tolist()) |

* Interpretation of the results

텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명

**Important**

**Feature**

that degrades accuracy performance

**More accurate**

* The larger values are the most important features, and the lower values are the least importance features.
* The first number in each row shows how much model performance decreased with a random shuffling (in this case, using "accuracy" as the performance metric).
* The number after the **±** measures how performance varied from one-reshuffling to the next.
* negative values for permutation importances. In those cases, the predictions on the shuffled (or noisy) data happened to be more accurate than the real data. This happens when the feature didn't matter (should have had an importance close to 0), but random chance caused the predictions on shuffled data to be more accurate. This is more common with small datasets

**Partial Dependence Plots (PDP)**

While feature importance shows what variables most affect predictions, partial dependence plots show how a feature affects predictions.

This is useful to answer questions like:

* Controlling for all other house features, what impact do longitude and latitude have on home prices? To restate this, how would similarly sized houses be priced in different areas?
* Are predicted health differences between two groups due to differences in their diets, or due to some other factor?

SHAP Values

what if you want to break down how the model works for an individual prediction? SHAP Values (an acronym from SHapley Additive exPlanations) break down a prediction to show the impact of each feature.

SHAP Values to explain individual predictions in this lesson. In the next lesson, you'll see how these can be aggregated into powerful model-level insights.

How they work

SHAP values interpret the impact of having a certain value for a given feature in comparison to the prediction we'd make if that feature took some baseline value.

|  |
| --- |
| sum(SHAP values for all features) = pred\_for\_team -pred\_for\_baseline\_values |

That is, the SHAP values of all features sum up to explain why my prediction was different from the baseline. This allows us to decompose a prediction in a graph like this:



How do you interpret this?

We predicted 0.7, whereas the base\_value is 0.4979. Feature values causing increased predictions are in pink, and their visual size shows the magnitude of the feature's effect. Feature values decreasing the prediction are in blue. The biggest impact comes from Goal Scored being 2. Though the ball possession value has a meaningful effect decreasing the prediction.

If you subtract the length of the blue bars from the length of the pink bars, it equals the distance from the base value to the output.

**Kaggle: Feature Engineering**

**What is Feature Engineering**

The Goal of Learning

* Learn the steps and principles of creating better features to build a great machine learning model
* Learn how to
  + determine which features are the most important with *mutual information*
  + invent new features in several real-world problem domains
  + encode high-cardinality categoricals with a *target encoding*
  + create segmentation features with *k-means clustering*
  + decompose a dataset's variation into features with *principal component analysis*

The Goal of Feature Engineering

Feature engineering is to make your data better suited to the problem at hand.

Perform feature engineering to

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

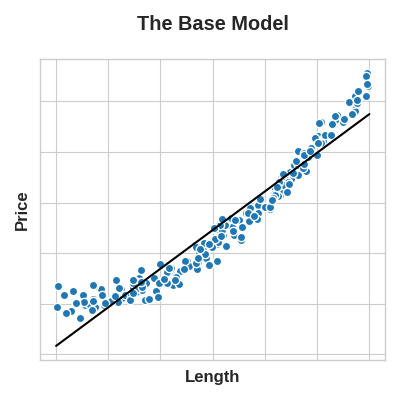
A guiding principle of feature engineering

Key Idea: Transformation of a feature

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

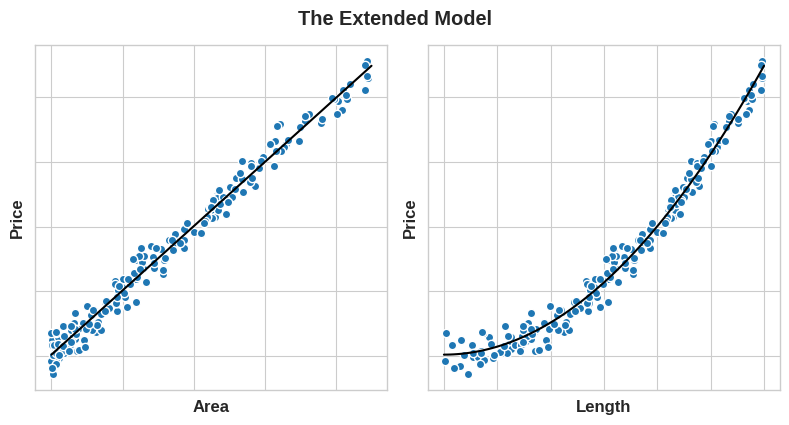
The key idea here is that a transformation you apply to a feature becomes in essence a part of the model itself.

EX. Fitting a linear model directly to Length gives poor results: the relationship is not linear.



If we square the Length feature 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.*

* Whatever relationships your model can't learn, you can provide yourself through transformations. As you develop your feature set, think about what information your model could use to achieve its best performance

Key Idea: Baseline construction

Baseline

* We'll first establish a baseline by training the model on the un-augmented dataset
* Establishing baselines like this is good practice at the start of the feature engineering process.

A baseline score can help you decide whether your new features are worth keeping, or whether you should discard them and possibly try something else

Common techniques and concepts related to feature engineering

1. **Handling Missing Data:**
   * Identify and fill missing values in the dataset using techniques such as mean, median, or mode imputation.
   * Create binary indicators to flag missing values in a particular feature.
2. **Encoding Categorical Variables:**
   * Convert categorical variables into numerical format, as many machine learning algorithms require numerical input.
   * Common encoding techniques include one-hot encoding, label encoding, and target encoding.
3. **Creating Interaction Terms:**
   * Introduce new features by combining existing ones to capture interaction effects. For example, if you have features A and B, you can create a new feature AB representing their product.
4. **Scaling and Normalization:**
   * Standardize or normalize numerical features to ensure that they have similar scales. This is important for algorithms sensitive to the magnitude of input features, such as gradient descent-based methods.
5. **Binning or Discretization:**
   * Convert continuous variables into discrete bins or categories. This can help capture non-linear relationships and patterns in the data.
6. **Feature Splitting:**
   * Divide a feature into multiple components to extract more information. For example, splitting a date into day, month, and year can provide additional insights.
7. **Transformation of Skewed Data:**
   * Apply mathematical transformations (e.g., logarithmic or square root transformations) to features with skewed distributions to make them more symmetric and suitable for certain algorithms.
8. **Text Data Processing:**
   * Convert textual data into numerical format using techniques like TF-IDF (Term Frequency-Inverse Document Frequency) or word embeddings.
9. **Feature Selection:**
   * Identify and keep only the most relevant features to reduce dimensionality and potentially improve model performance. Techniques include univariate feature selection, recursive feature elimination, and feature importance from tree-based models.
10. **Domain-Specific Feature Engineering:**
    * Leverage domain knowledge to create features that are specific to the problem at hand. This can involve creating ratios, averages, or other derived features based on a deep understanding of the data and the problem domain.

Effective feature engineering can significantly impact the performance of machine learning models, leading to better generalization, faster training times, and improved interpretability. It often requires a combination of domain expertise, creativity, and experimentation to identify the most informative features for a given task.

**Mutual Information**

Q engineering new features can improve model performance. But how do you identify features in the dataset that might be useful to combine?

A Locate features with the most potential.

Problem: You might be presented with hundreds or thousands of features without even a description to go by. Where do you even begin?

Solution: 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 features to develop initially.

**Mutual information** of the metric

* correlation in that it measures a relationship between two quantities.
* The advantage of mutual information is that it can detect any kind of relationship, while correlation only detects linear relationships.
* 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
* describes relationships in terms of uncertainty. The **mutual information** (MI) between two quantities is a measure of the extent to which knowledge of one quantity reduces uncertainty about the other. If you knew the value of a feature, how much more confident would you be about the target?
* 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.

**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. Conversely, in theory there's no upper bound to what MI can be. In practice though values above 2.0 or so are uncommon. (Mutual information is a logarithmic quantity, so it increases very slowly.)
* idea of how MI values correspond to the kind and degree of association a feature has with the target.

라인, 도표, 텍스트, 그래프이(가) 표시된 사진

자동 생성된 설명

***Left:****Mutual information increases as the dependence between feature and target becomes tighter.****Right:****Mutual information can capture any kind of association (not just linear, like correlation.)*

* Here are some things to remember when applying mutual information:
* MI can help you to understand the *relative potential* of a feature 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 interactions* between features. It is a **univariate** metric.
* The *actual* 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. Just because a feature has a high MI score doesn't mean your model will be able to do anything with that information. You may need to transform the feature first to expose the association.

Mutual information is a measure of the statistical dependence between two variables. In the context of feature engineering, mutual information is often used for feature selection. It quantifies the amount of information obtained about one variable through the observation of another variable. Techniques similar to the concept of mutual information include:

1. **Feature Selection Based on Mutual Information:**
   * Instead of using traditional measures like correlation, you can use mutual information to identify the most informative features. Features with high mutual information with the target variable are considered more relevant.
2. **Information Gain in Decision Trees:**
   * Decision trees use information gain as a criterion to decide how to split the data at each node. Information gain is closely related to mutual information, as both measure the reduction in uncertainty (entropy) when a variable is known.
3. **Entropy-Based Binning:**
   * When discretizing continuous variables, an entropy-based binning approach can be used. This technique involves finding bin boundaries that maximize the mutual information between the original continuous variable and the discretized version.
4. **Maximum Relevance, Minimum Redundancy (MRMR):**
   * MRMR is a feature selection technique that aims to find the subset of features that maximizes the mutual information with the target variable while minimizing redundancy between selected features.
5. **Joint Mutual Information:**
   * Instead of considering individual features, joint mutual information measures the dependence between sets of features and the target variable. It evaluates the information gained by considering combinations of features.
6. **Conditional Mutual Information:**
   * This measures the information gained about one variable when the value of another variable is known. It is an extension of mutual information and is used in some feature selection algorithms.
7. **Kernelized Mutual Information:**
   * In some cases, especially when dealing with non-linear relationships, kernelized versions of mutual information can be used. Kernelized mutual information allows for capturing non-linear dependencies between variables.

These techniques share the common theme of assessing the information content or dependence between variables, making them similar in spirit to the concept of mutual information. They are often employed in feature engineering and selection to identify the most relevant features for a given machine learning task.

**Creating Features**

**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](https://scholar.google.com/) will often have the best information.
* Study previous work. [Solution write-ups](https://www.kaggle.com/sudalairajkumar/winning-solutions-of-kaggle-competitions) 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.

**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.
* Group transformations

Methods of creating features

1. **Imputation:**
   * Handling missing values by imputing them with mean, median, or mode values.
   * Creating binary indicator variables to denote missing values.
2. **One-Hot Encoding:**
   * Converting categorical variables into binary vectors using one-hot encoding.
   * Each category becomes a binary feature, and the presence or absence of the category is represented by 1 or 0.
3. **Label Encoding:**
   * Assigning unique numerical labels to categories.
   * Useful for ordinal categorical variables where the order matters.
4. **Binning or Discretization:**
   * Grouping continuous variables into bins or discrete intervals.
   * Can help capture non-linear relationships or patterns in the data.
5. **Interaction Terms:**
   * Creating new features by combining two or more existing features.
   * For example, combining height and weight to create a Body Mass Index (BMI) feature.
6. **Polynomial Features:**
   * Generating higher-degree polynomial features from existing features.
   * Useful for capturing non-linear relationships in the data.
7. **Logarithmic and Exponential Transformations:**
   * Applying logarithmic or exponential functions to features.
   * Useful for handling skewed distributions or emphasizing certain patterns.
8. **Feature Scaling:**
   * Scaling numerical features to a standard range (e.g., 0 to 1) to ensure that they have similar magnitudes.
   * Common techniques include Min-Max scaling and Z-score normalization.
9. **Date and Time Features:**
   * Extracting relevant information from date and time variables, such as day of the week, month, quarter, or year.
   * Creating time-based features like time differences or time since a specific event.
10. **Text Features:**
    * Tokenizing and converting text data into numerical representations using techniques like TF-IDF (Term Frequency-Inverse Document Frequency) or word embeddings.
    * Extracting features like text length or word counts.
11. **Feature Crosses:**
    * Combining two or more features to create a new feature.
    * For example, combining 'age' and 'education' to create an 'education\_age' feature.
12. **Aggregation:**
    * Aggregating information across observations to create summary features.
    * Examples include calculating mean, median, sum, or standard deviation for a group of observations.
13. **Domain-Specific Features:**
    * Incorporating domain knowledge to create features that are relevant to the specific problem.
    * For example, in image processing, extracting texture features or edge features.
14. **Embeddings:**
    * Representing categorical variables with high cardinality using embeddings.
    * Commonly used in natural language processing and collaborative filtering.
15. **Feature Importance Techniques:**
    * Using algorithms or techniques that provide feature importance scores to identify and select the most relevant features.
16. **Dimensionality Reduction:**
    * Applying techniques like Principal Component Analysis (PCA) or t-Distributed Stochastic Neighbor Embedding (t-SNE) to reduce the dimensionality of the feature space.

Choosing the right set of features depends on the nature of the data and the problem at hand. It often involves a combination of exploratory data analysis, domain knowledge, and experimentation to identify the most informative features for a given machine learning task.

**Clustering with K-Means**

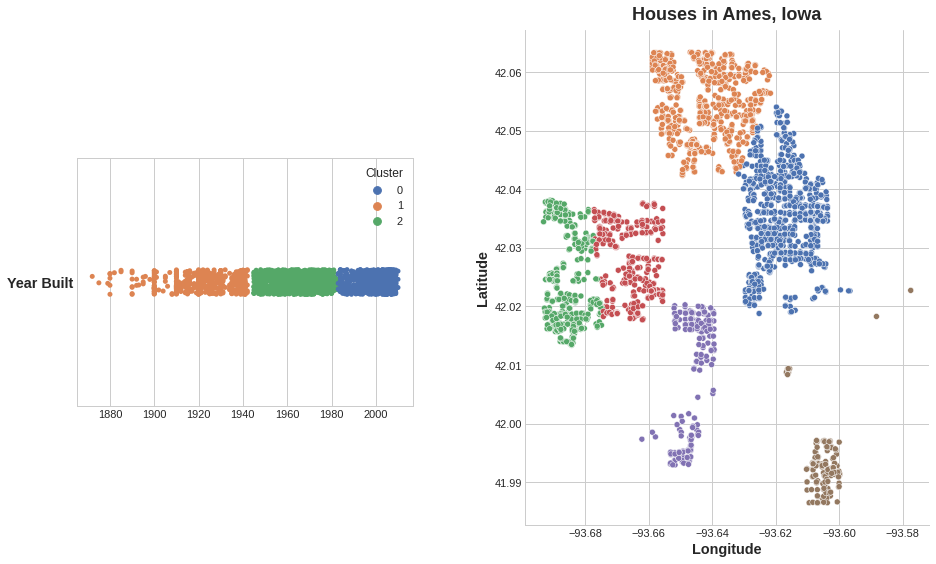
**Introduction**

unsupervised learning algorithms: the 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 prediction, you could think of an unsupervised algorithm as a "feature discovery" technique

**Clustering** simply means the assigning of data points to groups based upon how similar the points are to each other. Adding a feature of cluster labels can help machine learning models untangle complicated relationships of space or proximity

# **Cluster Labels as a Feature**

Applied to a single real-valued feature, clustering acts like a traditional "binning" or ["discretization"](https://scikit-learn.org/stable/auto_examples/preprocessing/plot_discretization_classification.html) transform. On multiple features, it's like "multi-dimensional binning" (sometimes called vector quantization).



***Left:****Clustering a single feature.****Right:****Clustering across two features*

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.

The motivating idea for adding cluster labels is that the clusters will break up complicated relationships across features into simpler chunks

# **Clustering algorithms**

# how they measure "similarity" or "proximity" and in what kinds of features they work with

# **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. 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 the k yourself.

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 tessallation shows you to what clusters future data will be assigned; the tessallation is essentially what k-means learns from its training data.

Since k-means clustering is sensitive to scale, it can be a good idea rescale or normalize data with extreme values. 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. Sometimes, the choice won't be clear though. In that case, you should try to use common sense, remembering that features with larger values will be weighted more heavily.

The k-means algorithm offers an alternative way of creating features. Instead of labelling each feature with the nearest cluster centroid, it can measure the distance from a point to all the centroids and return those distances as features.

**Principal Component Analysis**

* Discover new features by analyzing variation.
* PCA as a partitioning of the variation in the data
* help you discover important relationships in the data and can also be used to create more informative features.
* PCA is typically applied to [standardized](https://www.kaggle.com/alexisbcook/scaling-and-normalization) data. With standardized data "variation" means "correlation". With unstandardized data "variation" means "covariance". All data in this course will be standardized before applying PCA.
* within this data are "axes of variation"; Pictorially, these axes appear as perpendicular lines running along the natural dimensions of the data, one axis for each original feature.

텍스트, 라인, 도표, 그래프이(가) 표시된 사진

자동 생성된 설명

* names to these axes of variation. The longer axis we might call the "Size" component: small height and small diameter (lower left) contrasted with large height and large diameter (upper right). The shorter axis we might call the "Shape" component: small height and large diameter (flat shape) contrasted with large height and small diameter (round shape)
* Notice that instead of describing abalones by their '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 describe it with its axes of variation. The axes of variation become the new features.

텍스트, 도표, 그래프, 라인이(가) 표시된 사진

자동 생성된 설명

*The principal components become the new features by a rotation of the dataset in the feature space.*

* The new features PCA constructs are actually just linear combinations (weighted sums) of the original features: These new features are called the **principal components** of the data; 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 weights tell us what variation it expresses through signs and magnitudes:

|  |  |  |
| --- | --- | --- |
| Features \ Components | Size (PC1) | Shape (PC2) |
| Height | 1 | 1 |
| Diameter | 1 | -1 |

This table of weights 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 weights are all of the same magnitude and 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 the data along the Size component than along the Shape component. PCA makes this precise through each component's **percent of explained variance**.

텍스트, 라인, 도표, 그래프이(가) 표시된 사진

자동 생성된 설명

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

# **PCA for Feature Engineering**

# two ways you could use PCA for feature engineering.

* 1. 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
* 2. 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 basically gives you direct access to the correlational structure of your data.

**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.

# **Target Encoding**

* 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; **supervised** feature engineering technique
* any kind of encoding that replaces a feature's categories with some number derived from the target.
  + - **mean encoding;** Applied to a binary target, it's also called **bin counting**
    - likelihood encoding, impact encoding, and leave-one-out encoding.
* Problem: overfitting by unknown categories or rare categories

Solution: smoothing which 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.

* **Use Cases for Target Encoding**  
  Target encoding is great for:

1. **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.
2. **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.

* *The Art of Feature Engineering*, a book by Pablo Duboue.
* *An Empirical Analysis of Feature Engineering for Predictive Modeling*, an article by Jeff Heaton.
* *Feature Engineering for Machine Learning*, a book by Alice Zheng and Amanda Casari. The tutorial on clustering was inspired by this excellent book.
* *Feature Engineering and Selection*, a book by Max Kuhn and Kjell Johnson.

|  |
| --- |
| 1) Choose Features for Encoding # Encoding split  X\_encode = df.sample(frac=0.20, random\_state=0)  y\_encode = X\_encode.pop("SalePrice")  # Training split  X\_pretrain = df.drop(X\_encode.index)  y\_train = X\_pretrain.pop("SalePrice") 2) Apply M-Estimate Encoding encoder = MEstimateEncoder(cols=["Neighborhood"], m=5.0)  # Fit the encoder on the encoding split  encoder.fit(X\_encode, y\_encode)  # Encode the training split  X\_train = encoder.transform(X\_pretrain, y\_train) |

# Kaggle: Data Cleaning

# **Handling Missing Values**

Problems

Why are some of your text fields garbled? What should you do about those missing values? Why aren’t your dates formatted correctly? How can you quickly clean up inconsistent data entry?

Goals

Learn why you've run into these problems and, more importantly, how to fix them!

1. Take a first look at the data
2. Check How many missing data points do we have?

|  |
| --- |
| # get the number of missing data points per column  missing\_values\_count = nfl\_data.isnull().sum()  # look at the # of missing points in the first ten columns  missing\_values\_count[0:10] |

# Figure out why the data is missing

Data intuition: For dealing with missing values, you'll need to use your intution to figure out why the value is missing. One of the most important questions you can ask yourself to help figure this out is this

* **Is this value missing because it wasn't recorded or because it doesn't exist?**

If a value is missing becuase it doesn't exist, these values you probably do want to keep as NaN.

If a value is missing because it wasn't recorded, then you can try to guess what it might have been based on the other values in that column and row. This is called **imputation**

# Drop missing values

If you're in a hurry or don't have a reason to figure out why your values are missing, one option is to drop any columns or rows that contain missing values

Not recommend this approach for important projects

If you're sure you want to drop rows with missing values, pandas does have a handy function, dropna()

1. Filling in missing values automatically

Use Panda's fillna() function to fill in missing values in a dataframe or series. One option we have is to specify what we want the NaN values to be replaced with

* replace missing values with whatever value comes directly after it in the same column. (This makes a lot of sense for datasets where the observations have some sort of logical order to them.)

**Scaling and Normalization**

* Transform numeric variables to have helpful properties.
* Look at how to scale and normalize data (and what the difference is between the two!)

|  |
| --- |
| *# modules we'll use*  import pandas as pd  import numpy as np  *# for Box-Cox Transformation*  from scipy import stats  *# for min\_max scaling*  from mlxtend.preprocessing import minmax\_scaling  *# plotting modules*  import seaborn as sns  import matplotlib.pyplot as plt |

# **Scaling vs. Normalization: What's the difference?**

One of the reasons that it's easy to get confused between scaling and normalization is because the terms are sometimes used interchangeably and, to make it even more confusing, they are very similar! In both cases, you're transforming the values of numeric variables so that the transformed data points have specific helpful properties. The difference is that:

* in **scaling**, you're changing the range of your data, while
* in **normalization**, you're changing the shape of the distribution of your data.

# **Scaling**

you're transforming your data so that it fits within a specific scale, like 0-100 or 0-1. You want to scale data when you're using methods based on measures of how far apart data points are, like [support vector machines (SVM)](https://en.wikipedia.org/wiki/Support_vector_machine) or [k-nearest neighbors (KNN)](https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm). With these algorithms, a change of "1" in any numeric feature is given the same importance.

그래프, 라인, 도표, 스크린샷이(가) 표시된 사진

자동 생성된 설명

Notice that the shape of the data doesn't change, but that instead of ranging from 0 to 8, it now ranges from 0 to 1.

# **Normalization**

Scaling just changes the range of your data

The point of normalization is to change your observations so that they can be described as a normal distribution.

[**Normal distribution:**](https://en.wikipedia.org/wiki/Normal_distribution) Also known as the "bell curve", this is a specific statistical distribution where a roughly equal observations fall above and below the mean, the mean and the median are the same, and there are more observations closer to the mean. The normal distribution is also known as the Gaussian distribution.

도표, 그래프, 라인이(가) 표시된 사진

자동 생성된 설명

Notice that the shape of our data has changed. Before normalizing it was almost L-shaped. But after normalizing it looks more like the outline of a bell (hence "bell curve").

# Kaggle: Computer Vision

Learning is to

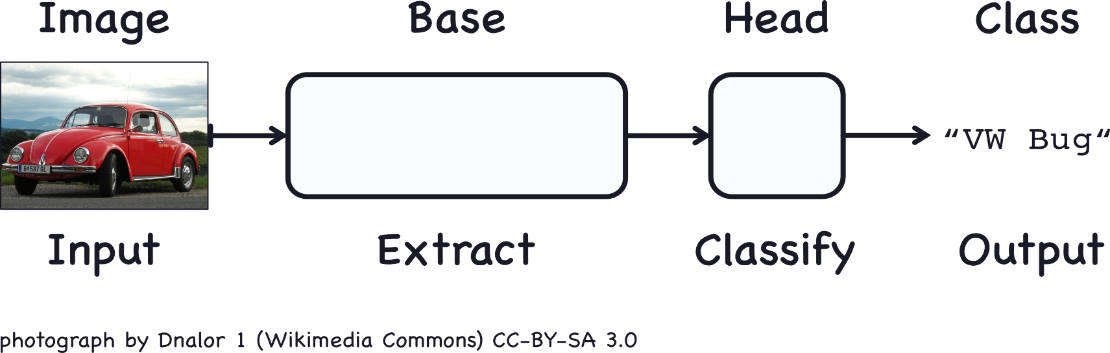
* Use modern deep-learning networks to build an **image classifier** with Keras
* Design your own **custom convnet** with reusable blocks
* Learn the fundamental ideas behind **visual feature extraction**
* Master the art of **transfer learning** to boost your models
* Utilize **data augmentation** to extend your dataset

Computer vision

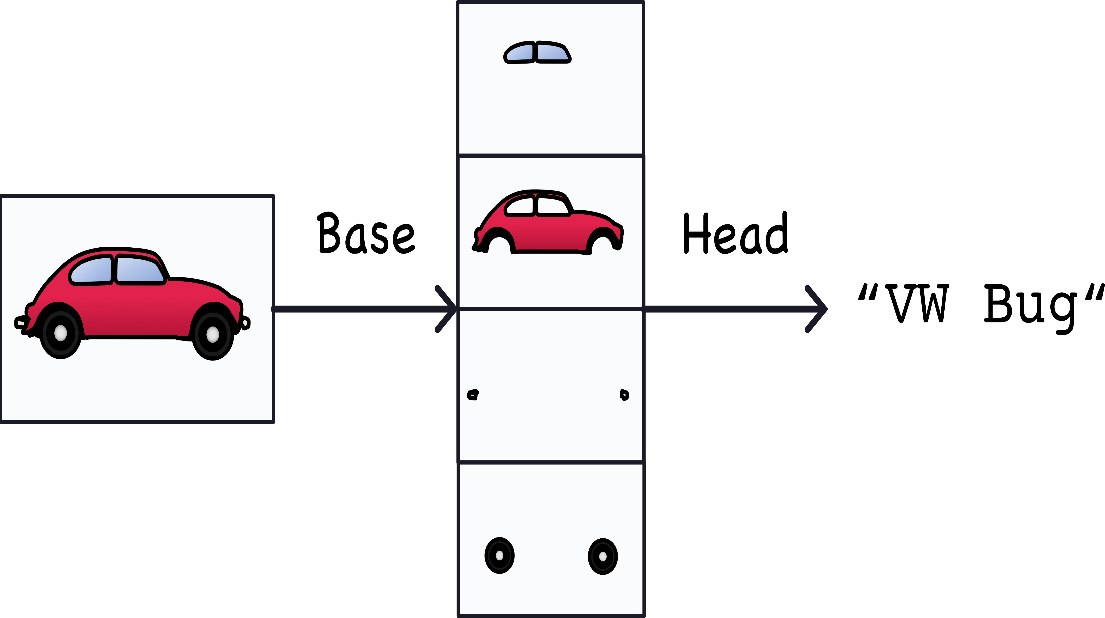
* multidisciplinary field of study that enables computers to gain high-level understanding from digital images or videos
* goal is to teach machines to interpret and understand visual information in a way that is similar to human vision
* key components and concepts
* **Image Acquisition:** The process of capturing visual data, often using cameras or other imaging devices. Images can be in the form of photographs, video frames, or other types of visual input.
* **Image Processing:** Techniques to enhance or modify the acquired images. This can include filtering, noise reduction, and other operations to improve the quality of the visual data.
* **Image Analysis:** The process of extracting meaningful information from images. This involves tasks such as object detection, segmentation, and feature extraction.
* **Pattern Recognition:** Identifying and categorizing patterns within the visual data. This can include recognizing objects, faces, text, or any other relevant information.
* **Machine Learning:** Many computer vision tasks involve training machine learning models to automatically learn patterns and features from data. Deep learning, a subset of machine learning, has been particularly successful in various computer vision applications.
* **Object Detection:** Identifying and locating objects within an image or video stream. This is crucial for applications like autonomous vehicles, surveillance, and robotics.
* **Image Classification:** Assigning a label or category to an entire image. For example, recognizing whether an image contains a cat or a dog.
* **Segmentation:** Dividing an image into meaningful segments or regions. This is useful for tasks such as identifying boundaries and understanding the spatial relationships between objects.
* **Feature Extraction:** Identifying and extracting relevant features from images. These features serve as input for machine learning models.
* **3D Computer Vision:** Extending the principles of computer vision to three-dimensional space, allowing machines to perceive depth and understand the three-dimensional structure of objects.

**Convolutional Classifier**

* Goal: to learn how a neural network can "understand" a natural image well-enough to solve the same kinds of problems the human visual system can solve.
* **convolutional neural networks** (Sometimes we say **convnet** or **CNN** instead.) Convolution is the mathematical operation that gives the layers of a convnet their unique structure
* **image classification**
* **Convolutional Classifier**
  + - A convnet used for image classification consists of two parts: a **convolutional base** and a **dense head**.



* + - The base is used to **extract the features** from an image. It is formed primarily of layers performing the convolution operation, but often includes other kinds of layers as well.
    - The head is used to **determine the class** of the image. It is formed primarily of dense layers, but might include other layers like dropout.
    - A feature could be a line, a color, a texture, a shape, a pattern -- or some complicated combination. The features actually extracted look a bit different, but it gives the idea. The whole process goes something like this:

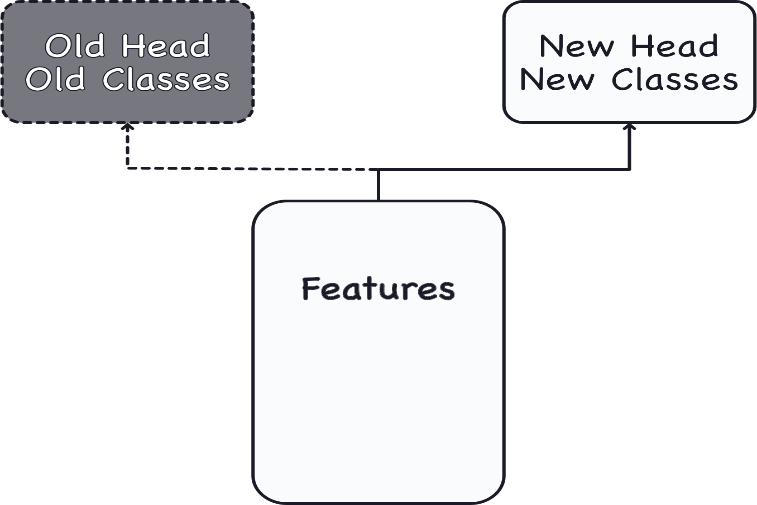


* **Training the classifier**
* The goal of the network during training is to learn two things:

which features to extract from an image (base),

which class goes with what features (head).

* + - These days, convnets **reuse the base of a pretrained model**; To the pretrained base we then **attach an untrained head**. In other words, we reuse the part of a network that has already learned to do 1. Extract features, and attach to it some fresh layers to learn 2. Classify.



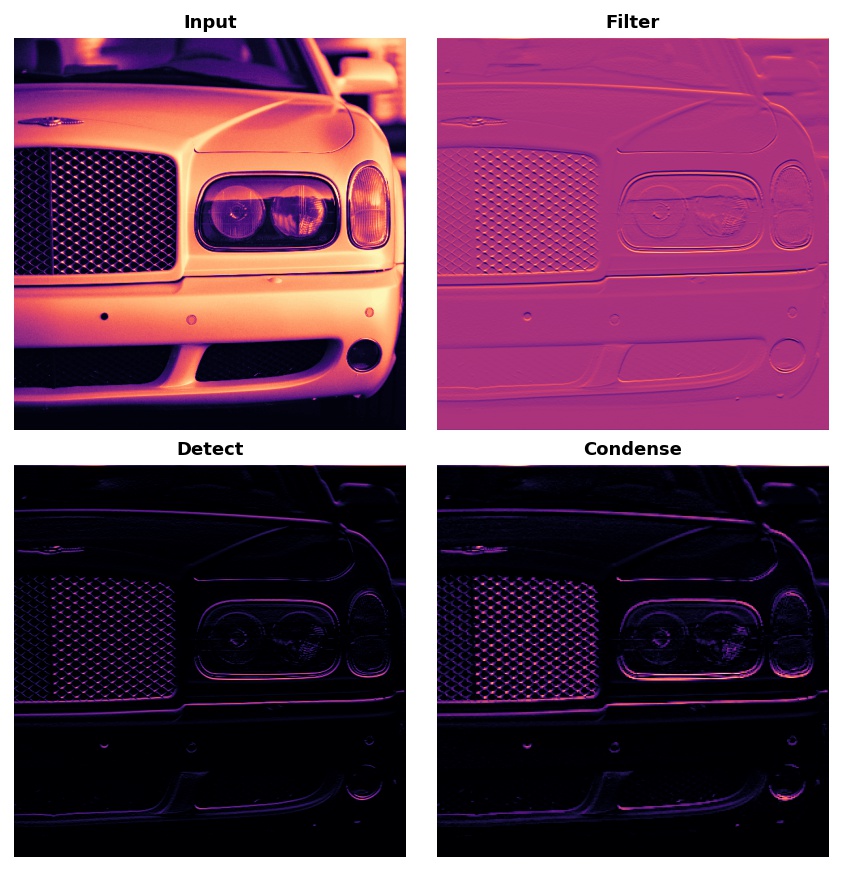
* + - Because the head usually consists of only a few dense layers, very accurate classifiers can be created from relatively little data.
    - **transfer learning:** Reusing a pretrained model is a technique; almost every image classifier these days will make use of it.

**Convolution and ReLU**

Discover how convnets create features with convolutional layers.

The **feature extraction** performed by the base consists of **three basic operations**:

1. **Filter** an image for a particular feature (convolution)
2. **Detect** that feature within the filtered image (ReLU)
3. **Condense** the image to enhance the features (maximum pooling)



Typically, the network will perform several extractions in parallel on a single image. In modern convnets, it's not uncommon for the final layer in the base to be producing over 1000 unique visual features.

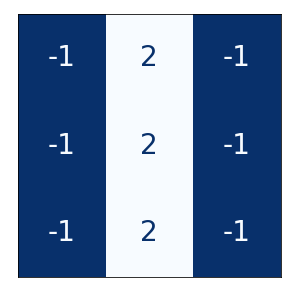
# **Filter with Convolution**

* A convolutional layer carries out the filtering step. You might define a convolutional layer in a Keras model

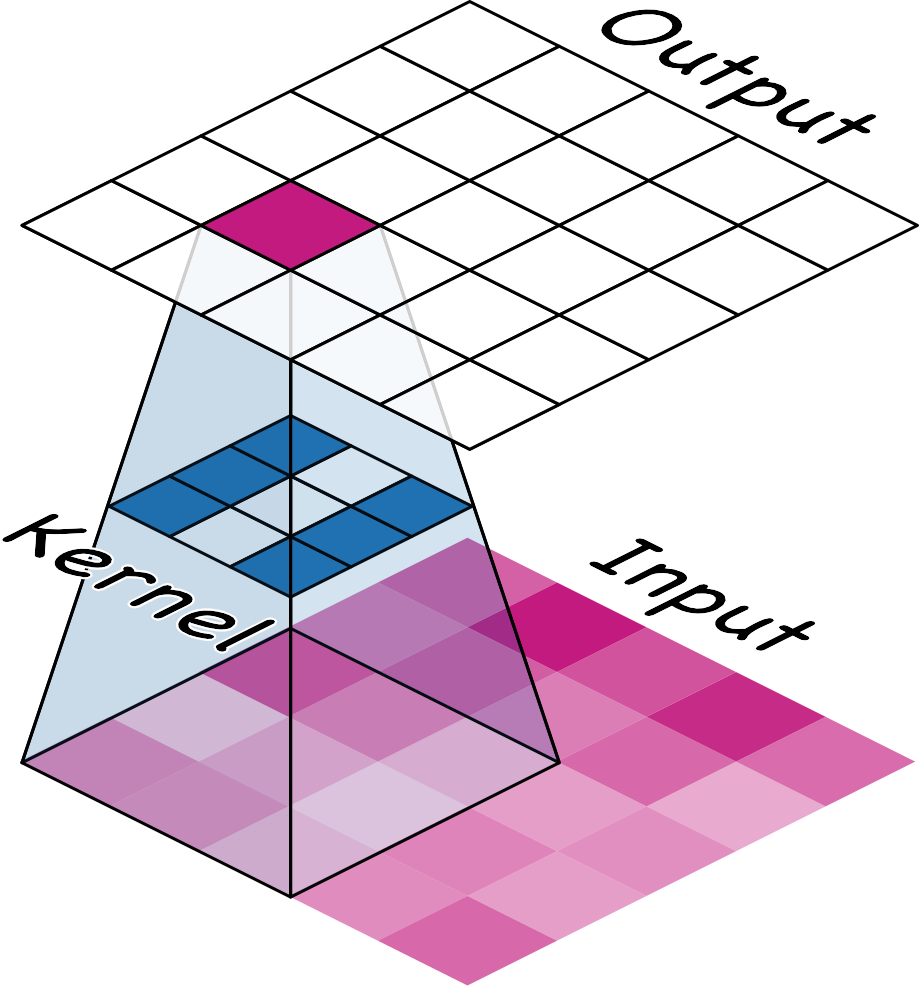
|  |
| --- |
| from tensorflow import keras  from tensorflow.keras import layers  model = keras.Sequential([  layers.Conv2D(filters=64, kernel\_size=3), *# activation is None*  *# More layers follow*  ]) |

* Understand the parameters by looking at their relationship to the weights and activations of the convolutional layer
* Convolutional layer는 filtering을 수행한다, filtering은 weight과 activation을 사용하여 수행된다, weight은 training 동안에 kernel을 사용하여 배열형태로 형성된다 kernel은 weight과 함께 feature를 추출한다. 이 결과로써 feature map이 생성되는 것이고 이것을 convolution layer에서의 activation이라고 한다; filter with convolution = kernel with weight -> activation with feature maps
  + **Weights**

The **weights** a convnet learns during training are primarily contained in its convolutional layers. These weights we call **kernels.** We can represent them as small arrays:



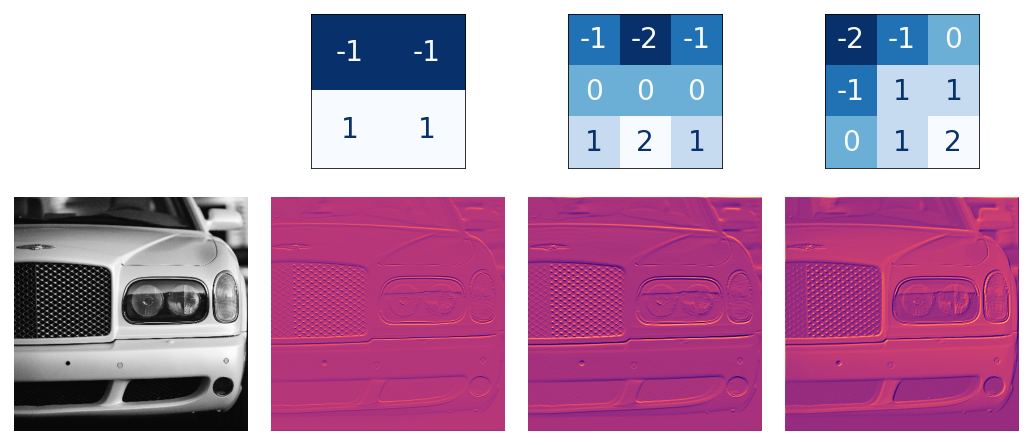
A kernel operates by scanning over an image and producing a weighted sum of pixel values. In this way, a kernel will act sort of like a polarized lens, emphasizing or deemphasizing certain patterns of information.

*A kernel acts as a kind of lens.*

Kernels define how a convolutional layer is connected to the layer that follows. The kernel above will connect each neuron in the output to nine neurons in the input. By setting the dimensions of the kernels with kernel\_size, you are telling the convnet how to form these connections. Most often, a kernel will have odd-numbered dimensions so that a single pixel sits at the center, but this is not a requirement. The kernels in a convolutional layer determine what kinds of features it creates. During training, a convnet tries to learn what features it needs to solve the classification problem. This means finding the best values for its kernels.

* + **Activations**

The **activations** in the network we call **feature maps**. They are what result when we apply a filter to an image; they contain the visual features the kernel extracts.

*Kernels and features.*

From the pattern of numbers in the kernel, you can tell the kinds of feature maps it creates. Generally, what a convolution accentuates in its inputs will match the shape of the positive numbers in the kernel. The left and middle kernels above will both filter for horizontal shapes.

With the filters parameter, you tell the convolutional layer how many feature maps you want it to create as output.

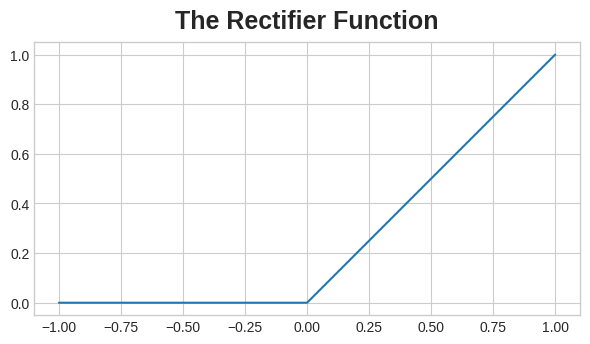
# **Detect with activation function(ex, ReLU)**

* After filtering, the feature maps pass through the activation function

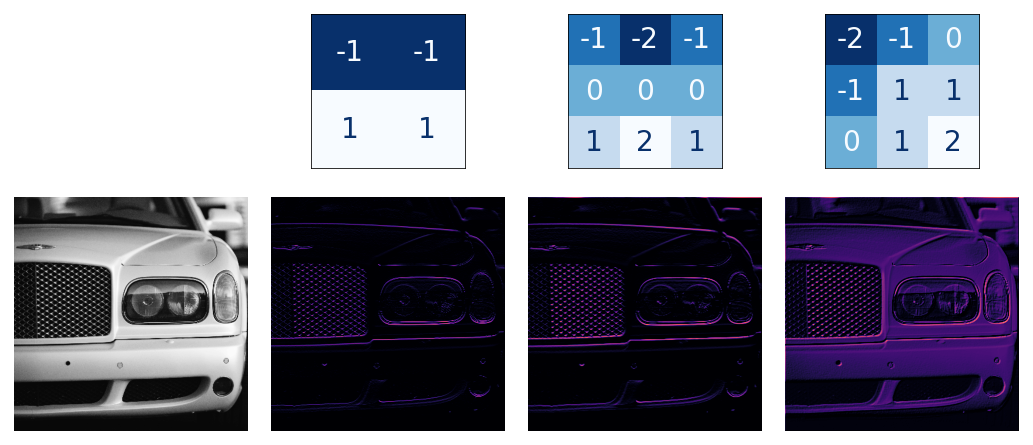
The activation function can be defined in its own Activation layer, but most often you'll just include it as the activation function of Conv2D.

|  |
| --- |
| model = keras.Sequential([  layers.Conv2D(filters=64, kernel\_size=3, activation='relu')  *# More layers follow*  ]) |

* activation function as scoring pixel values according to some measure of importance; ex. The ReLU activation says that negative values are not important and so sets them to 0. ("Everything unimportant is equally unimportant.")



Here is ReLU applied the feature maps above. Notice how it succeeds at isolating the features.

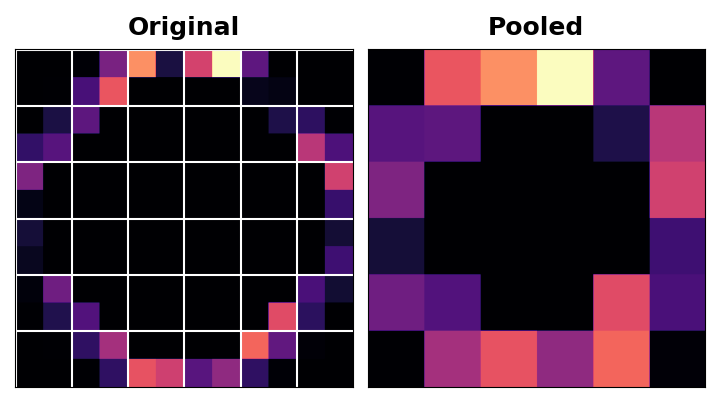


**Nonlinear activation function:** Essentially this means that the total effect of all the layers in the network becomes different than what you would get by just adding the effects together -- which would be the same as what you could achieve with only a single layer. The nonlinearity ensures features will combine in interesting ways as they move deeper into the network. (We'll explore this "feature compounding"

|  |
| --- |
| Import tensorflow as tf   * 1. Load image   image\_path = '../input/computer-vision-resources/car\_illus.jpg'  image = tf.io.read\_file(image\_path)  image = tf.io.decode\_jpeg(image, channels=1)  image = tf.image.resize(image, size=[400, 400])  img = tf.squeeze(image).numpy()  plt.figure(figsize=(6, 6))  plt.imshow(img, cmap='gray')  plt.axis('off')  plt.show();   * 1. Define kernel   kernel = tf.constant([[0,1,0], [0,0,1],[1,1,1],])   * 1. Reformat for batch compatibility   image= tf.image.convert\_image\_dtype(image, dtype=tf.float32)  image= tf.expand\_dims(image, axis=0)  kernel= tf.reshape(kernel, [\*kernel.shape, 1,1])  kernel= tf.cast(kernel, dtype=tf.float32)   * 1. Apply convolution(apply the kernel to the image by a convolution)   conv\_fn = tf.nn.conv2d(  input=image,  filters=kernel,  strides=1,  padding=’SAME’  )  plt.imshow(  # Reformat for plotting  tf.squeeze(conv\_fn)  )  plt.axis('off')  plt.show();   * 1. Apply ReLU(detect the feature with the ReLU function)   relu\_fn = tf.nn.relu(conv\_fn)  plt.imshow(  # Reformat for plotting  tf.squeeze(relu\_fn)  )  plt.axis('off')  plt.show(); |

**Maximum Pooling**

* **condense** with **maximum pooling,** which in Keras is done by a MaxPool2D layer.
* MaxPool2D layer doesn't have any trainable weights like a convolutional layer does in its kernel, however.
* Notice that after applying the ReLU function (**Detect**) the feature map ends up with a lot of "dead space," that is, large areas containing only 0's (the black areas in the image). Having to carry these 0 activations through the entire network would increase the size of the model without adding much useful information. Instead, we would like to condense the feature map to retain only the most useful part -- the feature itself.
* Max pooling takes a patch of activations in the original feature map and replaces them with the maximum activation in that patch.

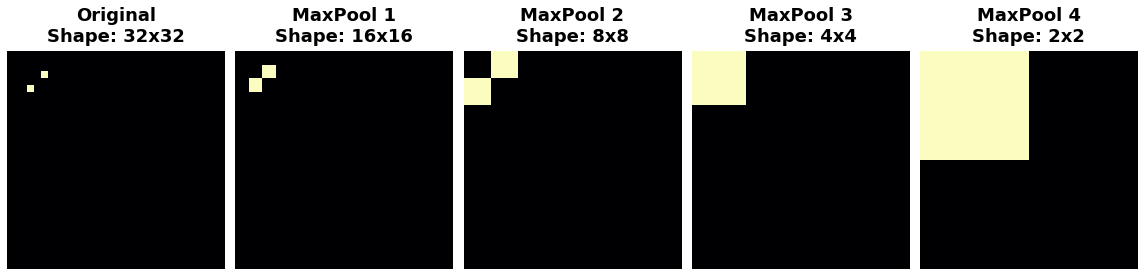


When applied after the ReLU activation, it has the effect of "intensifying" features. The pooling step increases the proportion of active pixels to zero pixels.

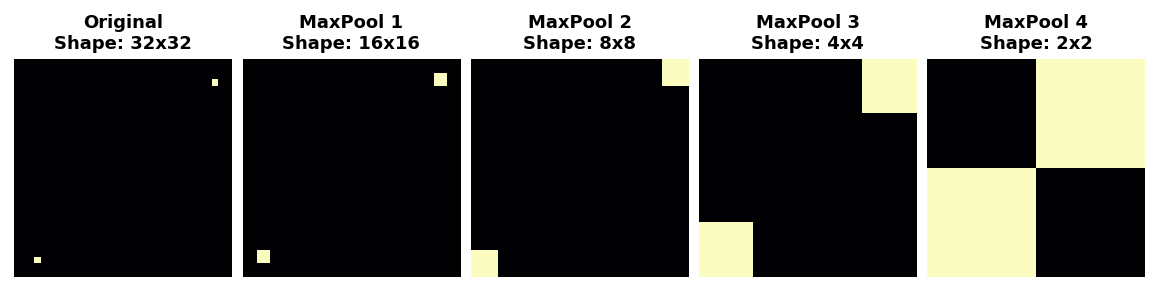
* **Translation Invariance**

We called the zero-pixels "unimportant". Does this mean they carry no information at all? In fact, the zero-pixels carry positional information. The blank space still positions the feature within the image. When MaxPool2D removes some of these pixels, it removes some of the positional information in the feature map. This gives a convnet a property called **translation invariance**. This means that a convnet with maximum pooling will tend not to distinguish features by their location in the image. ("Translation" is the mathematical word for changing the position of something without rotating it or changing its shape or size.)

Watch what happens when we repeatedly apply maximum pooling to the following feature map.



The two dots in the original image became indistinguishable after repeated pooling. In other words, pooling destroyed some of their positional information. Since the network can no longer distinguish between them in the feature maps, it can't distinguish them in the original image either: it has become invariant to that difference in position.



In fact, pooling only creates translation invariance in a network over small distances, as with the two dots in the image. Features that begin far apart will remain distinct after pooling; only some of the positional information was lost, but not all of it.

This invariance to small differences in the positions of features is a nice property for an image classifier to have. Just because of differences in perspective or framing, the same kind of feature might be positioned in various parts of the original image, but we would still like for the classifier to recognize that they are the same. Because this invariance is built into the network, we can get away with using much less data for training: we no longer have to teach it to ignore that difference. This gives convolutional networks a big efficiency advantage over a network with only dense layers. Cf. **Data Augmentation**!

**The Sliding Window**

The convolution and pooling operations share a common feature: they are both performed over a **sliding window**. With convolution, this "window" is given by the dimensions of the kernel, the parameter kernel\_size. With pooling, it is the pooling window, given by pool\_size.

There are two additional parameters affecting both convolution and pooling layers -- these are the strides of the window and whether to use padding at the image edges. The strides parameter says how far the window should move at each step, and the padding parameter describes how we handle the pixels at the edges of the input.

Stride

* The distance the window moves at each step
* specify the stride in both dimensions of the image: one for moving left to right and one for moving top to bottom. strides=(2, 2), a movement of 2 pixels each step.
* What effect does the stride have? Whenever the stride in either direction is greater than 1, the sliding window will skip over some of the pixels in the input at each step.
* Because we want high-quality features to use for classification, convolutional layers will most often have strides=(1, 1). Increasing the stride means that we miss out on potentially valuble information in our summary. Maximum pooling layers, however, will almost always have stride values greater than 1, like (2, 2) or (3, 3), but not larger than the window itself.

Padding

* When performing the sliding window computation, there is a question as to what to do at the boundaries of the input. Staying entirely inside the input image means the window will never sit squarely over these boundary pixels like it does for every other pixel in the input. Since we aren't treating all the pixels exactly the same, could there be a problem?
* What the convolution does with these boundary values is determined by its padding parameter
* In TensorFlow, you have two choices:

either padding='same' or padding='valid'. There are trade-offs with each.

* + When we set padding='valid', the convolution window will stay entirely inside the input. The drawback is that the output shrinks (loses pixels), and shrinks more for larger kernels. This will limit the number of layers the network can contain, especially when inputs are small in size.
  + The alternative is to use padding='same'. The trick here is to **pad** the input with 0's around its borders, using just enough 0's to make the size of the output the same as the size of the input. This can have the effect however of diluting the influence of pixels at the borders. The animation below shows a sliding window with 'same' padding.

**Custom Convnets**

|  |
| --- |
| import os, warnings  import matplotlib.pyplot as plt  from matplotlib import gridspec  import numpy as np  import tensorflow as tf  from tensorflow.keras.preprocessing import image\_dataset\_from\_directory  **Load Data**  *# Load training and validation sets*  ds\_train\_ = image\_dataset\_from\_directory(  '../input/car-or-truck/train',  labels='inferred',  label\_mode='binary',  image\_size=[128, 128],  interpolation='nearest',  batch\_size=64,  shuffle=True,  )  ds\_valid\_ = image\_dataset\_from\_directory(  '../input/car-or-truck/valid',  labels='inferred',  label\_mode='binary',  image\_size=[128, 128],  interpolation='nearest',  batch\_size=64,  shuffle=False,  )  **Data Pipelines**  def convert\_to\_float(image, label):  image = tf.image.convert\_image\_dtype(image, dtype=tf.float32)  return image, label  AUTOTUNE = tf.data.experimental.AUTOTUNE  ds\_train = (  ds\_train\_  .map(convert\_to\_float)  .cache()  .prefetch(buffer\_size=AUTOTUNE)  )  ds\_valid = (  ds\_valid\_  .map(convert\_to\_float)  .cache()  .prefetch(buffer\_size=AUTOTUNE)  )  **Build Convnets**  **Convolutional Block having convolution and pooling**  from tensorflow import keras  from tensorflow.keras import layers  model = keras.Sequential([  *# First Convolutional Block*  layers.Conv2D(filters=32, kernel\_size=5, activation="relu", padding='same',  *# give the input dimensions in the first layer*  *# [height, width, color channels(RGB)]*  input\_shape=[128, 128, 3]),  layers.MaxPool2D(),  *# Second Convolutional Block*  layers.Conv2D(filters=64, kernel\_size=3, activation="relu", padding='same'),  layers.MaxPool2D(),  *# Third Convolutional Block*  layers.Conv2D(filters=128, kernel\_size=3, activation="relu", padding='same'),  layers.MaxPool2D(),  **Dense layers with flatten**  *# Classifier Head*  layers.Flatten(),  layers.Dense(units=6, activation="relu"),  layers.Dense(units=1, activation="sigmoid"),  ])  **Train Convnets**  model.compile(  optimizer=tf.keras.optimizers.Adam(epsilon=0.01),  loss='binary\_crossentropy',  metrics=['binary\_accuracy']  )  history = model.fit(  ds\_train,  validation\_data=ds\_valid,  epochs=40,  verbose=0,  )  **Show results**  import pandas as pd  history\_frame = pd.DataFrame(history.history)  history\_frame.loc[:, ['loss', 'val\_loss']].plot()  history\_frame.loc[:, ['binary\_accuracy', 'val\_binary\_accuracy']].plot(); |

**Data augmentation**

Data augmentation is a technique to artificially increase the size of a dataset by applying various transformations to the existing data. The goal is to enhance the model's generalization ability and improve its robustness by exposing it to a wider range of variations in the input data.

* **common data augmentation techniques**;
* Image Augmentation:
  + Rotation: Rotating images by a certain degree.
  + Flip: Flipping images horizontally or vertically.
  + Zoom: Zooming in or out on an image.
  + Shift: Shifting the image horizontally or vertically.
  + Shear: Applying shearing transformations to the image.
* Text Augmentation:
  + Synonym Replacement: Replacing certain words with their synonyms.
  + Random Insertion: Inserting random words into the text.
  + Random Deletion: Removing random words from the text.
  + Random Swap: Swapping the positions of two words in the text.
* Audio Augmentation:
  + Pitch Shifting: Changing the pitch of audio signals.
  + Time Stretching: Altering the speed of audio signals.
  + Background Noise Addition: Introducing background noise to the audio.
* Data Augmentation in Time Series:
  + Time Warping: Distorting the time axis of a time series.
  + Jittering: Adding random noise to the time series data.
  + Scaling: Rescaling the amplitude of the time series.
* **Benefits of data augmentation**
* **Increased Diversity:** By applying various transformations, the model is exposed to a more diverse set of examples, helping it learn to be invariant to certain changes in the input data.
* **Reduced Overfitting:** Data augmentation can act as a regularization technique, preventing the model from memorizing the training data and improving its ability to generalize to unseen examples.
* **Improved Robustness:** Models trained with augmented data are often more robust and perform better on real-world data, especially when the test data contains variations not present in the original training set.

# **Kaggle: Time Series**

**Time series**

* **Forecasting**
* focuses on the application of modern machine learning methods to time series data with the goal of producing the most accurate predictions
* you'll know how to:
  + engineer features to model the major time series components (*trends*, *seasons*, and *cycles*),
  + visualize time series with many kinds of *time series plots*,
  + create forecasting *hybrids* that combine the strengths of complementary models, and
  + adapt machine learning methods to a variety of forecasting tasks.

# **What is a Time Series?**

* A time series is a sequence of data points ordered by time. In other words, it is a set of observations or measurements taken at successive points in time.
* The basic object of forecasting is the **time series**, which is a set of observations recorded over time

Key characteristics of time series data include:

1. **Temporal Order:** The data points are ordered chronologically, and the order of observations matters.
2. **Equally Spaced Intervals (usually):** In many cases, the time intervals between consecutive data points are uniform, such as hourly, daily, monthly, etc. However, this is not a strict requirement.

**Linear Regression with time series**

* The **linear regression** algorithm learns how to make a weighted sum from its input features. For two features, we would have:

target = weight\_1 \* feature\_1 + weight\_2 \* feature\_2 + bias

* During training, the regression algorithm learns values for the parameters weight\_1, weight\_2, and bias that best fit the target. The weights are also called regression coefficients and the bias is also called the intercept because it tells you where the graph of this function crosses the y-axis.
* Use two features unique to time series: lags and time steps.
  + **Time step features**

features we can derive directly from the time index. The most basic time-step feature is the **time dummy**, which counts off time steps in the series from beginning to end.

|  |  |  |
| --- | --- | --- |
|  | Hardcover | Time |
| Date |  |  |
| 2000-04-01 | 139 | 0 |
| 2000-04-02 | 128 | 1 |
| 2000-04-03 | 172 | 2 |
| 2000-04-04 | 139 | 3 |
| 2000-04-05 | 191 | 4 |

Linear regression with the time dummy produces the model:

target = weight \* time + bias

The time dummy then lets us fit curves to time series in a time plot, where Time forms the x-axis.

라인, 텍스트, 도표, 그래프이(가) 표시된 사진

자동 생성된 설명

Time-step features let you model **time dependence**. A series is time dependent if its values can be predicted from the time they occurred.

* + **Lag features**

To make a **lag feature** we shift the observations of the target series so that they appear to have occured later in time. Here we've created a 1-step lag feature, though shifting by multiple steps is possible too.

|  |  |  |
| --- | --- | --- |
|  | Hardcover | Lag\_1 |
| Date |  |  |
| 2000-04-01 | 139 | NaN |
| 2000-04-02 | 128 | 139.0 |
| 2000-04-03 | 172 | 128.0 |
| 2000-04-04 | 139 | 172.0 |
| 2000-04-05 | 191 | 139.0 |

Linear regression with a lag feature produces the model:

target = weight \* lag + bias

So lag features let us fit curves to lag plots where each observation in a series is plotted against the previous observation.

More generally, lag features let you model **serial dependence**. A time series has serial dependence when an observation can be predicted from previous observations.

* + Adapting machine learning algorithms to time series problems is largely about feature engineering with the time index and lags.

**Trend**

**What is Trend?**

The **trend** component of a time series represents a persistent, long-term change in the mean of the series. The trend is the slowest-moving part of a series, the part representing the largest time scale of importance.

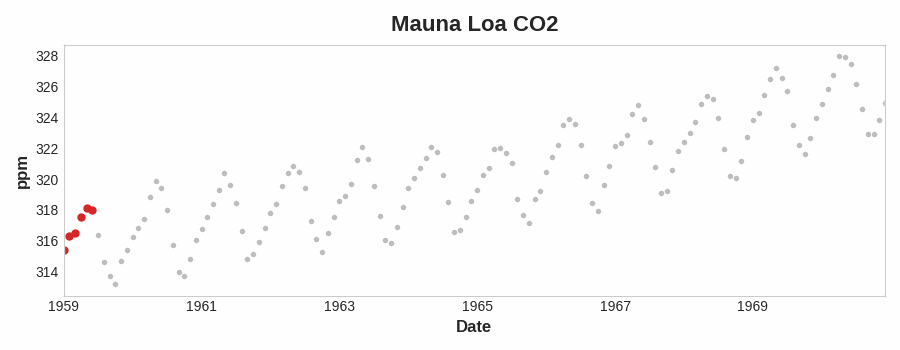
텍스트, 도표, 라인, 폰트이(가) 표시된 사진

자동 생성된 설명

*Trend patterns in four time series.*

# **Moving Average Plots**

To see what kind of trend a time series might have, we can use a **moving average plot**. To compute a moving average of a time series, we compute the average of the values within a sliding window of some defined width. Each point on the graph represents the average of all the values in the series that fall within the window on either side. The idea is to smooth out any short-term fluctuations in the series so that only long-term changes remain.



*A moving average plot illustrating a linear trend. Each point on the curve (blue) is the average of the points (red) within a window of size 12.*

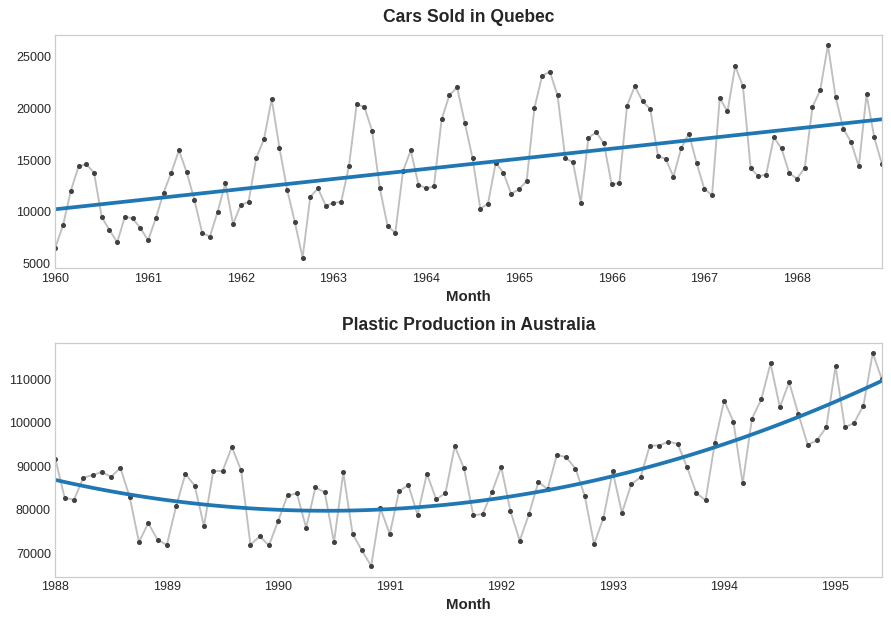
# **Engineering Trend**

Once we've identified the shape of the trend, we can attempt to model it using a time-step feature. We've already seen how using the time dummy itself will model a linear trend:

target = a \* time + b

We can fit many other kinds of trend through transformations of the time dummy. If the trend appears to be quadratic (a parabola), we just need to add the square of the time dummy to the feature set, giving us:

target = a \* time \*\* 2 + b \* time + c



***Top:****Series with a linear trend.****Below:****Series with a quadratic trend.*

If you haven't seen the trick before, you may not have realized that linear regression can fit curves other than lines. The idea is that if you can provide curves of the appropriate shape as features, then linear regression can learn how to combine them in the way that best fits the target.

# **Seasonality**

# **What is Seasonality?**

A time series exhibits **seasonality** whenever there is a regular, periodic change in the mean of the series. Seasonal changes generally follow the clock and calendar -- repetitions over a day, a week, or a year are common. Seasonality is often driven by the cycles of the natural world over days and years or by conventions of social behavior surrounding dates and times.

텍스트, 도표이(가) 표시된 사진

자동 생성된 설명*Seasonal patterns in four time series.*

Two kinds of features that model seasonality

* first kind, **indicators**, is best for a season with few observations, like a weekly season of daily observations
* second kind, **Fourier features**, is best for a season with many observations, like an annual season of daily observations.

# **Seasonal Plots and Seasonal Indicators**

**seasonal plot**

* **seasonal plot** to discover seasonal patterns.
* A seasonal plot shows segments of the time series plotted against some common period, the period being the "season" you want to observe.   
  라인, 그래프, 도표이(가) 표시된 사진

  자동 생성된 설명

*There is a clear weekly seasonal pattern in this series, higher on weekdays and falling towards the weekend.*

### Seasonal indicators

* **Seasonal indicators** are binary features that represent seasonal differences in the level of a time series
* Seasonal indicators are what you get if you treat a seasonal period as a categorical feature and apply one-hot encoding.
* By one-hot encoding days of the week, we get weekly seasonal indicators. Creating weekly indicators for the Trigonometry series will then give us six new "dummy" features. (Linear regression works best if you drop one of the indicators; we chose Monday in the frame below.)

| Date | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday |
| --- | --- | --- | --- | --- | --- | --- |
| 2016-01-04 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 2016-01-05 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 2016-01-06 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| 2016-01-07 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 | 0.0 |
| 2016-01-08 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 | 0.0 |
| 2016-01-09 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 | 0.0 |
| 2016-01-10 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 1.0 |
| 2016-01-11 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 | 0.0 |
| ... | ... | ... | ... | ... | ... | ... |

* Adding seasonal indicators to the training data helps models distinguish means within a seasonal period:

라인, 그래프, 스크린샷, 도표이(가) 표시된 사진

자동 생성된 설명

*Ordinary linear regression learns the mean values at each time in the season.*

# **Fourier Features and the Periodogram**

* Fourier features are better suited for long seasons over many observations where indicators would be impractical. Instead of creating a feature for each date, Fourier features try to capture the overall shape of the seasonal curve with just a few features.

스크린샷, 그래프, 라인, 도표이(가) 표시된 사진

자동 생성된 설명

*Annual seasonality in the*Wiki Trigonometry*series;* Notice the repetitions of various frequencies: a long up-and-down movement three times a year, short weekly movements 52 times a year, and perhaps others.

If we add a set of these sine / cosine curves to our training data, the linear regression algorithm will figure out the weights that will fit the seasonal component in the target series. The figure illustrates how linear regression used four Fourier pairs to model the annual seasonality in the *Wiki Trigonometry* series.

텍스트, 그래프, 라인, 도표이(가) 표시된 사진

자동 생성된 설명

**Top:**Curves for four Fourier pairs, a sum of sine and cosine with regression coefficients. Each curve models a different frequency. **Bottom:**The sum of these curves approximates the seasonal pattern.

# Kaggle: Intro to SQL

# **Getting Started With SQL and BigQuery**

Learn the workflow for handling big datasets with BigQuery and SQL

**BigQuery**, a web service that lets you apply SQL to huge datasets.

# **Your first BigQuery commands**