My ML studies

Babak Poursartip

July 29, 2023

Contents

Chapter 1

Introduction to ML

1.1 General

- Supervised learning (SL): functional approximation from labeled data.
- Unsupervised learning (UL): functional description. Learning from unlabeled data.
- Reinforcement learning (RL): Learning from delayed reward.

Chapter 2

Supervised learning (SL)

There are two types of supervised learning:

- classification: taking some inputs and mapping it to some discrete labels. (true of false; male or female; SUV or sedan or truck; class1, class2, class3.)
- regression: returns continuous valued function. Mapping from input to some **real number**. Something like age is more like discrete number, so, it is also classification.

2.1 Terms

- Instances: Inputs, such as pictures, pixels, etc.
- Concept: A set of *functions* that maps inputs to outputs.
- Target concept: The actual function that can map inputs to outputs. This is the actual answer.
- Hypothesis class: Set of all the functions that we are going to think about.
- Sample (Training set): A set of instances with correct labels.
- Candidate: The best approximation of the target concept.

- Testing set: A set of instances with correct labels that was not visible to the learning algorithm during the training phase. It's used to determine the algorithm performance on novel data. we can argue that we learned something by memorization, but indeed, we just memorized the concepts. What we want is generalization.

2.2 SL: Decision tree

2.2.1 Introduction

Decision tree is a classification learning in the form of a sequence of decisions based on the attributes/features/questions (which forms the nodes) applied to every instance to assign it to a specific class. Answers to the questions would be the edges of the trees. For example, deciding to eat at a restaurant or not, or classification of vehicles into sedans, SUVs, trucks, etc.

2.2.2 Representation of the decision tree:

A decision tree is a structure of nodes, edges and leaves that can be used to *represent* the data. We always start at the root of the tree, otherwise we don't look at any other attribute in the tree.

- Nodes represent attributes where you ask a question about it. (vehicle length, vehicle height, number of doors, etc.).
- Edges represent values/answers, a specific value for each attribute/question.
- Leaves represent the output (or final answer). For example, vehicle's type.

2.2.3 Algorithm to build a decision tree

Algorithm means how to create the decision tree. A decision tree *algorithm* is a sequence of steps that will lead you to the desired output. To form a decision tree, we need to come up with the attributes that can split the space, roughly in half.

- Pick the best attribute (the one that can split the data roughly in half). If this attribute added no valuable information (not a good split), it might cause overfitting.

- Ask a question about this attribute.
- Follow the correct answer path.
- Loop back to (1) till you narrow down the possibilities to one answer (the output).

Note that the usefulness of a question depends upon the answers we got to previous questions.

Purity: means all the samples are of the same kind. This is the ideal case for the leaf nodes. We have impurity if it is still a mix of various categories.

2.2.4 Expressiveness

- Decision trees can basically express any function using the input attributes.
- For Boolean functions (AND, OR, XOR, etc.), the truth table row will be translated into a path to a leaf.
- How many decision trees we can generate for a specific function/problem? For n boolean attributes with boolean output, the decision tree hypothesis space is very huge ($2^{(2^n)}$). (Babak note: this number of decision tree is not really distinct, see my notebook). This is why we need to design algorithms to efficiently search the hypothesis space for the best decision tree.

Decision trees with AND and OR are linear (they need linear number of nodes-ANY type problems). But XOR is a hard problem (requires 2^n nodes-PARITY type of problems).

2.2.5 ID3 algorithm to create a decision tree

We need to find the best attributes for each node, to narrow down the space with each question. ID3 (Inducing Decision Tree (3)) builds decision trees using a top-down, greedy approach. The greedy part of the approach comes from the fact that it will decide which attribute should be at the root of the tree by looking just one move ahead. It compares all available attributes to find which one classifies the data the best, but it doesn't look ahead (or behind) at other attributes to see which combinations of them classify the

data the best. ID3 algorithm returns an optimal decision tree, but as the size of the training data and the number of attributes increase, it becomes more likely that running ID3 on it will return a suboptimal decision tree.

To train a decision tree, answer these questions:

- How to find the best attribute/feature for the decision tree at each level?
- When do you stop splitting?

Pseudo code:

1. Pick the best attribute A (the definitions of best attribute comes later). To select this attribute, a statistical test is used to determine for each attribute how well it alone classifies the training examples.
2. Split the data into subsets that correspond to the possible values of the best attribute.
3. Assign A as a decision attribute for a node.
4. For each value of A, create a descendant node.
5. Sort training examples to leaves.
6. Stopping criteria (see down there): f examples perfectly classified (means all data point are of the same class) or there is no more attributes – Stop splitting – else –Iterate over leaves

How to find the best attribute/feature for the decision tree at each level:
There are a couple of options. The most common method is called information gain: a measure that expresses how well an attribute splits the data into groups based on classification (maximize purity or minimize impurity). For example, an attribution returns *low information gain*, if it divides samples to two classes with an even yes and no, but the information gain is high, if each class has more of yeses or nos.

It quantifies the reduction in randomness (Entropy) over the labels we have with a set of data, based upon knowing the value of a particular attribute. A mathematical way to measure the gain, is entropy. It measures the homogeneity of a data set S's classifications. *Entropy ranges from 0, which means that all of the classifications in the data set are the same (either yes or no), to log_2 of the number of different classifications, which means that the classifications are equally distributed within the data set.* For a binary classification the entropy is only $log_2 1 = 1$ (if yes and no samples are equally divided). entropy is calculated as follows:

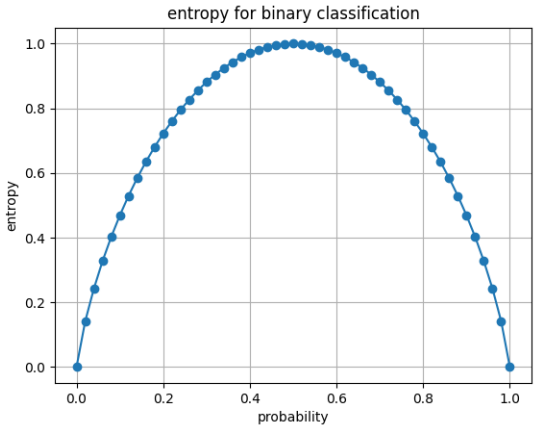
$$Entropy(S) = \sum_{i=1}^{c} -p_i \, log_2(p_i) \tag{2.1}$$

where:

- c corresponds to the number of different classifications (either for the attribute or the final output)
- p_i corresponds to the proportion of the data with the classification i

Here is the plot of entropy for a binary classifier, as the proportional of yes and no samples change:

Figure 2.1: Entropy



Information gain measures the reduction in entropy that results from partitioning the data on an attribute A, which is another way of saying that

it represents how effective an attribute is at classifying the data. Given a set of training data S and an attribute A, the formula for information gain is:

$$Gain(S, A) = Entropy(S) - \sum_{v} \frac{|S_v|}{|S|} Entropy(S_v) \tag{2.2}$$

where v is all the possible values of attribute the attribute (for example, sunny, cloudy, rainy), and S_v is the total number of samples corresponding to this value of attribute (for example sunny).

For *if I play tennis game* (yes and no is the final output) with attributes weather (sunny, cloudy, rainy), temperature (hot, cold, mild), humidity (normal, high), first, we calculate the entropy of the entire set (Entropy(S)) (wrt the the final output classification, yes and no). For each attribute, we calculate the the entropy corresponding to each value of entropy, then we calculate the final entropy.

We want to maximize information gain, so we want the entropies of the partitioned data to be as low as possible, which explains why attributes that exhibit high information gain split training data into relatively heterogeneous groups.

As the size of the training data and the number of attributes increase, it becomes likelier that running ID3 on it will return a sub-optimal decision tree.

The other option we can use instead of Entropy is the **Gini function.**

When do you stop splitting?

- when a node is 100% pure
- when splitting a node will result in the tree exceeding a maximum depth (to avoid over-fitting)
- when improvement in purity score (information gain) are below a threshold
- when the number of examples in a node is below a threshold

2.2.6 Inductive bias

Two types of bias for classifiers:

- Restriction Bias: Describes the hypothesis set space (H) that we will consider for learning the tree. Complete hypothesis space (low Restriction Bias). Instead of looking at infinitely many functions, we only consider those that can be represented by the decision tree.
- Preference Bias: Describes the subset of hypothesis n ($n \in H$), from the hypothesis set space H, that we prefer.

Inductive bias of ID3 algorithm:

- it prefers the decision tree with good splits at top (even if a bad split generates the same outcome). This is because ID3 is greedy using info gain.
- it prefers correct outcome over incorrect because ID3 repeats until the labels are correctly classified.
- it prefers shorter trees (comes from the fact that we use good splits from the top).

2.2.7 Extending ID3 algorithm-other consideration

For continuous values attributes/features, we can classify the attributes based on thresholds (that exists in the data set). Without threshhold, the continuous data attribute provides the most information gain, while giving a decision tree that is not generalize well. To find the threshold, we can plot the values and decide based on a threshold the returns the best outcome, by calculating the information gain values for each threshold. One way to select the threshold is to take all the values that are mid points between the sorted list of training. For example, for 10 training samples, we need to test the info gain for 9 values, for this specific feature. We can even have multiple threshold for one feature (by creating multiple features).

Does it make sense to repeat an attribute along a path in a decision tree? No, it does not make sense for discrete values, but for continuous values, it makes sense, because indeed we are asking a different question (for a different range).

If the data for some attributes is missing, see 5.8.

Over-fitting + When do we stop? When everything classified correctly or if there is no more attribute. What if we have noise in data (different

answer for the same instance)? Causes an infinite loop. What would be the stopping criterion, then? We want generalization, and we should avoid **over-fitting**. If the tree is too big/complicated, there is a chance that it over-fits. There are two popular approaches to avoid over-fitting in decision trees: stop growing the tree before it becomes too large or prune the tree after it becomes too large. Typically, a limit to a decision tree's growth will be specified in terms of the maximum number of layers, or depth, it's allowed to have.

One option to avoid over-fitting is that we can grow the trees, and use cross-validation to prevent over-fitting. The data available to train the decision tree will be split into a training set and validation/test set and we keep growing the tree with various maximum depths will be created based on the training set and then test it against the test set. The best will be selected (when the error grows, we stop growing the tree).

Pruning the tree, on the other hand, involves testing the original tree against pruned versions of it. Leaf nodes are taken away from the tree as long as the pruned tree performs better against test data than the larger tree.

One-hot encoding: If there are multiple categories for a feature, we can use one-hot encoding to convert each category to a binary feature. Thus, jf a feature can take k values, one-hot encoding creates k binary features. There are libraries in the sklearn library for this task. See the details in the 5.1.

Regression: how to adapt decision trees for regression type of problems (continuous outputs)? Decision trees as we've defined them here don't transfer directly to regression problems. We no longer have a useful notion of information gain, so our approach at attribute sorting falls through. Instead, we can rely on purely statistical methods (like variance and correlation) to determine how important an attribute is. For leaves, too, we can do averages, local linear fit, or a host of other approaches that mathematically generalize with no regard for the meaning of the data.

2.2.8 Regression with Decision tree

We can use decision trees for classification (DecisionTreeClassifier) or for regression (DecisionTreeRegressor). Training a regressor is not any different from the classifier, but for the leaf node, we use the average of the continuous values of all the samples in that node.

The major difference here is that how you choose to split the data at the node. Instead of maximizing the entropy, we try to minimize the variance of

the values for the continuous feature. Thus, we calculate the variance for the values at each leaf separated by the feature, then we use a weighted method to combine it.

$$\begin{aligned}\text{minimize the variance (means choose the feature with largest value)} = \\ \text{variance of all samples at the root} - \\ \Bigg[\\ \frac{\text{num of samples in the left leaf}}{\text{num of total samples}} \times \text{variance of left leaf} + \\ \frac{\text{num of samples in the right leaf}}{\text{num of total samples}} \times \text{variance of right leaf} \\ \Bigg] \end{aligned} \tag{2.3}$$

For example, if a feature divides 10 samples into two leaf nodes with 4 and 6 samples in each leaf node, then, the weighted average of variance is $\frac{4}{10} \times$ variance of left leaf $+ \frac{6}{10} \times$ variance of right leaf.

2.2.9 Changing the information gain formula is another option

The information gain formula used by the ID3 algorithm treats all of the variables the same, regardless of their distribution and their importance. This is a problem when it comes to continuous variables or discrete variables with many possible values because training examples may be few and far between for each possible value, which leads to low entropy and high information gain by virtue of splitting the data into small subsets, but results in a decision tree that might not generalize well.

One successful approach to deal with this is using a formula called GainRatio in the place of information gain. GainRatio tries to correct for information gain's natural bias toward attributes with many possible values by adding a denominator to information gain called SplitInformation. SplitInformation attempts to measure how broadly partitioned an attribute is and how evenly distributed those partitions are. In general, the SplitInformation of an attribute with n equally distributed values is $log_2 n$. These relatively large denominators significantly affect an attribute's chances of being the best attribute after an iteration of the ID3 algorithm and help to avoid choices

that perform particularly well on the training data but not so well outside of it.

$$GainRatio(S, A) = \frac{Gain(S, A)}{SplitInformation(S, A)}$$

$$SplitInformation(S, A) = -\sum_{i=1}^{c} \frac{|S_i|}{S} \ log_2 \frac{|S_i|}{|S|}$$

2.2.10 Advantages and Disadvantages of Decision Trees (copied from a text)

Advantages:

- Simple to understand and to interpret.
- Requires little data preparation. Other techniques often require data normalization, dummy variables need to be created and blank values to be removed.
- The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the tree.
- Able to handle both numerical and categorical data. Other techniques are usually specialized in analyzing datasets that have only one type of variable.
- Able to handle multi-output problems.
- Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by Boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.
- Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.
- Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.

Disadvantages:

- Can create over-complex trees that do not generalize the data well. This is called overfitting. Mechanisms such as pruning (setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree) are necessary to avoid this problem.
- Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble (random forest/XGBoost).
- The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.
- There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems.
- Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

2.2.11 When to use a decision tree

Decision trees and tree ensembles:

- Works well on tabular/structured data
- Not recommended for unstructured data such as text, image, audio. Use NN here.
- Decision tree is fast in training.
- Small decision trees may be human interpretable.

Neural Networks

- Works well on all types of data, including tabular/structured and unstructured data.
- May be slower than the decision tree for training.
- Works with transfer learning.
- When building a system of multiple models working together, it might be easier to string together multiple neural networks.

2.2.12 Random forest (Tree ensembles)

A single decision tree can be highly sensitive to small changes in the data. The random forest uses many trees, and it makes a prediction by averaging the predictions of each component tree. It generally has much better predictive accuracy than a single decision tree and it works well with default parameters. If you keep modeling, you can learn more models with even better performance, but many of those are sensitive to getting the right parameters.

To build a tree ensemble (random forest), we use a technique called *Sampling With Replacement*. We are going to construct random sets of the same size of the original set, that are slightly different from the original set, by randomly selecting instances from the set (so, some of the training instances might be repeated in each set). Then, we train a decision tree using this data set. We continue this process to build B decision trees (Bagged decision tree). To randomize the feature choice further, at each node, when choosing a feature to use to split, if n features are available, pick a random subset of $k < n$ features and allow the algorithm to only choose from that subset of features. If n is large $\sqrt{n}$ is valid.

Pseudo code:
Given training set of size m
For b=1 to B:

- use sampling with replacement to create a new training set of size m
- train a decision tree on the new dataset

2.2.13 Boosted tree (XGBoost)

This is the most commonly used method of decision tree on samples. It requires a modification to the Bagged decision tree from the previous section 2.2.12. Basically, it is focusing on the part the data that the tree is not doing well.

Pseudo code:
Given training set of size m
For b=1 to B:

- use sampling with replacement to create a new training set of size m
- Instead of picking from all examples with equal $1/m$ probability, make it more likely to pick misclassified examples from previously trained trees.
- train a decision tree on the new dataset

2.3 SL: Regression and classification

Regression is the mapping of continuous inputs to continuous outputs, as opposed to the classification where we had mapping from discrete input to discrete output. Regression is similar to function approximation. Back then, regression meant falling back to the average/mean, but now, we mean, using functional form to approximate a bunch of points.

We use regression because the data itself is not accurate and has noise. If the data is exact, we need to use interpolation and spline methods, to exactly fit the line with the data.

2.3.1 Linear regression

Finding a linear function/relationship between the input data and the output. It might not be a good fit though. We want to find a line (linear function) such that it minimizes the deviation, or to be more precise, the squared error between the data points and the line. This is least square regression. Basically, this is fitting data with a curve, so that we can predict the results based on the model. Here we do not try to intersect every point, rather the curve is designed to follow the pattern of points. (The other approach is interpolation that we try to pass the line/curve through each data point.)

In summary, this is an approximate function that fits the shape or general trend of the data w/o necessarily matching the individual points. To find the line:

$$\begin{aligned} \text{data points: } & \{(x_i, y_i) : i = 1...n\} \\ \text{find a and b for the linear fit: } & y_i = ax_i + b \\ \text{such that minimizes error: } & e = \sum_{i=1}^{n} (y_i - (ax_i + b))^2 \end{aligned} \tag{2.4}$$

To find the the two unknowns of the linear fit, a and b, take the derivative of the error with respect to a and b and set it zero. That would be a system of equations with two unknowns. After solving the equation you have the fit. To have a nonlinear fit, we can use a polynomial model:

2.3.2 Non-linear regression

$$f(x_i) = a_0 + a_1 x_i + a_2 x_i^2 + ... \tag{2.5}$$

The unknowns are $a_0, a_1, ...$ See my class notes for more details. To better fit the regression model, we can use higher order polynomial, but it leads to over-fitting.

2.3.3 Error

Training data has errors due to various reasons such as reading errors, sensor errors, transcription error, maliciously given data, etc. This is the reason we are using the regression not the interpolation.

To calculate the parameters of the model, we need to form the cost function (see 5.5) and minimize the error using an optimization technique, such as gradient descent **??**.

2.3.4 Other cases

For the regression, we can have multiple independent variables/features for the regression.

Using discrete numbers is also possible for regression, however this is no trivial. We need to encode the attributes. For example, by enumerating the categories (giving a number to each possible value of this feature). It is a

bit misleading because a correlation between the ordering of the enumeration and the its value can be interpolated. The other option is represent the value of the feature as a Boolean.

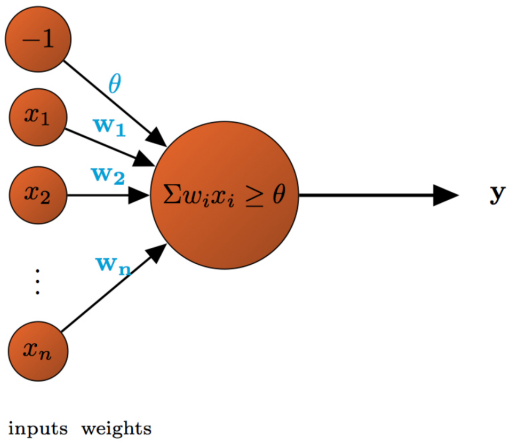
2.4 SL: Neural network

2.4.1 Perceptron

simplest model of a brain cell. The Perceptron calculates the sum of products of the inputs X and their corresponding weights W, and then compare the result with an activation threshold. If the sum is greater than or equal the firing threshold θ, the perceptron returns one, otherwise, it is zero.

$$\sum_{i=1}^{k} X_i W_i \geq \theta \Rightarrow y = 1 \text{ otherwise } y = 0 \tag{2.6}$$

Figure 2.2: Perceptron



2.5 SL: Instance based learning

2.6 SL: Ensemble B&B

Ensemble Learning is in general the process of combining some simple rules into a more complex rule that can generalize well.

It works by dividing the data into smaller subsets, learn over each individual subset to come up with a rule, then combine all these rules in a single more complex rule. If we looked at the whole data, it would be hard to come up with simple rules.

2.7 SL: Kernel methods and SVMs

2.8 SL: Comp Learning Theory

2.9 SL: VC dimensions

2.10 SL: Bayesian Learning

2.11 SL: Bayesian inference

Chapter 3

Unsupervised learning (UL)

3.1 Random optimization

3.2 Clustering

Chapter 4

Reinforced Learning (RL)

4.1 Markov Decision Process

Chapter 5

Miscellaneous topics

5.1 One hot encoding for categorical features

source1 and source2

A categorical feature takes only a few limited number of values. There are tree options to deal with the categorical features:

- Drop the feature: this option works only if the feature does not provide any useful information.
- Ordinal encoding: assigning a value to each category **(ordinal variables)**. This method works for tree-based (decision tree) models.
- One-hot encoding: creating one feature for each category, and assigning binary values. This method works particularly well if there is no clear ordering in the categorical data. We refer to categorical variables without an intrinsic ranking as **nominal variables**. One-hot encoding generally does not perform well if the categorical variable takes on a large number of values (15 and more). We refer to the number of unique entries of a categorical variable as the cardinality of that categorical variable. High cardinality columns can either be dropped from the dataset, or we can use ordinal encoding.

A one hot encoding is a representation of categorical features as binary vectors. This first requires that the categorical values be mapped to integer values. Then, each integer value is represented as a binary vector that is all zero values except the index of the integer, which is marked with a 1.

Example: assume we have a sequence of labels with the values *red* and *green*. We can assign *red* an integer value of 0 and *green* the integer value of 1. As long as we always assign these numbers to these labels, this is called an integer encoding. Consistency is important so that we can invert the encoding later and get labels back from integer values, such as in the case of making a prediction. Next, we can create a binary vector to represent each integer value. The vector will have a length of 2 for the 2 possible integer values. The *red* label encoded as a 0 will be represented with a binary vector [1, 0] where the zeroth index is marked with a value of 1. In turn, the *green* label encoded as a 1 will be represented with a binary vector [0, 1] where the first index is marked with a value of 1. If we had the sequence: *red, red, green*, we could represent it with the integer encoding: 0, 0, 1. And the one hot encoding of: [1, 0], [1, 0], and [0, 1].

Why Use a One Hot Encoding? A one hot encoding allows the representation of categorical data to be more expressive. Many machine learning algorithms cannot work with categorical data directly. The categories must be converted into numbers. This is required for both input and output variables that are categorical. We could use an integer encoding directly, rescaled where needed. This may work for problems where there is a natural ordinal relationship between the categories, and in turn the integer values, such as labels for temperature *cold, warm, and hot*.

There may be problems when there is no ordinal relationship and allowing the representation to lean on any such relationship might be damaging to learning to solve the problem. An example might be the labels dog and cat

In these cases, we would like to give the network more expressive power to learn a probability-like number for each possible label value. This can help in both making the problem easier for the network to model. When a one hot encoding is used for the output variable, it may offer a more nuanced set of predictions than a single label.

We can use libraries such is scikit-learn and keras to encode a categorical feature for ML.

5.2 Cross validation

The goal is always to generalize the model to fit the real world data. The test set is just a representation of the data that we may see in future. The model should be complex enough to model the training set without over-fitting.

The data points that we are using for training are assumed to be Independent and Identically Distributed (IID), which means that there's no inherent difference between training, testing and real-world data and all the data is coming from the same source. This is the *Fundamental Assumption of Supervised Learning.*

A fixed validation set leaves some random chance in determining model scores. That is, a model might do well on the fixed validation set, even if it would be inaccurate on a different set. 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! The idea is that we select part of the training set as the test set, not touching the actual test during the training set. This is indeed the cross validation set.

Cross Validation steps:

- Randomly partition the training data into k folds of equal size.
- Train the model on all the folds except for one (k-1).
- Validate the model's performance using the fold that was not used in training.
- Repeat steps 2 and 3 using different combinations of these folds.
- Average the error in predicting the polynomial trained on the training folds.

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). 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. Note that we no longer need to keep track of separate training and validation sets.

To use cross validation, it is more convenient to implement a pipeline.

Observation for using cross validation with linear regression:

- Average cross validation error is higher than training error for lower orders .
- Average cross validation error decreases as the polynomial order increases (same for training error).
- After the golden degree, increasing the degree of polynomial will result in overfitting, and the Cross Validation error will start to increase.

5.3 F1 score, precision, and recall

Refer to this source.

Precision (P) is defined as the number of true positives (T_p) over the number of true positives plus the number of false positives (F_p): $P = \frac{T_p}{T_p+F_p}$.

Recall (R) is defined as the number of true positives (T_p) over the number of true positives plus the number of false negatives (F_n): $R = \frac{T_p}{T_p+F_n}$.

These quantities are also related to the ($F1$) score, which is defined as the harmonic mean of precision and recall: $F1 = 2\frac{R*P}{P+R}$.

The F1 score can be interpreted as a harmonic mean of the precision and recall, where an F1 score reaches its best value at 1 and worst score at 0.

5.4 Overfitting vs underfitting

This is a phenomenon called overfitting, where a model matches the training data almost perfectly, but does poorly in validation and other new data. On the flip side, if we make our tree very shallow, it doesn't divide up the houses into very distinct groups.

At an extreme, if a tree divides houses into only 2 or 4, each group still has a wide variety of houses. Resulting predictions may be far off for most houses, even in the training data (and it will be bad in validation too for the same reason). When a model fails to capture important distinctions and

patterns in the data, so it performs poorly even in training data, that is called underfitting.

Since we care about accuracy on new data, which we estimate from our validation data, we want to find the sweet spot between underfitting and overfitting. Visually, we want the low point of the (red) validation curve in the figure below.

5.5 Cost function

Let's use the concept of multiple linear regression to discuss the cost function.

Parameters/coefficients/weights of the model (In ML, parameters of a model are the variables that you can do training in order to improve the model):

$$w_1, ..., w_n, b \equiv \overrightarrow{w}, b \tag{5.1}$$

Here is the model:

$$f_{\overrightarrow{w},b}(\overrightarrow{x}) = w_1x_1 + ... + w_nx_n + b \tag{5.2}$$

data points $\{(x_i, y_i) : i = 1...m\}$

$$\hat{y}^{(i)} = f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}) = \overrightarrow{w}\,\overrightarrow{x}^{(i)} + b \tag{5.3}$$

Problem statement: find $\overrightarrow{w}, b$ such that $\hat{y}^{(i)}$ is close to $y^{(i)}$ for all $(x^{(i)}, y^{(i)})$. $\left(\hat{y}^{(i)} - y^{(i)}\right)^2$, the error for one single sample is called the loss function.

To find $\overrightarrow{w}, b$, we form the the cost function:

find $\overrightarrow{w}, b$:

$$\text{minimize: } J(\overrightarrow{w}, b) = \frac{1}{2m}\sum_{i=1}^{m}\left(\hat{y}^{(i)} - y^{(i)}\right)^2 = \frac{1}{2m}\sum_{i=1}^{m}\left(wx^{(i)} + b - y^{(i)}\right)^2 \tag{5.4}$$

m is the to total number of data points. The extra 2 in the denominator is just for convenience. This is the squared error cost function, which is widely used. However, there are other options for the cost functions as well.

5.6 Gradient descent (steepest descent)

An optimization method to minimize any function. Depending on the initial starting point, the method may end up at a different local minimum.

Method:
For a cost function $J(\overrightarrow{w}, b)$, we want to find $w_1, w_2, ...$ to minimize J.

Algorithm:

- start with some initial $\overrightarrow{w}, b$.
- update w_j *simultaneously* until the algorithm converges, meaning you reach a local minimum, where additional iteration does not change the point. It is incorrect to use new values of w_j to update other parameters in the same iteration.

$$\begin{aligned} w_j &= w_j - \alpha \frac{\partial}{\partial w_j} J(\overrightarrow{w}, b) = w_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right) x_j^{(i)} \\ &= w_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}) - y^{(i)}\right) x_j^{(i)} \quad \text{for each j} \\ b &= b - \alpha \frac{\partial}{\partial b} J(\overrightarrow{w}, b) = b - \alpha \frac{1}{m} \sum_{i=1}^{m} \left(f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}) - y^{(i)}\right) \quad \text{for each i} \end{aligned} \tag{5.5}$$

The negative is because we want to move in the opposite direction.

α is the learning rate (step size). The choice of learning rate affects the efficiency of gradient descent. If it is too small, it converges very slowly. If it is too large (overshoot), may cause divergence and never reaches the minimum. If we are exactly at the minimum, since slope is zero, the value of the learning does not affect the results. Thus, even a fixed learning can reach the minimum.

Various types of gradient descent:

Batch gradient descent: each step of gradient descent uses all the training examples. While this batching provides computation efficiency, it can still have a long processing time for large training datasets as it still needs to store

all of the data into memory. Batch gradient descent also usually produces a stable error gradient and convergence, but sometimes that convergence point isn't the most ideal, finding the local minimum versus the global one.

Mini-batch gradient descent updates the parameters using a subset of training samples.

5.7 Model validation

Always start with the dataset documentation.

In most (though not all) applications, the relevant measure of model quality is predictive accuracy. In other words, will the model's predictions be close to what actually happens.

There are many metrics for summarizing model quality, but we'll start with one called Mean Absolute Error (also called MAE). With the MAE metric, we take the absolute value of each error. This converts each error to a positive number. We then take the average of those absolute errors. This is our measure of model quality. sklearn has a function, mean_absolute_error, to calculate MAE.

Many people make a mistake when measuring predictive accuracy. They make predictions with their training data and compare those predictions to the target values in the training data. This is not valid. The most straightforward way to validate the model is to exclude some data from the model-building process, and then use those to test the model's accuracy on data it hasn't seen before. This data is called validation data.

5.8 Data processing/cleaning: Missing Values

Before we treat missing data, we need to know **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 (like the height of the oldest child of someone who doesn't have any children) then it doesn't make sense to try and guess what it might be. These values you probably do want to keep as NaN. On the other hand, 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. There are many ways the data misses: wrong measurement, data is private, data is not available, etc.

Generally, there are three approaches deal with the missing data. These are quick and dirty approaches that probably remove some useful info or adding some noise to your dataset.

- The simplest option: drop columns or features with missing values. Unless most values in the dropped columns are missing, the model loses access to a lot of (potentially useful!) information with this approach.
- Imputation: fills in the missing values with some number. For instance, we can fill in the mean value along each column, or we can choose the most popular data for the missing item. We can also assign the probability of each value of the attribute to the missing entries. The imputed value won't be exactly right in most cases, but it usually leads to more accurate models than you would get from dropping the column entirely.
- Extending imputation: we impute the missing values, as before. And, additionally, for each column with missing entries in the original dataset, we add a new column that shows the location of the imputed entries. For example, if the size of bedroom is missing for some instances, we add a new column, missing_bedromm_size, and set it to FALSE for all valid instances and TRUE for all missing instances. This will meaningfully improve results. In other cases, it doesn't help at all.

In pandas, we can use .isnul() to detect columns with missing data. We can use .drop to drop columns. We can also use dropna to either drop columns or instances.

5.9 Data processing: scaling vs normalization

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**normalization**, you're changing the shape of the distribution of your data.

5.9.1 scaling

This means that 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) or k-nearest neighbors (KNN). With these algorithms, a change of "1" in any numeric feature is given the same importance. By scaling your variables, you can help compare different variables on equal footing.

5.9.2 normalization

Scaling just changes the range of your data. Normalization is a more radical transformation. The point of normalization is to change your observations so that they can be described as a normal distribution.

In general, you'll normalize your data if you're going to be using a machine learning or statistics technique that assumes your data is normally distributed. Some examples of these include linear discriminant analysis (LDA) and Gaussian naive Bayes. (Pro tip: any method with "Gaussian" in the name probably assumes normality.)

5.10 Ensemble methods

Ensemble methods combine the predictions of several models (e.g., several trees, in the case of random forests).

5.10.1 Gradient Boosting

Gradient boosting is a 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, 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 on the loss function to determine the parameters in this new model.)
- Finally, we add the new model to ensemble,
- repeat!

XGBoost stands for extreme gradient boosting, which is an implementation of gradient boosting with several additional features focused on performance and speed. (Scikit-learn has another version of gradient boosting, but XGBoost has some technical advantages.) scikit-learn API for XGBoost (xgboost.XGBRegressor) allows us to build and fit a model just as we would in scikit-learn. The XGBRegressor class has many tunable parameters. See the code.

5.10.2 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 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: 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. Since validation data comes from the same source as training data, the pattern will repeat itself in validation, and the model will have great validation (or cross-validation) scores. But the model will be very inaccurate when subsequently deployed in the real world, because even patients who will get pneumonia won't have received antibiotics yet when we need to make predictions about their future health.

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.

For example, imagine you run preprocessing (like fitting an imputer for missing values) before calling train_test_split(). The end result? Your model may get good validation scores, giving you great confidence in it, but perform poorly when you deploy it to make decisions.

After all, you incorporated data from the validation or test data into how you make predictions, so the may do well on that particular data even if it can't generalize to new data. This problem becomes even more subtle (and more dangerous) when you do more complex feature engineering.

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!