Machine Learning

CHAPTER 7

*How much machine learning do you **actually** need to know to land a top job in Silicon Valley or Wall Street? Probably less than you think! From coaching hundreds of data folks on the job hunt, one of the most common misconceptions we saw was candidates thinking their lack of deep learning expertise would tank their performance in data science interviews. However, the truth is that most data scientists are hired to solve business problems — not blindly throw complicated neural networks on top of dirty data. As such, a data scientist with strong business intuition can create more business value by applying linear regression in an Excel sheet than a script kiddie whose knowledge doesn't extend beyond the Keras API.*

So, unless you're interviewing for ML Engineering or research scientist roles, a solid understanding of the classical machine learning techniques covered in this chapter is all you need to ace the data science interview. However, if you are aiming for ML-heavy roles that do require advanced knowledge, this chapter will still be handy! Throughout this chapter, we frequently call attention to which topics and types of questions show up in tougher ML interviews. Plus, the 35 questions at the end of the chapter — especially the hard ones — will challenge even the most seasoned ML practitioner.

What to Expect for ML Interview Questions

When machine learning is brought up in an interview context, the problems fall into three major buckets:

- ***Conceptual questions:*** Do you have a strong theoretical ML background?
- ***Resume-driven questions:*** Have you actively applied ML before?
- ***End-to-end modeling questions:*** Can you apply ML to a hypothetical business problem?

Conceptual Questions

Conceptual questions usually center around what different machine learning terms mean and how popular machine learning techniques operate. For example, two frequently asked questions are "What is the bias-variance tradeoff?" and "How does PCA work?" To test your ability to communicate with nontechnical stakeholders, a common twist on these conceptual questions is to ask you to explain the answer as if they (the interviewer) were five years old (similar to Reddit's popular r/ELI5 subreddit).

Because many data science roles don't require hardcore machine learning knowledge, easier, straightforward questions such as these represent the vast majority of questions you'd expect during a typical interview. Being asked easier ML questions is especially the case when interviewing for a data science role that's more product and business analytics oriented, as having to build models just simply isn't part of the day-to-day work.

For ML-intensive positions like ML Engineer or Research Scientist, interviews also start with high-level easier conceptual questions but then push you to dive deeper into the details via follow-up questions. Companies do this to make sure you aren't a walking, talking ML buzzword generator. For example, as a follow-up to defining the bias-variance trade-off, you might be asked to whiteboard the math behind the concept. Instead of simply asking you how PCA (principal components analysis) works, you might also be asked about the most common pitfalls of using PCA.

Since ML interviews are so expansive in scope, if asked about a particular technique you may not be overly familiar with, it's perfectly okay to say, "I've read about it in the past. I don't have any experience with these types of techniques, but I am interested to learn more about them!"

This signals honesty and an eagerness to learn (and don't be ashamed to admit not knowing something — nobody knows all the techniques in detail! Trust us, it's better than pretending you know the techniques and then falling apart when questions are asked).

If nothing on your resume seems interesting to an interviewer, but they still want to go deep into one ML topic, they have you pick the topic. They do this by either asking "What's your favorite ML algorithm?" or "What's a model you use often and why?" Consequently, it pays to have a deep understanding of at least a single technique — something you've actually used before and that is listed on your resume.

Word of caution: don't choose something about a state-of-the-art transformer model to discuss as your favorite technique. Your details on it may be hazy, and your interviewer might not know enough about it to carry on a good conversation. You are better off picking something fundamental yet interesting (to you) so that you and your interviewer can have a meaningful discussion. For example, our answer happens to be that we both like random forests because they can handle classification or regression tasks with all kinds of input features with minimal preprocessing needed. Additionally, we both have projects on our resume to back up our interest in random forests.

Resume-Driven Questions

The next most common type of interview question for ML interviews is the resume-driven question. Resume-driven questions are often about showcasing that you have practical experience (as opposed to conceptual knowledge). As such, if you have job experience that is directly relevant, interviewers will often ask about that. If not, they'll often fall back to asking about your projects.

While anything listed on your resume is fair game to be picked apart, this is especially true for more ML-heavy roles. Because the field is so vast and continually evolving, an interviewer isn't able to assess your fit for the job by asking about some niche topic unrelated to the position at hand. For example, say you are going for a general data science role — it's not fair to ask a candidate about CNNs and their use in computer vision if they have no experience with this topic and it's not relevant to the job. But, suppose you hacked together a self-driving toy car last summer, and listed it on your resume. In that case — even though the role at hand may not require computer vision — it's totally fair game to be asked more about the neural network architecture you used, model training issues you faced, and trade-offs you made versus other techniques. Plus, in an effort to see if you know the details not just of your project, but of the greater landscape, you'd also be expected to answer questions tangentially related to the project.

End-to-End Modeling Questions

Finally, the last type of ML-related problems can expect during interviews are end-to-end modeling questions. Interviewers are testing your ability to go beyond the ML theory covered in books like *An Introduction to Statistical Learning* and actually apply what you learned to solve real-world problems. Examples of questions include "How would you match Uber drivers to riders?" and "How would you build a search autocomplete feature for Pinterest?" While these open-ended problems are an interview staple for any machine-learning-heavy role, they do also pop up during generalist data science interviews.

At the end of this chapter, we cover the end-to-end machine learning workflow, which can serve as a framework for answering these broad ML questions. We cover steps like problem definition, feature engineering, and performance metric selection — things you'd do before jumping into the various ML techniques we soon cover. To better solve these ML case study problems, we also recommend reading Chapter 11: Case Study to understand the non-ML-specific advice we offer for tackling open-ended problems.

The Math Behind Machine Learning

While the probability and statistics concepts upon which machine learning's foundation is built are fair game for interviews, you're less likely to be asked about the linear algebra and multivariable calculus concepts that underlie machine learning. There are, however, two notable exceptions: if you're interviewing for a research scientist position or for quant finance. In these cases, you may be expected to whiteboard proofs and derive formulas. For example, you could be asked to derive the least squares estimator in linear regression or explain how to calculate the principal components in PCA. Sometimes, to see how strong your first principles are, you'll be given a math problem more indirectly. For instance, you could be asked to analyze the statistical factors driving portfolio returns (which essentially boils down to explaining the math behind PCA). Regardless of the role and company, we still recommend you review the basics, since understanding them will help you grok the theoretical underpinnings of the techniques covered later in this chapter.

Linear Algebra

The main linear algebra subtopic worth touching on for interviews is eigenvalues and eigenvectors. Mechanically, for some $n \times n$ matrix A, x is an eigenvector of A if: $Ax = \lambda x$, where λ is a scalar. A

matrix can represent a linear transformation and, when applied to a vector x, results in another vector called an *eigenvector*, which has the same direction as x and is in fact x multiplied by a scaling factor λ known as an *eigenvalue*.

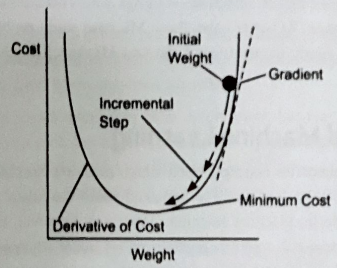
The decomposition of a square matrix into its eigenvectors is called an *eigendecomposition*. However, not all matrices are square. Non-square matrices are decomposed using a method called singular value decomposition (SVD). A matrix to which SVD is applied has a decomposition of the form: $A = U\Sigma V^T$, where U is an $m \times m$ matrix, Σ is an $m \times n$ matrix, and V is an $n \times n$ matrix.

There are many applications of linear algebra in ML, ranging from the matrix multiplications during backpropagation in neural networks, to using eigendecomposition of a covariance matrix in PCA. As such, during technical interviews for ML engineering and quantitative finance roles, you should be able to whiteboard any follow-up questions on the linear algebra concepts underlying techniques like PCA and linear regression. Other linear algebra topics you're expected to know are core building blocks like vector spaces, projections, inverses, matrix transformations, determinants, orthonormality, and diagonalization.

Gradient Descent

Machine learning is concerned with minimizing some particular objective function (most commonly known as a loss or cost function). A loss function measures how well a particular model fits a given dataset, and the lower the cost, the more desirable. Techniques to optimize the loss function are known as optimization methods.

One popular optimization method is gradient descent, which takes small steps in the direction of steepest descent for a particular objective function. It's akin to racing down a hill. To win, you always take a "next step" in the steepest direction downhill.



For convex functions, the gradient descent algorithm eventually finds the optimal point by updating the below equation until the value at the next iteration is very close to the current iteration (convergence):

$$x_{t+1} = x_t - \alpha_t \nabla f(x_t)$$

that is, it calculates the negative of the gradient of the cost function and scales that by some constant α_t, which is known as the learning rate, and then moves in that direction at each iteration of the algorithm.

Since many cost functions in machine learning can be broken down into the sum of individual functions, the gradient step can be broken down into adding separate gradients. However, this process can be computationally expensive, and the algorithm may get stuck at a local minimum or saddle

point. Therefore, we can use a version of gradient descent called *stochastic gradient descent* (SGD), which adds an element of randomness so that the gradient does not get stuck. SGD uses one data point at a time for a single step and uses a much smaller subset of data points at any given step, but is nonetheless able to obtain an unbiased estimate of the true gradient. Alternatively, we can use *batch gradient descent* (BGD), which uses a fixed, small number (a mini-batch) of data points per step.

Gradient descent and SGD are popular topics for ML interviews since they are used to optimize the training of almost all machine learning methods. Besides the usual questions on the high-level concepts and mathematical details, you may be asked when you would want to use one or the other. You might even be asked to implement a basic version of SGD in a coding interview (which we cover in Chapter 9, problem #35).

Model Evaluation and Selection

With the math underlying machine learning techniques out of the way, how do we actually choose the best model for our problem, or compare two models against each other? *Model evaluation* is the process of evaluating how well a model performs on the *test set* after it's been trained on the *train set*. Separating out your training data — usually 80% for the train set — from the 20% of the test set is critical because the usefulness of a model boils down to how good predictions are on data that has not been seen before.

Model selection, as the name implies, is the process of selecting which model to implement after each model has been evaluated. Both steps (evaluation and selection) are critical to get right, because even tiny changes in model performance can lead to massive gains at big tech companies. For example, at Facebook, a model that can cause even a 0.1% lift in ad click-through rates can lead to $10+ million in extra revenue.

That's why in interviews, especially during case-study questions where you solve an open-ended problem, discussions often head toward comparing and contrasting models, and selecting the most suitable one after factoring in business and product constraints. Thus, internalizing the concepts covered in this section is key to succeeding in ML interviews.

Bias-Variance Trade-off

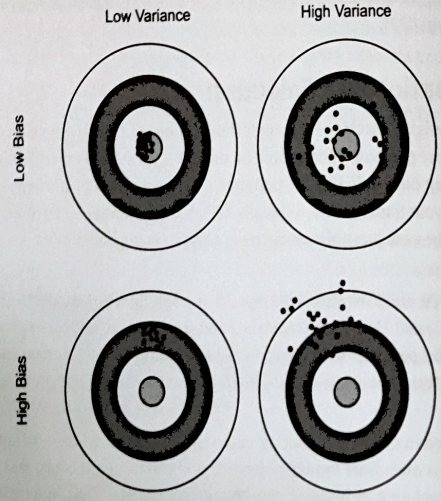
The bias-variance trade-off is an interview classic, and is a key framework for understanding different kinds of models. With any model, we are usually trying to estimate a function $f(x)$, which predicts our target variable y based on our input x. This relationship can be described as follows:

$$y = f(x) + w$$

where w is noise, not captured by $f(x)$, and is assumed to be distributed as a zero-mean Gaussian random variable for certain regression problems. To assess how well the model fits, we can decompose the error of y into the following:

1. ***Bias:*** how close the model's predicted values come to the true underlying $f(x)$ values, with smaller being better
2. ***Variance:*** the extent to which model prediction error changes based on training inputs, with smaller being better
3. ***Irreducible error:*** variation due to inherently noisy observation processes

The trade-off between bias and variance provides a lens through which you can analyze different models. Say we want to predict housing prices given a large set of potential predictors (square footage of a house, the number of bathrooms, and so on). A model with high bias but low variance, such as linear regression, is easy to implement but may oversimplify the situation at hand. This high bias but low variance situation would mean that predicted house prices are frequently off from the market value, but the variance in these predicted prices is low. On the flip side, a model with low bias and high variance, such as neural networks, would lead to predicted house prices closer to market value, but with predictions varying wildly based on the input features.

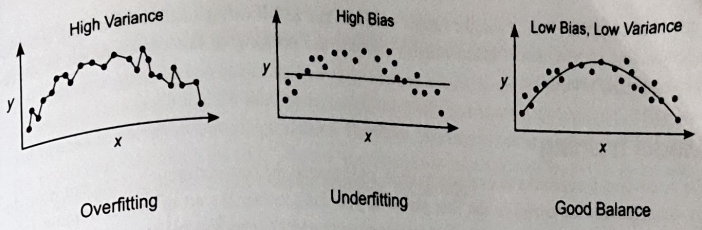


While the bias-variance trade-off equation occasionally shows up in data science interviews, more frequently, you'll be asked to reason about the bias-variance trade-off given a specific situation. For example, presented with a model that has high variance, you could mention how you'd source additional data to fix the issue. Posed with a situation where the model has high bias, you could discuss how increasing the complexity of the model could help. By understanding the business and product requirements, you'll know how to make the bias-variance trade-off for the interview problem posed.

Model Complexity and Overfitting

"All models are wrong, but some are useful" is a well-known adage, coined by statistician George Box. Ultimately, our goal is to discover a model that can generalize to learn some relationship within datasets. Occam's razor, applied to machine learning, suggests that simpler models are generally more useful and correct than more complicated models. That's because simpler, more parsimonious models tend to generalize better.

Said another way, simpler, smaller models are less likely to *overfit* (fit too closely to the training data). Overfit models tend not to generalize well out of sample. That's because during overfitting, the models pick up too much noise or random fluctuations using the training data, which hinders performance on data the model has never seen before.



Underfitting refers to the opposite case — the scenario where the model is not learning enough of the true relationship underlying the data. Because overfitting is so common in real-world machine learning, interviewers commonly ask you how you can detect it, and what you can do to avoid it, which brings us to our next topic: regularization.

Regularization

Regularization aims to reduce the complexity of models. In relation to the bias-variance trade-off, regularization aims to decrease complexity in a way that significantly reduces variance while only slightly increasing bias. The most widely used forms of regularization are L1 and L2. Both methods add a simple penalty term to the objective function. The penalty helps shrink coefficients of features, which reduces overfitting. This is why, not surprisingly, they are also known as shrinkage methods.

Specifically, L1, also known as *lasso*, uses the absolute value of a coefficient to the objective function as a penalty. On the other hand, L2, also known as *ridge*, uses the squared magnitude of a coefficient to the objective function. The L1 and L2 penalties can also be linearly combined, resulting in the popular form of regularization called *elastic net*. Since having models overfit is a prevalent problem in machine learning, it's important to understand when to use each type of regularization. For example, L1 serves as a feature selection method, since many coefficients shrink to 0 (are zeroed out), and hence, are removed from the model. L2 is less likely to shrink any coefficients to 0. Therefore, L1 regularization leads to sparser models, and is thus considered a more strict shrinkage operation.

Interpretability & Explainability

In Kaggle competitions and classwork, you might be expected to maximize a model performance metric like accuracy. However, in the real world, rather than just maximizing a particular metric, you might also be responsible for explaining how your model came up with that output. For example, if your model predicts that someone shouldn't get a loan, doesn't that person deserve to know why? More broadly, interpretable models can help you identify biases in the model, which leads to more ethical AI. Plus, in some domains like healthcare, there can be deep auditing on decisions, and explainable models can help you stay compliant. However, there's usually a trade-off between performance and model interpretability. Often, using a more complex model might increase performance, but make results harder to interpret.

Various models have their own way of interpreting feature importance. For example, linear models have weights which can be visualized and analyzed to interpret the decision making. Similarly, random forests have feature importance readily available to identify what the model is using and learning. There are also some general frameworks that can help with more "black-box" models. One is *SHAP* (SHapley Additive exPlanation), which uses "Shapley" values to denote the average marginal contribution of a feature over all possible combinations of inputs. Another technique is *LIME* (Local Interpretable Model-agnostic Explanations), which uses sparse linear models built around various predictions to understand how any model performs in that local vicinity.

While it's rare to be asked about the details of SHAP and LIME during interviews, having a basic understanding of why model interpretability matters, and bringing up this consideration in more open-ended problems is key.

Model Training

We've covered frameworks to evaluate models, and selected the best-performing ones, but how do we actually train the model in the first place? If you don't master the art of model training (aka teaching machines to learn), even the best machine learning techniques will fail. Recall the basics: we first train models on a training dataset and then test the models on a testing dataset. Normally, 80% of the data will go towards training data, and 20% serves as the test set. But as we soon cover, there's much more to model training than the 80/20 train vs. test split.

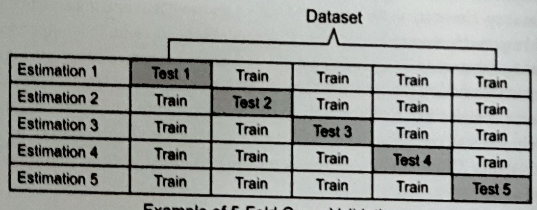
Cross-Validation

Cross-validation assesses the performance of an algorithm in several subsamples of training data. It consists of running the algorithm on subsamples of the training data, such as the original data without some of the original observations, and evaluating model performance on the portion of the data that was excluded from the subsample. This process is repeated many times for the different subsamples, and the results are combined at the end.

Cross-validation helps you avoid training and testing on the same subsets of data points, which would lead to overfitting. As mentioned earlier, in cases where there isn't enough data or getting more data is costly, cross-validation enables you to have more faith in the quality and consistency of a model's test performance. Because of this, questions about how cross-validation works and when to use it are routinely asked in data science interviews.

One popular way to do cross-validation is called *k-fold cross-validation*. The process is as follows:

1. Randomly shuffle data into equally-sized blocks (folds).
2. For each fold k, train the model on all the data except for fold i, and evaluate the validation error using block i.
3. Average the k validation errors from step 2 to get an estimate of the true error.



Example of 5-Fold Cross Validation

Another form of cross-validation you're expected to know for the interview is *leave-one-out cross-validation*. LOOCV is a special case of k-fold cross-validation where k is equal to the size of the dataset (n). That is, it is where the model is testing on every single data point during the cross-validation.

In the case of larger datasets, cross-validation can become computationally expensive, because every fold is used for evaluation. In this case, it can be better to use *train validation split*, where you split

the data into three parts: a training set, a dedicated validation set (also known as a "dev" set), and a test set. The validation set usually ranges from 10%-20% of the entire dataset.

An interview question that comes up from time to time is how to apply cross-validation for time-series data. Standard k-fold CV can't be applied, since the time-series data is not randomly distributed but instead is already in chronological order. Therefore, you should not be using data "in the future" for predicting data "from the past." Instead, you should use historical data up until a given point in time, and vary that point in time from the beginning till the end.

Bootstrapping and Bagging

The process of bootstrapping is simply drawing observations from a large data sample repeatedly (sampling with replacement) and estimating some quantity of a population by averaging estimates from multiple smaller samples. Besides being useful in cases where the dataset is small, bootstrapping is also useful for helping deal with class imbalance: for the classes that are rare, we can generate new samples via bootstrapping.

Another common application of bootstrapping is in ensemble learning: the process of averaging estimates from many smaller models into a main model. Each individual model is produced using a particular sample from the process. This process of bootstrap aggregation is also known as *bagging*. Later in this chapter, we'll show concrete examples of how random forests utilize bootstrapping and bagging.

Ensemble methods like random forests, AdaBoost, and XGBoost are industry favorites, and as such, interviewers tend to ask questions about bootstrapping and ensemble learning. For example, one of the most common interview questions is: "What is the difference between XGBoost and a random forest?"

Hyperparameter Tuning

Hyperparameters are important because they impact a model's training time, compute resources needed (and hence cost), and, ultimately, performance. One popular method for tuning hyperparameters is *grid search*, which involves forming a grid that is the Cartesian product of those parameters and then sequentially trying all such combinations and seeing which yields the best results. While comprehensive, this method can take a long time to run since the cost increases exponentially with the number of hyperparameters. Another popular hyperparameter tuning method is *random search*, where we define a distribution for each parameter and randomly sample from the joint distribution over all parameters. This solves the problem of exploring an exponentially increasing search space, but is not necessarily guaranteed to achieve an optimal result.

While not generally asked about in data science interviews for research scientist or machine learning engineering roles, hyperparameter tuning techniques such as the methods mentioned earlier, along with Bayesian hyperparameter optimization, might be brought up. This discussion mostly happens in the context of neural networks, random forests, or XGBoost. For interviews, you should be able to list a couple of the hyperparameters for your favorite modeling technique, along with what impacts they have on generalization.

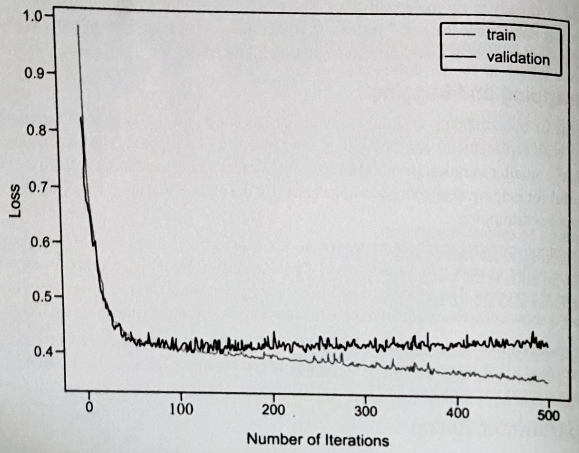
Training Times and Learning Curves

Training time is another factor to consider when it comes to model selection, especially for exceedingl large datasets. As we explain later in the coding chapter, it's possible to use big-O notation to clarif the theoretical bounds on training time for each algorithm. These training time estimates are base on the number of data points and the dimensionality of the data.

For real-life training ML models, you should also factor in training time considerations and resour constraints during model selection. While you can always train more complex models that mi

achieve marginally higher model performance metrics, the trade-off versus increased resource usage and training time might make such a decision suboptimal.

Learning curves are plots of model learning performance over time. The *y*-axis is some metric of learning (for example, classification accuracy), and the *x*-axis is experience (time).



A popular data science interview question involving learning curves is "How would you identify if your model was overfitting?" By analyzing the learning curves, you should be able to spot whether the model is underfitting or overfitting. For example, above, you can see that as the number of iterations is increasing, the training error is getting better. However, the validation error is not improving — in fact, it is increasing at the end — a clear sign that the model is overfitting and training can be stopped. Additionally, learning curves should help you discover whether a dataset is representative or not. If the data was not representative, the plot would show a large gap between the training curve and validation curve, which doesn't get smaller even as training time increases.

Linear Regression

Linear regression is a form of *supervised learning*, where a model is trained on labeled input data. Linear regression is one of the most popular methods employed in machine learning and has many real-life applications due to its quick runtime and interpretability. That's why there's the joke about regression to regression: where you try to solve a problem with more advanced methods but end up falling back to tried and true linear regression.

As such, linear regression questions are asked in all types of data science and machine learning interviews. Essentially, interviewers are trying to make sure your knowledge goes beyond importing linear regression from scikit-learn and then blindly calling linear_regression.fit(*X*,*Y*). That's why deep knowledge of linear regression — understanding its assumptions, addressing edge cases that come up in real-life scenarios, and knowing the different evaluation metrics — will set you apart from other candidates.

In linear regression, the goal is to estimate a function $f(x)$, such that each feature has a linear relationship to the target variable y, or:

$$y = X\beta$$

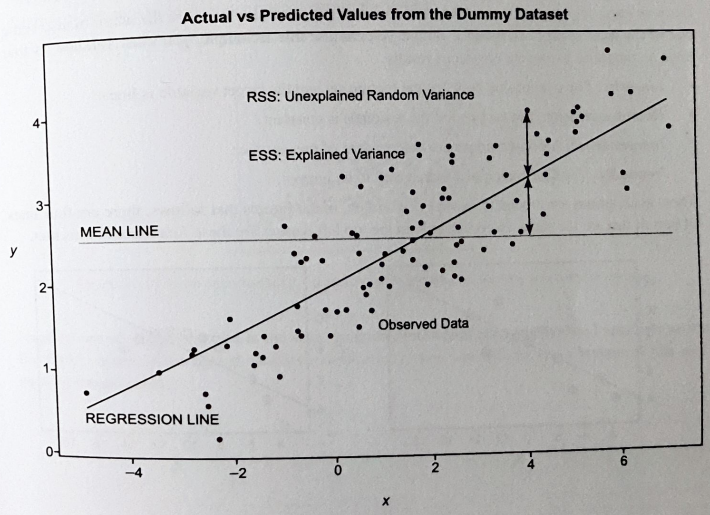
where X is a matrix of predictor variables and β is a vector of parameters that determines the weight of each variable in predicting the target variable. So, how do you compare the performance of two linear regression models?

Evaluating Linear Regression

Evaluation of a regression model is built on the concept of a *residual:* the distance between what the model predicted versus the actual value. Linear regression estimates β by minimizing the residual sum of squares (RSS), which is given by the following:

$$RSS(\beta) = (y - X\beta)^T(y - X\beta)$$

Two other sum of squares concepts to know besides the RSS are the total sum of squares (TSS) and explained sum of squares (ESS). The total sum of squares is the combined variation in the data ($ESS + RSS$). The explained sum of squares is the difference between TSS and RSS. R^2, a popular metric for assessing goodness-of-fit, is given by $R^2 = 1 - \frac{RSS}{TSS}$. It ranges between zero and one, and represents the proportion of variability in the data explained by the model. Other prominent error metrics to measure the goodness-of-fit of linear regression are MSE (mean squared error) and MAE (mean absolute error). MSE measures the *variance* of the residuals, whereas MAE measures the *average* of the residuals; hence, MSE penalizes larger errors more than MAE, making it more sensitive to outliers.



A common interview question is "What's the expected impact on R^2 when adding more features to a model?" While adding more features to a model always increases the R^2, that doesn't necessarily make for a better model. Since any machine learning model can overfit by having more parameters, a goodness-of-fit measure like R^2 should likely also be assessed with model complexity in mind. Metrics that take into account the number of features of linear regression models include AIC, BIC, Mallow's CP, and adjusted R^2.

Subset Selection

So, how do you reduce model complexity of a regression model? *Subset selection*. By default, we use all the predictors in a linear model. However, in practice, it's important to narrow down the number of features, and only include the most important features. One way is *best subset selection*, which tries each model with k predictors, out of p possible ones, where $k < p$. Then, you choose the best subset model using a regression metric like R^2. While this guarantees the best result, it can be computationally infeasible as p increases (due to the exponential number of combinations to try). Additionally, by trying every option in a large search space, you're likely to get a model that overfits with a high variance in coefficient estimates.

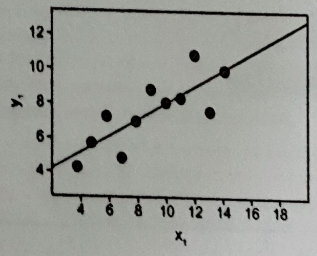
Therefore, an alternative is to use *stepwise selection*. In forward stepwise selection, we start with an empty model and iteratively add the most useful predictor. In backward stepwise selection, we start with the full model and iteratively remove the least useful predictor. While doing stepwise selection, we aim to find a model with high R^2 and low RSS, while considering the number of predictors using metrics like AIC or adjusted R^2.

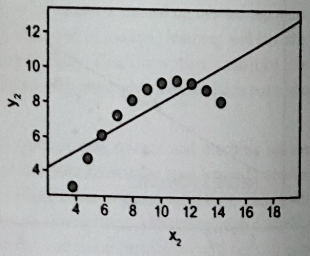
Linear Regression Assumptions

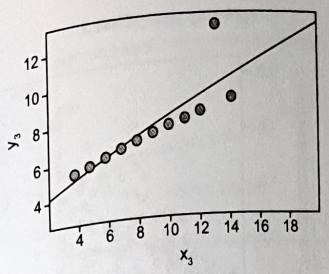
Because linear regression is one of the most commonly applied models, it has the honor of also being one of the most misapplied models. Before you can use this technique, you must validate its four main assumptions to prevent erroneous results:

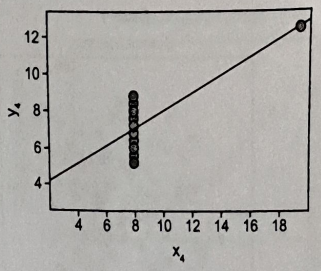
- ***Linearity:*** The relationship between the feature set and the target variable is linear.
- ***Homoscedasticity:*** The variance of the residuals is constant.
- ***Independence:*** All observations are independent of one another.
- ***Normality:*** The distribution of Y is assumed to be normal.

These assumptions are crucial to know. For example, in the figures that follows, there are four lines of best fit that are the same. However, only in the top left dataset are these four assumptions met.







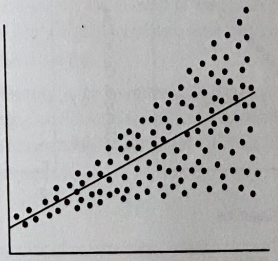


Note: for the independence and normality assumption, use of the term "i.i.d." (independent and identically distributed) is also common. If any of these assumptions are violated, any forecasts or confidence intervals based on the model will most likely be misleading or biased. As a result, the linear regression model will likely perform poorly out of sample.

Avoiding Linear Regression Pitfalls

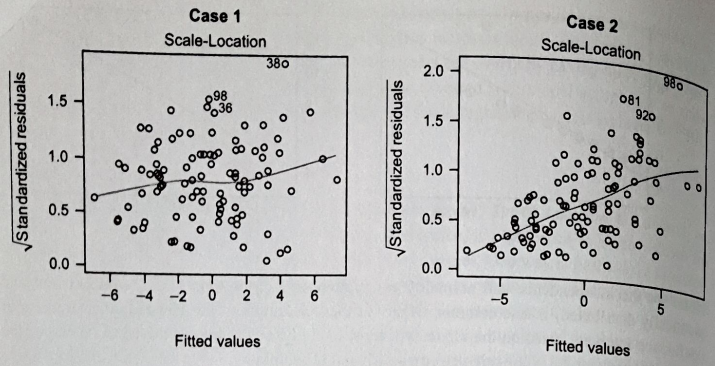
Heteroscedasticity

Linear regression assumes that the residuals (the distance between what the model predicted versus the actual value) are identically distributed. If the variance of the residuals is not constant, then *heteroscedasticity* is most likely present, meaning that the residuals are not identically distributed. To find heteroscedasticity, you can plot the residuals versus the fitted values. If the relationship between residuals and fitted values has a nonlinear pattern, this indicates that you should try to transform the dependent variable or include nonlinear terms in the model.



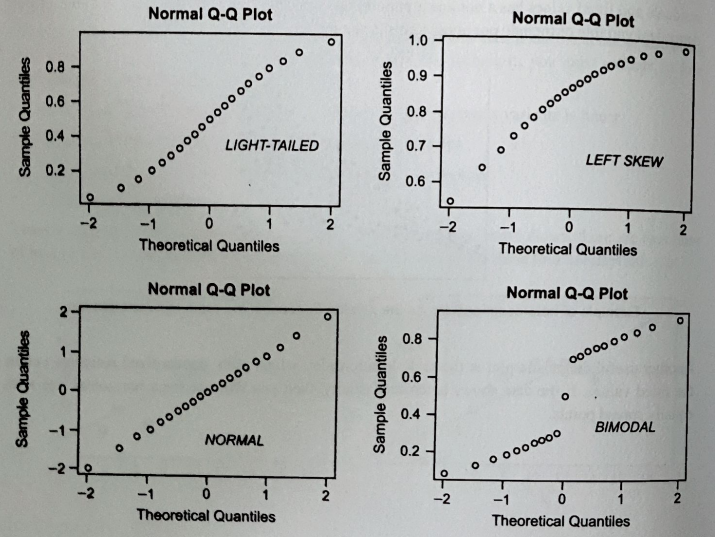
Example of heteroscedasticity. As the x-value increases, the residuals increase too

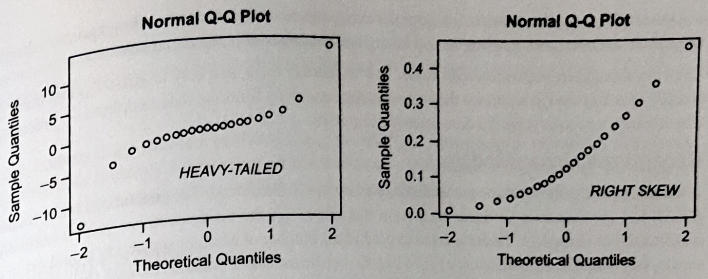
Another useful diagnostic plot is the scale-location plot, which plots standardized residuals versus the fitted values. If the data shows heteroscedasticity, then you will not see a horizontal line with equally spread points.



Normality

Linear regression assumes the residuals are normally distributed. We can test this through a QQ plot. Also known as a quantile plot, a QQ plot graphs the standardized residuals versus theoretical quantiles and shows whether the residuals appear to be normally distributed (i.e., the plot resembles a straight line). If the QQ plot is not a reasonably straight line, this is a sign that the residuals are not normally distributed, and hence, the model should be reexamined. In that case, transforming the dependent variable (with a log or square-root transformation, for example) can help reduce skew.





Outliers

Outliers can have an outsized impact on regression results. There are several ways to identify outliers. One of the more popular methods is examining *Cook's distance*, which is the estimate of the influence of any given data point. Cook's distance takes into account the residual and leverage (how far away the X value differs from that of other observations) of every point. In practice, it can be useful to remove points with a Cook's distance value above a certain threshold.

Multicollinearity

Another pitfall is if the predictors are correlated. This phenomenon, known as ***multicollinearity***, affects the resulting coefficient estimates by making it problematic to distinguish the true underlying individual weights of variables. Multicollinearity is most commonly observed by weights that flip magnitude. It is one of the reasons why model weights cannot be directly interpreted as the importance of a feature in linear regression. Features that initially would appear to be independent variables can often be highly correlated: for example, the number of Instagram posts made and the number of notifications received are most likely highly correlated, since both are related to user activity on the platform, and one generally causes another.

One way to assess multicollinearity is by examining the variance inflation factor (VIF), which quantifies how much the estimated coefficients are inflated when multicollinearity exists. Methods to address multicollinearity include removing the correlated variables, linearly combining the variables, or using PCA/PLS (partial least squares).

Confounding Variables

Multicollinearity is an extreme case of *confounding*, which occurs when a variable (but not the main independent or dependent variables) affects the relationship between the independent and dependent variables. This can cause invalid correlations. For example, say you were studying the effects of ice cream consumption on sunburns and find that higher ice cream consumption leads to a higher likelihood of sunburn. That would be an incorrect conclusion because temperature is the confounding variable — higher summer temperatures lead to people eating more ice cream and also spending more time outdoors (which leads to more sunburn).

Confounding can occur in many other ways, too. For example, one way is *selection bias*, where the data are biased due to the way they were collected (for example, group imbalance). Another problem, known as *omitted variable bias*, occurs when important variables are omitted, resulting in a linear regression model that is biased and inconsistent. Omitted variables can stem from dataset generation issues or choices made during modeling. A common way to handle confounding is stratification, a

process where you create multiple categories or subgroups in which the confounding variables do not vary much, and then test significance and strength of associations using chi square.

Knowing about these regression edge cases, how to identify them, and how to guard against them is crucial. This knowledge separates the seasoned data scientists from the data neophyte — precisely why it's such a popular topic for data science interviews.

Generalized Linear Models

In linear regression, the residuals are assumed to be normally distributed. The generalized linear model (GLM) is a generalization of linear regression that allows for the residuals to not just be normally distributed. For example, if Tinder wanted to predict the number of matches somebody would get in a month, they would likely want to use a GLM like the one below with a Poisson response (called Poisson regression) instead of a standard linear regression.The three common components to any GLM are:

Link Function	Systematic Component	Random Component
$ln\,\lambda_i$	$= b_0 + b_1 x_i$	$+\varepsilon$
y_i	$\sim$ Poisson (λ_i)	

- ***Random Component:*** is the distribution of the error term, i.e., normal distribution for linear regression.
- ***Systematic Component:*** consists of the explanatory variables, i.e., the predictors combined in a linear combination.
- ***Link function:*** is the link between the random and system components, i.e., a linear regression, logit regression, etc.

Note that in GLMs, the response variable is still a linear combination of weights and predictors.

Regression can also use the weights and predictors nonlinearly; the most common examples of this are polynomial regressions, splines, and general additive models. While interesting, these techniques are rarely asked about in interviews and thus are beyond the scope of this book.

Classification

General Framework

Interview questions related to classification algorithms are commonly asked during interviews due to the abundance of real-life applications for assigning categories to things. For example, classifying users as likely to churn or not, predicting whether a person will click on an ad or not, and distinguishing fraudulent transactions from legitimate ones are all applications of the classification techniques we mention in this section.

The goal of classification is to assign a given data point to one of K possible classes instead of calculating a continuous value (as in regression). The two types of classification models are ***generative models*** and ***discriminative models***. Generative models deal with the joint distribution of X and Y, which is defined as follows:

$$p(X, Y) = p(Y|X)p(X)$$

Maximizing a posterior probability distribution produces decision boundaries between classes where the resulting posterior probability is equivalent. The second type of model is discriminative. It directly determines a decision boundary by choosing the class that maximizes the probability:

$$\hat{y} = \arg\max_k p(Y = k|x)$$

Thus, both methods choose a predicted class that maximizes the posterior probability distribution; the difference is simply the approach. While traditional classification deals with just two classes (0 or 1), multi-class classification is common, and many of the below methods can be adapted to handle multiple labels.

Evaluating Classifiers

Before we detail the various classification algorithms like logistic regression and Naive Bayes, it's essential to understand how to evaluate the predictive power of a classification model.

Say you are trying to predict whether an individual has a rare cancer that only happens to 1 in 10,000 people. By default, you could simply predict that every person doesn't have cancer and be accurate 99.99% of the time. But clearly, this isn't a helpful model — Pfizer won't be acquiring our diagnostic test anytime soon! Given imbalanced classes, assessing accuracy alone is not enough — this is known as the "accuracy paradox" and is the reason why it's critical to look at other measures for misclassified observations.

Building and Interpreting a Confusion Matrix

When building a classifier, we want to minimize the number of misclassified observations, which in binary cases can be termed false positives and false negatives. In a false positive, the model incorrectly predicts that an instance belongs to the positive class. For the cancer detection example, a false positive would be classifying an individual as having cancer, when in reality, the person does not have it. On the other hand, a false negative occurs when the model incorrectly produces a negative class. In the cancer diagnostic case, this would mean saying a person doesn't have cancer, when in fact they do.

A *confusion matrix* helps organize and visualize this information. Each row represents the actual number of observations in a class, and each column represents the number of observations predicted as belonging to a class.

		Predicted: Positive	Predicted: Negative	
Actual Class	Positive	True Positive (TP)	False Negative (FN) **Type 2 Error**	**Sensitivity** $\frac{TP}{(TP + FN)}$
	Negative	False Positive (FP) **Type 1 Error**	True Negative (TN)	**Specificity** $\frac{TN}{(TN + FP)}$
		Precision $\frac{TP}{(TP + FP)}$	**Negative Predictive Value** $\frac{TN}{(TN + FN)}$	**Accuracy** $\frac{TP + TN}{(TP + TN + FP + FN)}$

Precision and Recall

Two metrics that go beyond accuracy are *precision* and *recall*. In classification, precision is the actual positive proportion of observations that were predicted positive by the classifier. In the cancer

diagnostic example, it's the percentage of people you said would have cancer who actually ended up having the disease. Recall, also known as *sensitivity*, is the percentage of total positive cases captured, out of all positive cases. It's essentially how well you do in finding people with cancer.

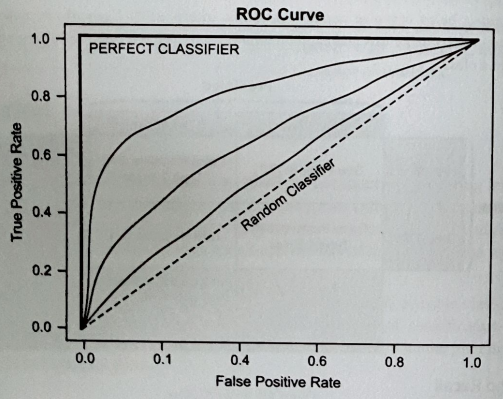
In real-world modeling, there's a natural trade-off between optimizing for precision or recall. For example, having high recall — catching most people who have cancer — ends up saving the lives of some people with the disease. However, this often leads to misdiagnosing others who didn't truly have cancer, which subjects healthy people to costly and dangerous treatments like chemotherapy for a cancer they never had. On the flip side, having high precision means being confident that when the diagnostic comes back positive, the person really has cancer. However, this often means missing some people who truly have the disease. These patients with missed diagnoses may gain a false sense of security, and their cancer, left unchecked, could lead to fatal outcomes.

During interviews, be prepared to talk about the precision versus recall trade-off. For open-ended case questions and take-home challenges, be sure to contextualize the business and product impact of a false positive or a false negative. In cases where both precision and recall are equally important, you can optimize the *F1 score*: the harmonic mean of precision and recall.

$$F_1 = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}}$$

Visualizing Classifier Performance

Besides precision, recall, and the F1 score, another popular way to evaluate classifiers is the receiver operating characteristic (ROC) curve. The ROC curve plots the true positive rate versus the false positive rate for various thresholds. The area under the curve (AUC) measures how well the classifier separates classes. The AUC of the ROC curve is between zero and one, and a higher number means the model performs better in separating the classes. The most optimal is a curve that "hugs" the top left of the plot, as shown below. This indicates that a model has a high true-positive rate and relatively low false-positive rate.



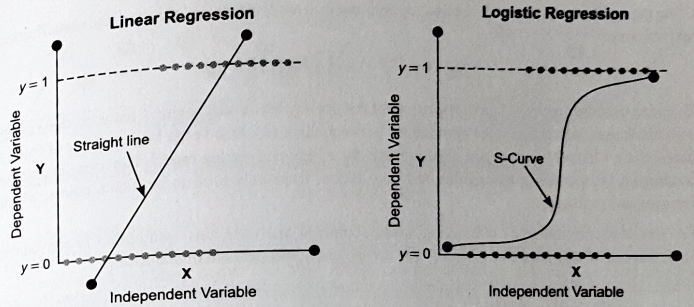
Logistic Regression

One of the most popular classification algorithms is logistic regression, and it is asked about almost as frequently as linear regression during interviews. In logistic regression, a linear output is converted into a probability between 0 and 1 using the sigmoid function:

$$S(x) = \frac{1}{1 + e^{-X\beta}}$$

In the equation above, X is the set of predictor features and β is the corresponding vector of weights. Computing $S(x)$ above produces a probability that indicates if an observation should be classified as a "1" (if the calculated probability is at least 0.5), and a "0" otherwise.

$$P(\hat{Y} = 1|X) = S(X\beta)$$



The loss function for logistic regression, also known as log-loss, is formulated as follows:

$$L(\omega) = \sum_{i=1}^{n} y_i \log\left(\frac{1}{S(X\beta)}\right) + (1 - y_i)\log\left(\frac{1}{1 - S(X\beta)}\right)$$

Note that in cases where more than two outcome classes exist, softmax regression is a commonly used technique that generalizes logistic regression.

In practice, logistic regression, much like its cousin linear regression, is often used because it is highly interpretable: its output, a predicted probability, is easy to explain to decision makers. Additionally, its quickness to compute and ease of use often make it the first model employed for classification problems in a business context.

Note, however, that logistic regression does not work well under certain circumstances. Its relative simplicity makes it a high-bias and low-variance model, so it may not perform well when the decision boundary is not linear. Additionally, when features are highly correlated, the coefficients β won't be as accurate. To address these cases, you can use techniques similar to those used in linear regression (regularization, removal of features, etc.) for dealing with this issue. For interviews, it is critical to understand both the mechanics and pitfalls of logistic regression.

Naive Bayes

Naive Bayes classifiers require only a small amount of training data to estimate the necessary parameters. They can be extremely fast compared to more sophisticated methods (such as support

vector machines). These advantages lead to Naive Bayes being a popularly used first technique in modeling, and is why this type of classifier shows up in interviews.

Naive Bayes uses Bayes' rule (covered in Chapter 6: Statistics) and a set of conditional independence assumptions in order to learn $P(Y|X)$. There are two assumptions to know about Naive Bayes:

1. It assumes each X_i is independent of any other X_j given Y for any pair of features X_i and X_j.
2. It assumes each feature is given the same weight.

The decoupling of the class conditional feature distributions means that each distribution can be independently estimated as a one-dimensional distribution. That is, we have the following:

$$P(X_1 ... X_n | Y) = \prod_{i=1}^{n} P(X_i | Y)$$

Using the conditional independence assumption, and then applying Bayes' theorem, the classification rule becomes:

$$\hat{y} = \arg\max_{y_i} P(Y = y_i) \prod_j P(X_j \mid Y = y_i)$$

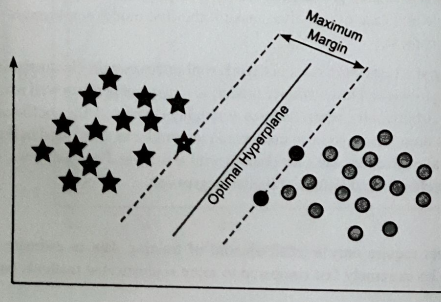
To understand the beauty of Naive Bayes, recall that for any ML model having k features, there are 2^k possible feature interactions (the correlations between them all). Due to the large number of feature interactions, typically you'd need 2^k data points for a high-performing model. However, due to the conditional independence assumption in Naive Bayes, there only need to be k data points, which removes this problem.

For text classification (e.g., classifying spam, sentiment analysis), this assumption is convenient since there are many predictors (words) that are generally independent of one another.

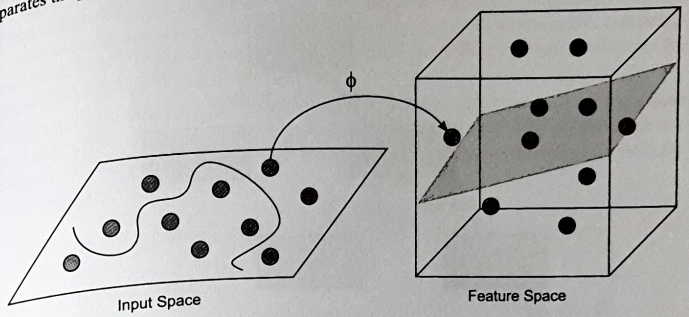
While the assumptions simplify calculations and make Naive Bayes highly scalable to run, they are often not valid. In fact, the first conditional independence assumption generally never holds true, since features do tend to be correlated. Nevertheless, this technique performs well in practice since most data is linearly separable.

SVMs

The goal of SVM is to form a hyperplane that linearly separates the training data. Specifically, it aims to maximize the margin, which is the minimum distance from the decision boundary to any training point. The points closest to the hyperplane are called the support vectors. Note that the decision boundaries for SVMs can be nonlinear, which is unlike that of logistic regression, for example.



In the image above, it's easy to visualize how a line can be found that separates the points correctly into their two classes. In practice, splitting the points isn't that straightforward. Thus, SVMs rely on a kernel to transform data into a higherdimensional space, where it then finds the hyperplane that best separates the points. The image below visualizes this kernel transformation:



Mathematically, the kernel generalizes the dot product to a higher dimension:

$$k(x, y) = \phi(x)^T \phi(y)$$

The RBF (radial basis function) and Gaussian kernels are the two most popular kernels used in practice. The general rule of thumb is this: for linear problems, use a linear kernel, and for nonlinear problems, use a nonlinear kernel like RBF. SVMs can be viewed as a kernelized form of ridge regression because they modify the loss function employed in ridge regression.

SVMs work well in high-dimensional spaces (a larger number of dimensions versus the number of data points) or when a clear hyperplane divides the points. Conversely, SVMs don't work well on enormous data sets, since computational complexity is high, or when the target classes overlap and there is no clean separation. Compared to simpler methods with linear decision boundaries such as logistic regression and Naive Bayes, you may want to use SVMs if you have nonlinear decision boundaries and/or a much smaller amount of data. However, if interpretability is important, SVMs are not preferred because they do not have simple-to-understand outputs (like logistic regression does with a probability).

For ML-heavy roles, know when to use which kernel, the kernel trick, and the underlying optimization problem that SVMs solve.

Decision Trees

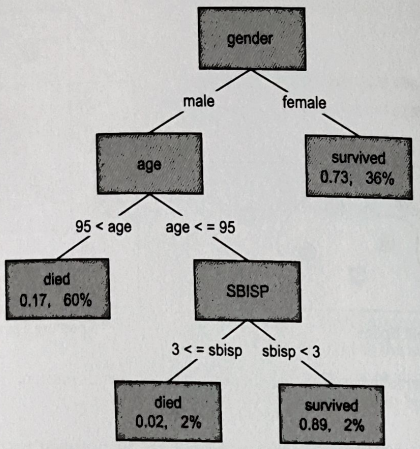
Decision trees and random forests are commonly discussed during interviews since they are flexible and often perform well in practice for both classification and regression use cases. Since both use cases are possible, decision trees are also known as CART (classification and regression trees). For this section, we'll focus on the classification use case for decision trees. While reading this section, keep in mind that for interviews, it helps to understand how both decision trees and random forests are trained. Related topics of entropy and information gain are also crucial to review before a data science interview.

Training

A decision tree is a model that can be represented in a treelike form determined by binary splits made in the feature space and resulting in various leaf nodes, each with a different prediction. Trees are

trained in a greedy and recursive fashion, starting at a root node and subsequently proceeding through a series of binary splits in features (i.e., variables) that lead to minimal error in the classification of observations.

Survival of Passengers on the Titanic



Entropy

The entropy of a random variable Y quantifies the uncertainty in Y. For a discrete variable Y (assuming k states) it is stated as follows:

$$H(Y) = -\sum_{i=1}^{k} P(y = k) \log P(Y = k)$$

For example, for a simple Bernoulli random variable, this quantity is highest when $p = 0.5$ and lowest when $p = 0$ or $p = 1$, a behavior that aligns intuitively with its definition since if $p = 0$ or 1, then there is no uncertainty with respect to the result. Generally, if a random variable has high entropy, its distribution is closer to uniform than a skewed one. There are many measures of entropy — in practice, the Gini index is commonly used for decision trees.

In the context of decision trees, consider an arbitrary split. We have $H(Y)$ from the initial training labels and assume that we have some feature X on which we want to split. We can characterize the reduction in uncertainty given by the feature X, known as information gain, which can be formulated as follows:

$$IG(Y, X) = H(Y) - H(Y|X)$$

The larger $IG(Y, X)$ is, the higher the reduction in uncertainty in Y by splitting on X. Therefore, the general process assesses all features in consideration and chooses the feature that maximizes this information gain, then recursively repeats the process on the two resulting branches.

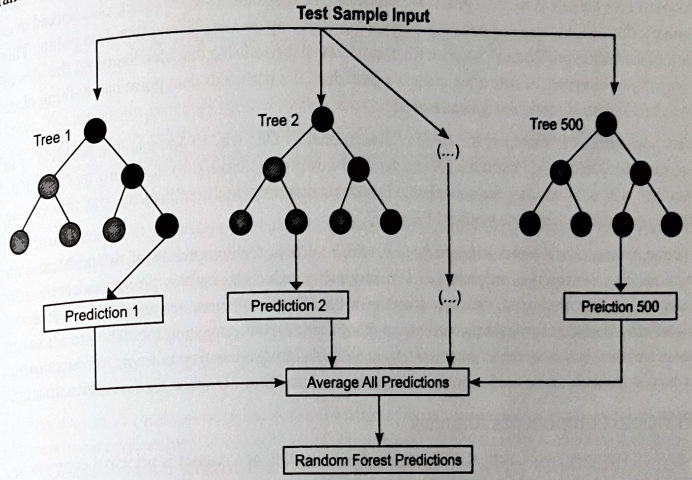
Random Forests

Typically, an individual decision tree may be prone to overfitting because a leaf node can be created for each observation. In practice, random forests yield better out-ofsample predictions than decision

trees. A random forest is an ensemble method that can utilize many decision trees, whose decisions it averages.

Two characteristics of random forests allow a reduction in overfitting and the correlation between the trees. The first is bagging, where individual decision trees are fitted following each bootstrap sample and then averaged afterwards. Bagging significantly reduces the variance of the random forest versus the variance of any individual decision trees. The second way random forests reduce overfitting is that a random subset of features is considered at each split, preventing the important features from always being present at the tops of individual trees.

Random forests are often used due to their versatility, interpretability (you can quickly see feature importance), quick training times (they can be trained in parallel), and prediction performance. In interviews, you'll be asked about how they work versus a decision tree, and when you would use a random forest over other techniques.



Boosting

Boosting is a type of ensemble model that trains a sequence of "weak" models (such as small decision trees), where each one sequentially compensates for the weaknesses of the preceding models. Such weaknesses can be measured by the current model's error rate, and the relative error rates can be used to weigh which observations the next models should focus on. Each training point within a dataset is assigned a particular weight and is continually re-weighted in an iterative fashion such that points that are mispredicted take on higher weights in each iteration. In this way, more emphasis is placed on points that are harder to predict. This can lead to overfitting if the data is especially noisy.

One example is *AdaBoost* (adaptive boosting), which is a popular technique used to train a model based on tuning a variety of weak learners. That is, it sequentially combines decision trees with a single split, and then weights are uniformly set for all data points. At each iteration, data points are re-weighted according to whether each was classified correctly or incorrectly by a classifier. At the end, weighted predictions of each classifier are combined to obtain a final prediction.

The generalized form of AdaBoost is called gradient boosting. A well-known form of gradient boosting used in practice is called XGBoost (extreme gradient boosting). Gradient boosting is similar to AdaBoost, except that shortcomings of previous models are identified by the gradient rather than high weight points, and all classifiers have equal weights instead of having different weights. In industry, XGBoost is used heavily due to its execution speed and model performance.

Since random forests and boosting are both ensemble methods, interviewers tend to ask questions comparing and contrasting the two. For example, one of the most common interview questions is "What is the difference between XGBoost and a random forest?"

Dimensionality Reduction

Imagine you have a dataset with one million rows but two million features, most of which are null across the data points. You can intuitively guess that it would be hard to tease out which features are predictive for the task at hand. In geometric terms, this situation demonstrates sparse data spread over multiple dimensions, meaning that each data point is relatively far away from other data points. This lack of distance is problematic, because when extracting patterns using machine learning, the idea of similarity or closeness of data often matters a great deal. If a particular data point has nothing close to it, how can an algorithm make sense of it?

This phenomenon is known as the *curse of dimensionality*. One way to address this problem is to increase the dataset size, but often, in practice, it's costly or infeasible to get more training data. Another way is to conduct feature selection, such as removing multicollinearity, but this can be challenging with a very large number of features.

Instead, we can use *dimensionality reduction*, which reduces the complexity of the problem with minimal loss of important information. Dimensionality reduction enables you to extract useful information from such data, but can sometimes be difficult or even too expensive, since the algorithm we would use would incorporate so many features. Decomposing the data into a smaller set of variables is also useful for summarizing and visualizing datasets. For example, dimensionality reduction methods can be used to project a large dataset into 2D or 3D space for easier visualization.

Principal Components Analysis

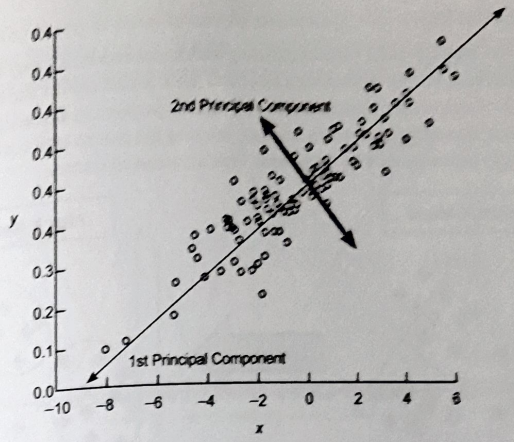
The most commonly used method to reduce the dimensionality of a dataset is principal components analysis (PCA). PCA combines highly correlated variables into a new, smaller set of constructs called *principal components*, which capture most of the variance present in the data. The algorithm looks for a small number of linear combinations for each row vector to explain the variance within X. For example, in the image below, the variation in the data is largely summarized by two principal components.

More specifically, PCA finds the vector w of weights such that we can define the following linear combination:

$$y_i = w_i^T x = \sum_{j=1}^{p} w_{ij} x_j$$

subject to the following:

y_i is uncorrelated with y_j, $var(y_i)$ is maximized



Hence, the algorithm proceeds by first finding the component having maximal variance. Then, the second component found is uncorrelated with the first and has the second-highest variance, and so on for the other components. The algorithm ends with some number k dimensions such that

$$y_1, \ldots y_k \text{ explain the majority of } k \text{ variance, } k << p$$

The final result is an eigendecomposition of the covariance matrix of X, where the first principal component is the eigenvector corresponding to the largest eigenvalue and the second principal component corresponds to the eigenvector with the second largest eigenvalue, and so on. Generally, the number of components you choose is based on your threshold for the percent of variance your principal components can explain. Note that while PCA is a linear dimensionality reduction method, t-distributed stochastic neighbor embedding (t-SNE) is a non-linear, non-deterministic method used for data visualization.

In interviews, PCA questions often test your knowledge of the assumptions (like that the variables need to have a linear relationship). Commonly asked about as well are pitfalls of PCA, like how it struggles with outliers, or how it is sensitive to the units of measurement for the input features (data should be standardized). For more ML-heavy roles, you may be asked to whiteboard the eigendecomposition.

Clustering

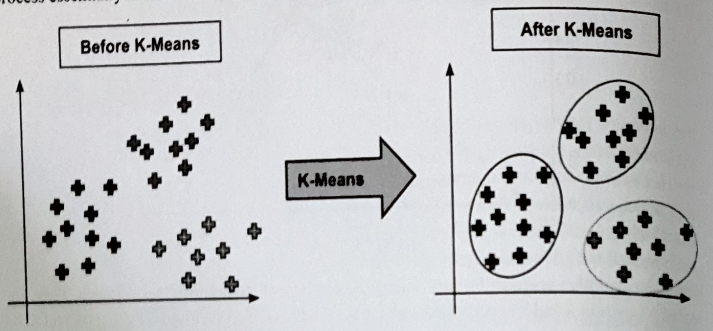
Clustering is a popular interview topic since it is the most commonly employed unsupervised machine learning technique. Recall that unsupervised learning means that there is no labeled training data, i.e., the algorithm is trying to infer structural patterns within the data, without a prediction task in mind. Clustering is often done to find "hidden" groupings in data, like segmenting customers into different groups, where the customers in a group have similar characteristics. Clustering can also be used for data visualization and outlier identification, as in fraud detection, for instance. The goal of clustering is to partition a dataset into various clusters or groups by looking only at the data's input features.

Ideally, the clustered groups have two properties:

- Points within a given cluster are similar (i.e., high intra-cluster similarity).
- Points in different clusters are not similar (i.e., low inter-cluster similarity).

K-Means clustering

A well-known clustering algorithm, *k*-means clustering is often used because it is easy to interpret and implement. It proceeds, first, by partitioning a set of data into *k* distinct clusters and then arbitrarily selects centroids of each of these clusters. It iteratively updates partitions by first assigning points to the closest cluster, then updating centroids, and then repeating this process until convergence. This process essentially minimizes the total inter-cluster variation across all clusters.



Mathematically, *k*-means clustering reaches a solution by minimizing a loss function (also known as distortion function). In this example, we minimize Euclidean distance (given x_i points and centroid value μ_j):

$$L = \sum_{j=1}^{k} \sum_{x \in S_j} \|x_i - \mu_j\|^2$$

where S_j represents the particular cluster.

The iterative process continues until the cluster assignment updates fail to improve the objective function. Note that *k*-means clustering uses Euclidean distance when assessing how close points are to one another and that *k*, the number of clusters to be estimated, is set by the user and can be optimized if necessary.

K-means Alternatives

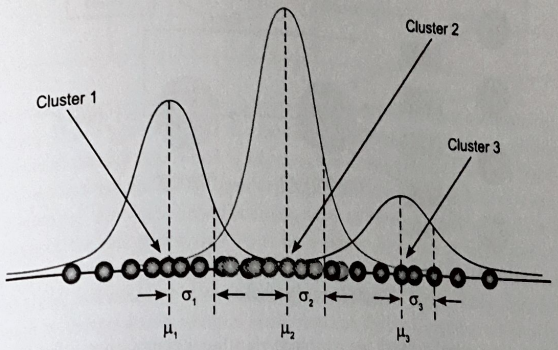
One alternative to *k*-means is *hierarchical clustering*. Hierarchical clustering assigns data points to their own cluster and merges clusters that are the nearest (based on any variety of distance metrics) until there is only one cluster left, generally visualized using a dendrogram. In cases where there is not a specific number of clusters, or you want a more interpretable and informative output, hierarchical clustering is more useful than *k*-means.

While quite similar to *k*-means, *density clustering* is another distinct technique. The most well-known implementation of this technique is DBSCAN. Density clustering does not require a number of clusters as a parameter. Instead, it infers that number, and learns to identify clusters of arbitrary shapes. Generally, density clustering is more helpful for outlier detection than *k*-means.

Gaussian Mixture Model (GMM)

A GMM assumes that the data being analyzed come from a "mixture" of *k* Gaussian/normal distributions, each having a different mean and variance, where the mixture components are basically the proportion of observations in each group. Compared to *k*-means, which is a deterministic algorithm where *k* is set in advance, GMMs essentially try to learn the true value of *k*.

For example, TikTok may be on the lookout for anomalous profiles, and can use GMMs to cluster various accounts based on features (number of likes sent, messages sent, and comments made) and identify any accounts whose activity metrics don't seem to fall within the typical user activity distributions.



Compared to *k*-means, GMMs are more flexible because *k*-means only takes into account the mean of a cluster, while GMMs take into account the mean and variance. Therefore, GMMs are particularly useful in cases with low-dimensional data or where cluster shapes may be arbitrary. While practically never asked about for data science interviews (compared to *k*-means), we brought up GMMs for those seeking more technical ML research and ML engineering positions.

Neural Networks

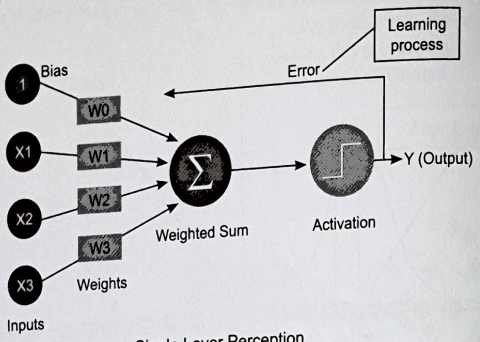
While the concepts behind neural networks have been around since the 1950s, it's only in the last 15 years that they've grown in popularity, thanks to an explosion of data being created, along with the rise of cheap cloud computing resources needed to store and process the massive amounts of newly created data. As mentioned earlier in the chapter, if your résumé has any machine learning projects involving deep learning experience, then the technical details behind neural networks will be considered fair game by most interviewers. But for a product data science position or a finance role (where data can be very noisy, so most models are not purely neural networks), don't expect to be bombarded with tough neural network questions. Knowing the basics of classical ML techniques should suffice.

When neural nets are brought up during interviews, questions can range anywhere from qualitative assessments on how deep learning compares to more traditional machine learning models to mathematical details on gradient descent and backpropagation. On the qualitative side, it helps to understand all of the components that go into training neural networks, as well as how neural networks compare to simpler methods.

Perceptron

Neural networks function in a way similar to biological neurons. They take in various inputs (at input layers), weight these inputs, and then combine the weighted inputs through a linear combination (much like linear regression). If the combined weighted output is past some threshold set by an *activation function*, the output is then sent out to other layers. This base unit is generally referred to as

a perceptron. Perceptrons are combined to form neural networks, which is why they are also known as *multi-layer perceptrons* (MLPs).



Single Layer Perception

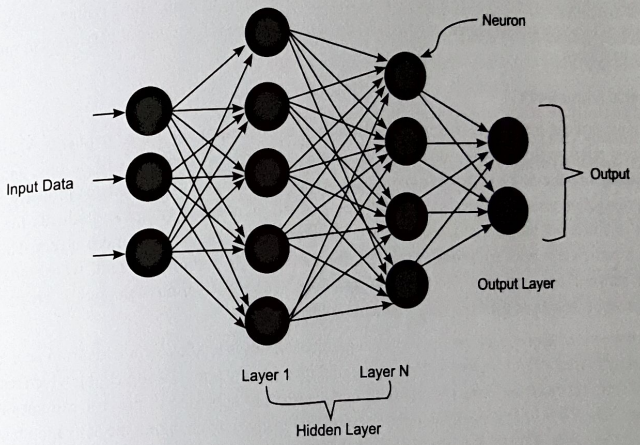
While the inputs for a neural network are combined via a linear combination, often, the activation function is nonlinear. Thus, the relationship between the target variable and the predictor features (variables) frequently ends up also being nonlinear. Therefore, neural networks are most useful when representing and learning nonlinear functions.

For reference, we include a list of common activation functions below. The scope of when to use which activation function is outside of this text, but any person interviewing for an ML-intensive role should know these use cases along with the activation function's formula.

Activation Function	Equation	Example	1D Graph
Unit Step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perception variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perception variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq 0, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1+e^{-z}}$	Logistic regression Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{-z} + e^{-z}}$	Multi-layer Neural Network	

Rectifier, ReLU (Rectifier Linear Unit)	$\phi(z) = \max(0, z)$	Multi-layer Neural Network	
Rectifier, softplus	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Network	

In neural networks, the process of receiving inputs and generating an output continues until an output layer is reached. This is generally done in a forward manner, meaning that layers process incoming data in a sequential forward way (which is why most neural networks are known as "feed-forward"). The layers of neurons that are not the input or output layers are called the *hidden layers* (hence the name "deep learning" for neural networks having many of these). Hidden layers allow for specific transformations of the data within each layer. Each hidden layer can be specialized to produce a particular output — for example, in a neural network used for navigating roads, one hidden layer may identify stop signs, and another hidden layer may identify traffic lights. While those hidden layers are not enough to independently navigate roads, they can function together within a larger neural network to drive better than Nick at age 16.



Backpropagation

The learning process for neural networks is called *backpropagation*. This technique modifies the weights of the neural network iteratively through calculation of deltas between predicted and expected outputs. After this calculation, the weights are updated backward through earlier layers via stochastic gradient descent. This process continues until the weights that minimize the loss function are found.

For regression tasks, the commonly used loss function to be optimized is mean squared error, whereas for classification tasks the common loss function used is cross-entropy. Given a loss function L, we can update the weights w through the chain rule of the following form, where z is the model's output (and the best guess of our target variable y):

$$\frac{\partial L(z,y)}{\partial w} = \frac{\partial L(z,y)}{\partial x} * \frac{\partial x}{\partial z} * \frac{\partial z}{\partial w}$$

and the weights are updated via:

$$w = w - \alpha \frac{\partial L(z,y)}{\partial w}$$

For ML-heavy roles, we've seen interviewers expect candidates to explain the technical details behind basic backpropagation on a whiteboard, for basic methods such as linear regression or logistic regression.

Interviewers also like to ask about the hyperparameters involved in neural networks. For example, the amount that the weights are updated during each training step, α, is called the *learning rate*. If the learning rate is too small, the optimization process may freeze. Conversely, if the learning rate is too large, the optimization might converge prematurely at a suboptimal solution. Besides the learning rate, other hyperparameters in neural networks include the number of hidden layers, the activation functions used, batch size, and so on. For an interview, it's helpful to know how each hyperparameter affects a neural network's training time and model performance.

Training Neural Networks

Because neural networks require a gargantuan amount of weights to train, along with a considerable amount of hyperparameters to be searched for, the training process for a neural network can run into many problems.

General Framework

One issue that can come up in training neural nets is the problem of *vanishing gradients*. Vanishing gradients refers to the fact that sometimes the gradient of the loss function will be tiny, and may completely stop the neural network from training because the weights aren't updated properly. Since backpropagation uses the chain rule, multiplying n small numbers to compute gradients for early layers in a network means that the gradient gets exponentially smaller with more layers. This can happen particularly with traditional activation functions like hyperbolic tangent, whose gradients range between zero and one. The opposite problem, where activation functions p create large derivatives, is known as the exploding gradient problem.

One common technique to address extremes in gradient values is to allow gradients from later layers to directly pass into earlier layers without being multiplied many times — something which residual neural networks (ResNets) and LSTMs both utilize. Another approach to prevent extremes in the gradient values is to alter the magnitude of the gradient changes by changing the activation function used (for example, ReLU). The details behind these methods are beyond this book's scope but are worth looking into for ML-heavy interviews.

Training Optimization Techniques

Additionally, there are quite a few challenges in using vanilla gradient descent to train deep learning models. A few examples include getting trapped in suboptimal local minima or saddle points, not using a good learning rate, or dealing with sparse data with features of different frequencies where we may not want all features to update to the same extent. To address these concerns, there are a variety of optimization algorithms used.

Momentum is one such optimization method used to accelerate learning while using SGD. While using SGD, we can sometimes see small and noisy gradients. To solve this, we can introduce a new

parameter, velocity, which is the direction and speed at which the learning dynamics change. The velocity changes based on previous gradients (in an exponentially decaying manner) and increases the step size for learning in any iteration, which helps the gradient maintain a consistent direction and pace throughout the training process.

Transfer Learning

Lastly, for training neural networks, practitioners use or repurpose a pre-trained layer (components of a model that have already been trained and published). This approach is called *transfer learning* and is especially common in cases where models require a large amount of data (for example, BERT for language models and ImageNet for image classification). Transfer learning is beneficial when you have insufficient data for a new domain, and there is a large pool of existing data that can be transferred to the problem of interest. For example, say you wanted to help Jian Yang from the TV show *Silicon Valley* build an app to detect whether something was a hot dog or not a hotdog. Rather than just using your 100 images of hot dogs, you can use ImageNet (which was trained on many millions of images) to get a great model right off the bat, and then layer on any extra specific training data you might have to further improve accuracy.

Addressing Overfitting

Deep neural networks are prone to overfitting because of the model complexity (there are many parameters involved). As such, interviewers frequently ask about the variety of techniques which are used to reduce the likelihood of a neural network overfitting. Adding more training data is the simplest way to address variance if you have access to significantly more data and computational power to process that data. Another way is to standardize features (so each feature has 0 mean and unit variance), since this speeds up the learning algorithm. Without normalized inputs, each feature takes on a wide range of values, and the corresponding weights for those features can vary dramatically, resulting in larger updates in backpropagation. These large updates may cause oscillation in the weights during the learning stage, which causes overfitting and high variance.

Batch normalization is another technique to address overfitting. In this process, activation values are normalized within a given batch so that the representations at the hidden layers do not vary drastically, thereby allowing each layer of a network to learn more independently of one another. This is done for each hidden neuron, and also improves training speed. Here, applying a standardization process similar to how inputs are standardized is recommended.

Lastly, *dropout* is a regularization technique that deactivates several neurons randomly at each training step to avoid overfitting. Dropout enables simulation of different architectures, because instead of a full original neural network, there will be random nodes dropped at each layer. Both batch normalization and dropout help with regularization since the effects they have are similar to adding noise to various parts of the training process.

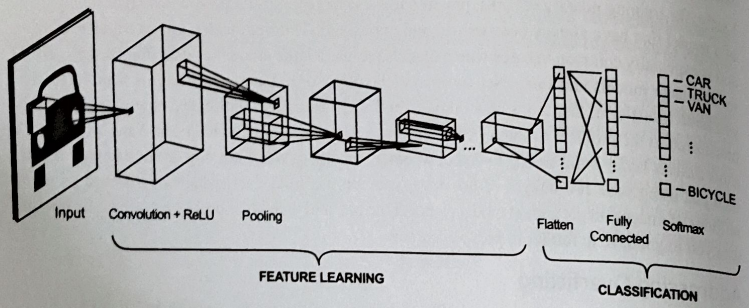
Types of Neural Networks

While a deep dive into the various neural network architectures is beyond the scope of this book, below, we jog your memory with a few of the most popular ones along with their applications. For ML-heavy roles, interviewers tend to ask about the different layer types used within each architecture, and to compare and contrast each architecture against one another.

CNNs

Convolutional neural networks (CNNs) are heavily used in computer vision because they can capture the spatial dependencies of an image through a series of filters. Imagine you were looking at a picture

of some traffic lights. Intuitively, you need to figure out the components of the lights (i.e., red, green, yellow) when processing the image. CNNs can determine elements within that picture by looking at various neighborhoods of the pixels in the image. Specifically, *convolution layers* can extract features such as edges, color, and gradient orientation. Then, *pooling layers* apply a version of dimensionality reduction in order to extract the most prominent features that are invariant to rotation and position. Lastly, the results are mapped into the final output by a fully connected layer.



RNNs

Recurrent neural networks (RNNs) are another common type of neural network. In an RNN, the nodes form a directed graph along a temporal sequence and use their internal state (called memory). RNNs are often used in learning sequential data such as audio or video — cases where the current context depends on past history. For example, say you are looking through the frames of a video. What will happen in the next frame is likely to be highly related to the current frame, but not as related to the first frame of the video. Therefore, when dealing with sequential data, having a notion of memory is crucial for accurate predictions. In contrast to CNNs, RNNs can handle arbitrarily input and output lengths and are not feed-forward neural networks, instead using this internal memory to process arbitrary sequences of data.

LSTMs

Long Short-Term Memory (LSTMs) are a fancier version of RNNs. In LSTMs, a common unit is composed of a cell, an input gate (writing to a cell or not), an output gate (how much to write to a cell), and a forget gate (how much to erase from a cell). This architecture allows for regulating the flow of information into and out of any cell. Compared to vanilla RNNs, which only learn short-term dependencies, LSTMs have additional properties that allow them to learn long-term dependencies. Therefore, in most real-world scenarios, LSTMs are used instead of RNNs.

Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning outside of supervised and unsupervised learning. RL is about teaching an agent to learn which decisions to make in an environment to maximize some reward function. The agent takes a series of actions throughout a variety of states and is rewarded accordingly. During the learning process, the agent receives feedback based on the actions taken and aims to maximize the overall value acquired.

The main components of an RL algorithm include:

- ***Reward function:*** defines the goal of the entire RL problem and quantifies what a "good" or "bad" action is for the agent in any given state.
- ***Policy:*** defines how the agent picks its actions by mapping states to actions.
- ***Model:*** defines how the agent predicts what to do next as the agent understands the environment — given a state and action, the model will predict the reward and the next state.
- ***Value function:*** predicts the overall expected future reward discounted over time; it is consistently re-estimated over time to optimize long-term value.

RL is most widely known for use cases in gaming (AlphaGo, chess, Starcraft, etc.) and robotics. It is best used when the problem at hand is one of actions rather than purely predictions (that is, you do not know what constitutes "good" actions — supervised learning assumes you already know what the output should be). Unless you have particular projects or experience with reinforcement learning, it won't be brought up in data science interviews.

The End-to-End ML Workflow

End-to-end machine learning questions are asked in interviews to see how well you apply machine learning theory to solve business problems. It isn't just about confronting a real-world problem like "How would you design Uber's surge pricing algorithm?" and then jumping to a technique like linear regression or random forests. Instead, it's about asking the right questions about the business goals and constraints that inform the machine learning system design. It's about walking through how you'd explore and clean the data and the features you would try to engineer. It's about picking the right model evaluation metrics and modeling techniques, contextualizing model performance in terms of business impact, mentioning model deployment strategies, and much, much more.

To help you solve these all-encompassing problems — something most ML-theory textbooks don't cover — we walk you through the entire machine learning workflow below. However, to really ace these open-ended ML problems come interview time, also read Chapter 11: Case Studies. In addition to tips on dealing with open-ended problems, the case study chapter includes ML case interview questions that force you to apply the concepts detailed below.

Step 1: Clarify the Problem and Constraints

If you frame the business and product problem correctly, you've done half the work. That's because it's easy to throw random ML techniques at data — but it's harder to understand the business motivations, technical requirements, and stakeholder concerns that ultimately affect the success of a deployed machine learning solution. As such, make sure to start your answer by discussing what the problem at hand is, the business process and context surrounding the problem, any assumptions you might have, and what prospective stakeholder concerns are likely to be.

Some questions you can use to clarify the problem and the constraints include:

- What is the dependent variable we are trying to model? For example, if we are building a user churn model, what criteria are we using to define churn in the first place?
- How has the problem been approached in the past by the business? Is there a baseline performance we can compare against? How much do we need to beat this baseline for the project to be considered a success?

- Is ML even needed? Maybe a simple heuristics or a rules-based approach works well enough? Or perhaps a hybrid approach with humans in the loop would work best?
- Is it even legal or ethical to apply ML to this problem? Are there regulatory issues at play dictating what kinds of data or models you can use? For example, lending institutions cannot legally use some demographic variables like race.
- How do end users benefit from the solution, and how would they use the solution (as a standalone, or an integration with existing systems)?
- Is there a clear value add to the business from a successful solution? Are there any other stakeholders who would be affected?
- If an incorrect prediction is made, how will it impact the business? For example, a spam email making its way into your inbox isn't as problematic as a high-risk mortgage application accidentally being approved.
- Does ML need to solve the entire problem, end-to-end, or can smaller decoupled systems be made to solve sub-problems, whose output is then combined? For example, do you need to make a full self-driving algorithm, or separate smaller algorithms for environment perception, path planning, and vehicle control?

Once you've understood the business problem that your ML solution is trying to solve, you can clarify some of the technical requirements. Aligning on the technical requirements is especially important when confronted with a ML systems design problem. Some questions to ask to anchor the conversation:

- What's the latency needed? For example, search autocomplete is useless if it takes predictions longer to load than it takes users to type out their full query. Does every part of the system need to be real time—while inference may need to be fast, can training be slow?
- Are there any throughput requirements? How many predictions do you need to serve every minute?
- Where is this model being deployed? Does the model need to fit on-device? If so, how big is too big to deploy? And how costly is deployment? For example, adding a high-end GPU to a car is feasible cost-wise, but adding one to a drone might not be.

While spending so much time on problem definition may seem tedious, the reality is that defining the right solution for the right problem can save you many weeks of technical work and painful iterations later down the road. That's why interviewers, when posing open-ended ML problems, expect you to ask the right questions — ones that scope down your solution. By clarifying these constraints and objectives up front, you make better decisions on downstream steps of the end-to-end workflow. To further your business and product clarification skills, read the sections on product sense and company research in Chapter 10: Product Sense.

And one last piece of advice: don't go overboard with the questions! Remember, this is a time-bound interview, so make sure your questions and assumptions are reasonable and relevant (and concise). You don't want to be like a toddler and ask 57 questions without getting anywhere.

Step 2: Establish Metrics

Once you've understood the stakeholder objectives and constraints imposed, it's best to pick simple, observable, and attributable metrics that encapsulate solving the problem. Note that sometimes the business is only interested in optimizing their existing business KPIs (for example, the time to resolve a customer request). In that case, you need to be able to align your model performance metrics with

solving the business problem. For example, for a customer support request classification model, a 90% model accuracy means that 50% of the customer tickets that previously needed to be rerouted now end up in the right place, resulting in a 10% decrease in time to resolution.

In real-world scenarios, it's best to opt for a single metric rather than picking multiple metrics to capture different sub-goals. That's because a single metric makes it easier to rank model performance. Plus, it's easier to align the team around optimizing a single number. However, in interview contexts, it may be beneficial to mention multiple metrics to show you've thought about the various goals and trade-offs your ML solution needs to satisfy. As such, in an interview, we recommend you start your answer with a single metric, but then hedge your answer by mentioning other potential metrics to track.

For example, if posed a question about evaluation metrics for a spam classifier, you could start off by talking about accuracy, and then move on to precision and recall as the conversation becomes more nuanced. In an effort to optimize a single metric, you could recommend using the F-1 score. A nuanced answer could also incorporate an element of satisficing — where a secondary metric is just good enough. For example, you could optimize precision @ recall 0.95 — i.e., constraining the recall to be at least 0.95 while optimizing for precision. Or you could suggest blending multiple metrics into one by weighting different sub-metrics, such as false positives versus false negatives, to create a final metric to track. This is often known as an OEC (overall evaluation criterion), and gives you a balance between different metrics.

Once you've picked a metric, you need to establish what success looks like. While for a classifier, you might desire 100% accuracy, is this a realistic bar for measuring success? Is there a threshold that's good enough? This is why inquiring about baseline performance in Step 1 becomes crucial. If possible, you should use the performance of the existing setup for comparison (for example, if the average time to resolution for customer support tickets is 2 hours, you could aim for 1 hour — not a 97% ticket classification accuracy). Note: in real-world scenarios, the bar for model performance isn't as high as you'd think to still have a positive business impact.

Be sure to voice all these metric considerations to your interviewer so that you can show you've thought critically about the problem. For more guidance, read Chapter 10: Product Sense, which covers the nuances and pitfalls of metric selection.

Step 3: Understand Your Data Sources

Your machine learning model is only as good as the data it sees; hence, the phrase "garbage in, garbage out." For classroom projects or Kaggle competitions, there isn't much you can do since you usually have a fixed dataset, and it's all about fitting a model that maximizes some metric. However, in the real world, you have leeway in what data you use to solve the business problem. As such, clearly articulate what data sources you would prefer to solve the interview problem. While it's intuitive to use the internal company data relevant to the problem at hand, that's not the be-all end-all of data sourcing.

For open-ended ML questions, especially at startups that might be testing your scrappiness, you should think outside the box regarding what data to use. Data sources to consider:

- Can you acquire more data by crowdsourcing it via Amazon Mechanical Turk?
- Can you ask users for data as part of the user onboarding process?
- Can you buy second- and third-party datasets?
- Can you ethically scrape the data from online sources?
- Can you send your unlabeled internal data off to a labeling and annotation service?

To boost model performance, it might not be about collecting more data generally. Instead, you can intentionally source more examples of edge cases via data augmentation or artificial data synthesis. For example, suppose your traffic light detector struggles in low-contrast situations. You could make a version of your training images that has less contrast in order to give your neural network more practice on these trickier photos. Taken to the extreme, you can even simulate the entire environment, as is common in the self-driving car industry. Simulation is used in the autonomous vehicle space because encountering the volume of rare and risky situations needed to adequately train a model based on only real-world driving is infeasible.

Finally, do you understand the data? Questions to consider:

- How fresh is the data? How often will the data be updated?
- Is there a data dictionary available? Have you talked to subject matter experts about it?
- How was the data collected? Was there any sampling, selection, or response bias?

Step 4: Explore Your Data

A good first step in exploratory data analysis is to profile the columns at first glance: which ones might be useful? Which ones have practically no variance and thus wouldn't offer up any real predictive value? Which columns look noisy? Which ones have a lot of missing or odd values? Besides skimming through your data, also look at summary statistics like the mean, median, and quantiles.

> *"The greatest value of a picture is when it forces*
> *us to notice what we never expected to see."*
>
> ***—John Tukey***

Because a picture is worth a thousand words, visualizing your data is also a crucial step in exploratory data analysis. For columns of interest, you want to visualize their distributions to understand their statistical properties like skewness and kurtosis. Certain features (e.g., age, weight) may be better visualized with a histogram through binning.It also helps to visualize the range of continuous variables and plot categories of categorical variables. Finally, you can visually inspect the basic relationships between variables using a correlation matrix. This can help you quickly spot which variables are correlated with one another, as well as what might be correlated with the target variable at hand.

Step 5: Clean Your Data

Did you get suckered into data science and machine learning because you thought most of your time would be spent doing sexy data analysis and modeling work? Don't worry, we got fooled too! The joke (and grievance) that most data scientists spend 80% of their time cleaning data stems from a frustrating truth: if you feed garbage data into a model, you get garbage out. Between the logging issues, missing values, data entry errors, data merges gone wrong, changing schemas due to product changes, and columns whose meaning changes over time, there are many reasons why your data might be a hot mess. That's precisely why a critical step before modeling is data munging.

One aspect of data munging is dropping irrelevant data or erroneously duplicated rows and columns. You should also handle incorrect values that don't match up with the supposed data schema. For example, for fields containing human-inputted data, there is often a pattern of errors or typos that you could find and then use to clean up.

To handle missing data, you should first understand the root cause. Based on the results, several techniques to deal with missing values include:

- Imputing the missing values via basic methods such as column mean/median

- Using a model or distribution to impute the missing data
- Dropping the rows with missing values (as a last resort)

Another critical data cleaning step is dealing with outliers. Outliers may be due to issues in data collection, like manual data entry issues or logging hiccups. Or maybe they accurately reflect the actual data. Outliers can be removed outright, truncated, or left as is, depending on their source and the business implications of including them or not.

Note that outliers may be univariate, while others are multivariate and require looking over many dimensions. For an example of a multivariate outlier, consider a dataset of human ages and heights. A 4-year-old human wouldn't be strange, a 5-foot-tall person isn't odd, but a 4-year-old that's 5-feet tall would either be an outlier, data entry error, or a Guinness world record holder.

Step 6: Feature Engineering

Feature engineering is the art of presenting data to machine learning models in the best way possible. One part of feature engineering is feature selection: the process of selecting a relevant subset of features for model construction, based on domain knowledge. The other is feature preprocessing: transforming your data in a way that allows an algorithm to learn the underlying structure of the data without having to sift through noisy inputs. Careful feature selection and feature processing can boost model performance and let you get away with using simpler models. The specific workflows for feature engineering depend on the type of data involved.

For quantitative data, several common operations are performed:

- ***Transformations:*** applying a function (like log, capping, or flooring) can help when the data is skewed or when you want to make the data conform to more standard statistical distributions (a requirement for certain models).
- ***Binning:*** also known as discretization or bucketing, this process breaks down a continuous variable into discrete bins, enabling a reduction of noise associated with the data.
- ***Dimensionality Reduction:*** to generate a reduced set of uncorrelated features, you can use a technique like PCA.

Also, standardize and scale data as needed: this is especially important for machine learning algorithms that may be sensitive to variance in the feature values. For example, *K*-means uses Euclidean distance to measure distances between points and clusters, so all features need comparable variances. To normalize data, you can use min/max scaling, so that all data lies between zero and one. To standardize features, you can use z-scores, so that data has a mean of zero and a variance of one.

For categorical data, two common approaches are:

- ***One-hot encoding:*** turns each category into a vector of all zeroes, with the exception of a "one" for the category at hand
- ***Hashing:*** turns the data into a fixed dimensional vector using a hashing function; great for when features have a very high cardinality (large range of values) and the output vector of one-hot encoding would be too big

While you might not be asked about NLP during most interviews, if it's listed on your resume, it's good to know a few text preprocessing techniques as well:

- ***Stemming:*** reduces a word down to a root word by deleting characters (for example, turning the words "liked" and "likes" into "like").

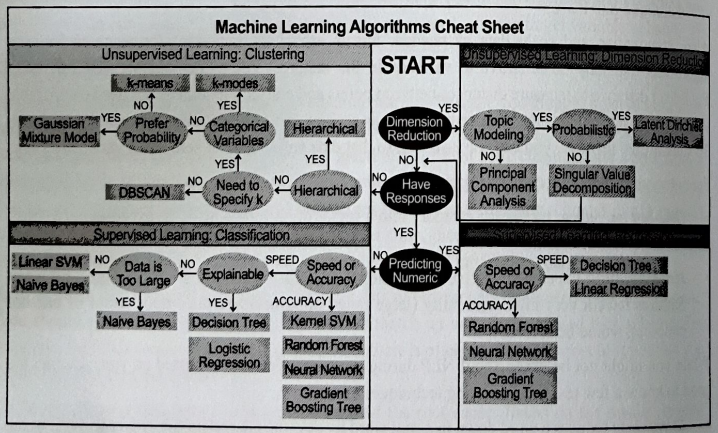
- ***Lemmatization:*** somewhat similar to stemming, but instead of just reducing words into roots, it takes into account the context and meaning of the word (for example, it would turn "caring" to "care," whereas stemming would turn "caring" to "car").
- ***Filtering:*** removes "stop words" that don't add value to a sentence — like "the" and "a", along with removing punctuation.
- ***Bag-of-words:*** represents text as a collection of words by associating each word and its frequency.
- ***N-grams:*** an extension of bag-of-words where we use N words in a sequence.
- ***Word embeddings:*** a representation that converts words to vectors that encode the meaning of the word, where words that are closer in meaning are closer in vector space (popular methods include word2vec and GloVe).

Step 7: Model Selection

Given the business constraints, evaluation metrics, and data sources available, what types of models make the most sense to try? Factors to consider when selecting a model include:

- ***Training & Prediction Speed:*** for example, linear regression is much quicker than neural networks for the same amount of data
- ***Budget:*** neural networks, for instance, can be computationally intensive models to train
- ***Volume & Dimensionality of Data:*** for example, neural networks can handle large amounts of data and higher-dimensional data versus *k-NN*)
- ***Categorical vs. Numerical Features:*** for example, linear regression cannot handle categorical variables directly, as they need to be one-hot encoded, versus trees (which can generally handle them directly)
- ***Explainability:*** choosing interpretable models like linear regression may be favorable to "black box" neural networks due to regulatory concerns or their ease of debugging

Below is a quick cheat sheet to also consider:



Step 8: Model Training & Evaluation

So you picked a type of model, or at least narrowed it down to a few candidates. At this point, interviewers will expect you to talk about how to train the model. This is a good time to mention the train-validation-test split, cross-validation, and hyper-parameter tuning.

You might also be asked how to compare your model to other models, how to learn its parameters, and how you'd know if its performance is good enough to ship. They might ask about edge cases, like handling biased training data, how you'd assess feature importance, or dealing with data imbalances.

The primary ways to deal with these issues, like picking the right model evaluation metric, using a validation set, regularizing your models to avoid overfitting, and using learning curves to find performance plateaus, are explained in-depth earlier in the chapter.

On the flip side, if you are dealing with so much data that your models are taking forever to train, then it is worth looking into various sampling techniques. The most common ones in practice are random sampling (sampling with or without replacement) and stratified sampling (sampling from specific groups among the entire population). In addition, for imbalanced datasets, undersampling and oversampling techniques are frequently used (SMOTE, for example). It is important to make sure you understand the sampling method at hand because a wrong sampling strategy may lead to incorrect results.

Step 9: Deployment

So, now that you've picked out a model, how do you deploy it? The process of operationalizing the entire deployment process is referred to as "MLOps" — when DevOps meets ML. Two popular tools in this space are Airflow and MLFlow. Because frameworks come and go, and many large tech companies use their own internal versions of these tools, it's rare to be asked about these specific technologies in interviews. However, knowledge of high-level deployment concepts is still helpful.

Generally, systems are deployed online, in batch, or as a hybrid of the two approaches. Online means latency is critical, and thus model predictions are made in real time. Since model predictions generally need to be served in real time, there will typically be a caching layer of cached features. Downsides for online deployment are that it can be computationally intensive to meet latency requirements and requires robust infrastructure monitoring and redundancy.

Batch means predictions are generated periodically and is helpful for cases where you don't need immediate results (most recommendation systems, for example) or require high throughput. But the downside is that batch predictions may not be available for new data (for example, a recommendation list cannot be updated until the next batch is computed). Ideally, you can work with stakeholders to find the sweet spot, where a batch predictor is updated frequently enough to be "good enough" to solve the problem at hand.

One deployment issue worth bringing up, that's common to both batch and online ML systems, is model degradation.Models degrade because the underlying distributions of data for your model change. For a concrete example, suppose you were working for a clothing e-commerce site and were training a product recommendation model in the winter time. Come summer, you might accidentally be recommending Canada Goose jackets in July — not a very relevant product suggestion to anyone besides Drake.

This feature drift leads to the phenomenon known as the *training-serving skew*, where there's a performance hit between the model in training and evaluation time versus when the model is served in production. To show awareness for the training-serving skew issue, be sure to mention to your

interviewer how often you'd retrain a model, what events might trigger a model refresh, and how much new data to use versus how much to rely on historical data. Also, mention how you'd add logging to monitor and catch model degradation.

Step 10: Iterate

Deploying a model isn't the end of the machine learning workflow. Business objectives change. Evaluation metrics change. Data features change. As such, you should plan to iterate on deployed models.

A big part of knowing how to iterate on your system can be learned via error analysis — the act of analyzing the wrong predictions manually. By looking at bad predictions and bucketing them into the types of reasons they occurred, you can better prioritize what projects to work on next. For example, say our traffic light detector tended to mislabel photos taken during rain — this tells you that you should collect more images in inclement weather. Or maybe this tells you to source whether it's raining or not as a feature for your model.

By focusing on iterating your design, and continuously seeking ways for your system to improve, you can generate increasing amounts of business value with your deployed models.

Machine Learning Interview Questions

As mentioned in the introduction, most machine learning interview questions take the simplistic form of "What does term X mean?" or "How does technique Y work?" To make the 35 interview questions more useful to you, dear reader, we intentionally shied away from including too many boring Google-able definitional questions. Instead, we chose to include the more interesting ML problems that showed up in real interviews.

In addition to the problems below, look to the multi-part case study problems in Chapter 11. We intentionally put these end-to-end machine learning modeling questions in the last chapter, because they incorporate system design and product-sense skills — concepts which we cover later.

Easy

7.1. Robinhood: Say you are building a binary classifier for an unbalanced dataset (where one class is much rarer than the other, say 1% and 99%, respectively). How do you handle this situation?

7.2. Square: What are some differences you would expect in a model that minimizes *squared* error versus a model that minimizes *absolute* error? In which cases would each error metric be appropriate?

7.3. Facebook: When performing K-means clustering, how do you choose k?

7.4. Salesforce: How can you make your models more robust to outliers?

7.5. AQR: Say that you are running a multiple linear regression and that you have reason to believe that several of the predictors are correlated. How will the results of the regression be affected if several are indeed correlated? How would you deal with this problem?

7.6. Point72: Describe the motivation behind random forests. What are two ways in which they improve upon individual decision trees?

7.7. PayPal: Given a large dataset of payment transactions, say we want to predict the likelihood of a given transaction being fraudulent. However, there are many rows with missing values for various columns. How would you deal with this?

7.8. Airbnb: Say you are running a simple logistic regression to solve a problem but find the results to be unsatisfactory. What are some ways you might improve your model, or what other models might you look into using instead?

7.9. Two Sigma: Say you were running a linear regression for a dataset but you accidentally duplicated every data point. What happens to your beta coefficient?

7.10. PWC: Compare and contrast gradient boosting and random forests.

7.11. DoorDash: Say that DoorDash is launching in Singapore. For this new market, you want to predict the estimated time of arrival (ETA) for a delivery to reach a customer after an order has been placed on the app. From an earlier beta test in Singapore, there were 10,000 deliveries made. Do you have enough training data to create an accurate ETA model?

Medium

7.12. Affirm: Say we are running a binary classification loan model, and rejected applicants must be supplied with a reason why they were rejected. Without digging into the weights of features, how would you supply these reasons?

7.13. Google: Say you are given a very large corpus of words. How would you identify synonyms?

7.14. Facebook: What is the bias-variance trade-off? How is it expressed using an equation?

7.15. Uber: Define the cross-validation process. What is the motivation behind using it?

7.16. Salesforce: How would you build a lead scoring algorithm to predict whether a prospective company is likely to convert into being an enterprise customer?

7.17. Spotify: How would you approach creating a music recommendation algorithm?

7.18. Amazon: Define what it means for a function to be convex. What is an example of a machine learning algorithm that is not convex and describe why that is so?

7.19. Microsoft: Explain what information gain and entropy are in the context of a decision tree and walk through a numerical example.

7.20. Uber: What is L1 and L2 regularization? What are the differences between the two?

7.21. Amazon: Describe gradient descent and the motivations behind stochastic gradient descent.

7.22. Affirm: Assume we have a classifier that produces a score between 0 and 1 for the probability of a particular loan application being fraudulent. Say that for each application's score, we take the square root of that score. How would the ROC curve change? If it doesn't change, what kinds of functions would change the curve?

7.23. IBM: Say X is a univariate Gaussian random variable. What is the entropy of X?

7.24. Stitch Fix: How would you build a model to calculate a customer's propensity to buy a particular item? What are some pros and cons of your approach?

7.25. Citadel: Compare and contrast Gaussian Naive Bayes (GNB) and logistic regression. When would you use one over the other?

Hard

7.26. Walmart: What loss function is used in k-means clustering given k clusters and n sample points? Compute the update formula using (1) batch gradient descent and (2) stochastic gradient descent for the cluster mean for cluster k using a learning rate ϵ.

7.27. Two Sigma: Describe the kernel trick in SVMs and give a simple example. How do you decide what kernel to choose?

7.28. Morgan Stanley: Say we have N observations for some variable which we model as being drawn from a Gaussian distribution. What are your best guesses for the parameters of the distribution?

7.29. Stripe: Say we are using a Gaussian mixture model (GMM) for anomaly detection of fraudulent transactions to classify incoming transactions into K classes. Describe the model setup formulaically and how to evaluate the posterior probabilities and log likelihood. How can we determine if a new transaction should be deemed fraudulent?

7.30. Robinhood: Walk me through how you'd build a model to predict whether a particular Robinhood user will churn?

7.31. Two Sigma: Suppose you are running a linear regression and model the error terms as being normally distributed. Show that in this setup, maximizing the likelihood of the data is equivalent to minimizing the sum of the squared residuals.

7.32. Uber: Describe the idea behind Principle Components Analysis (PCA) and describe its formulation and derivation in matrix form. Next, go through the procedural description and solve the constrained maximization.

7.33. Citadel: Describe the model formulation behind logistic regression. How do you maximize the log-likelihood of a given model (using the two-class case)?

7.34. Spotify: How would you approach creating a music recommendation algorithm for Discover Weekly (a 30-song weekly playlist personalized to an individual user)?

7.35. Google: Derive the variance-covariance matrix of the least squares parameter estimates in matrix form.

Machine Learning Solutions

Solution #7.1

Unbalanced classes can be dealt with in several ways.

First, you want to check whether you can get more data or not. While in many scenarios, data may be expensive or difficult to acquire, it's important to not overlook this approach, and at least mention it to your interviewer.

Next, make sure you're looking at appropriate performance metrics. For example, accuracy is not a correct metric to use when classes are imbalanced — instead, you want to look at precision, recall, F1 score, and the ROC curve.

Then, you can resample the training set by either oversampling the rare samples or undersampling the abundant samples; both can be accomplished via bootstrapping. These approaches are easy and quick to run, so they should be good starting points. Note, if the event is inherently rare, then oversampling may not be necessary, and you should focus more on the evaluation function.

Additionally, you could try generating synthetic examples. There are several algorithms for doing so — the most popular is called SMOTE (synthetic minority oversampling technique), which creates synthetic samples of the rare class rather than pure copies by selecting various instances. It does this by modifying the attributes slightly by a random amount proportional to the difference in neighboring instances.

Another way is to resample classes by running ensemble models with different ratios of the classes, or by running an ensemble model using all samples of the rare class and a differing amount of the abundant class. Note that some models, such as logistic regression, are able to handle unbalanced classes relatively well in a standalone manner. You can also adjust the probability threshold to something besides 0.5 for classifying the unbalanced outcome.

Lastly, you can design your own cost function that penalizes wrong classification of the rare class more than wrong classifications of the abundant class. This is useful if you have to use a particular kind of model and you're unable to resample. However, it can be complex to set up the penalty matrix, especially with many classes.

Solution #7.2

We can denote squared error as MSE and absolute error as MAE. Both are measures of distances between vectors and express average model prediction in units of the target variable. Both can range from 0 to infinity; the lower the score, the better the model.

The main difference is that errors are squared before being averaged in MSE, meaning there is a relatively high weight given to large errors. Therefore, MSE is useful when large errors in the model are trying to be avoided. This means that outliers disproportionately affect MSE more than MAE — meaning that MAE is more robust to outliers. Computation-wise, MSE is easier to use, since the gradient calculation is more straightforward than that of MAE, which requires some linear programming to compute the gradient.

Therefore, if the model needs to be computationally easier to train or needs to be robust to outliers, then MSE should be used. Otherwise, MAE is the better option. Lastly, MSE corresponds to maximizing the likelihood of Gaussian random variables, and MAE does not. MSE is minimized by the conditional mean, whereas MAE is minimized by the conditional *median*.

Solution #7.3

The elbow method is the most well-known method for choosing k in k-means clustering. The intuition behind this technique is that the first few clusters will explain a lot of the variation in the data, but past a certain point, the amount of information added is diminishing. Looking at a graph of explained variation (on the y-axis) versus the number of clusters (k), there should be a sharp change in the y-axis at some level of k. For example, in the graph that follows, we see a dropoff at approximately $k = 6$.

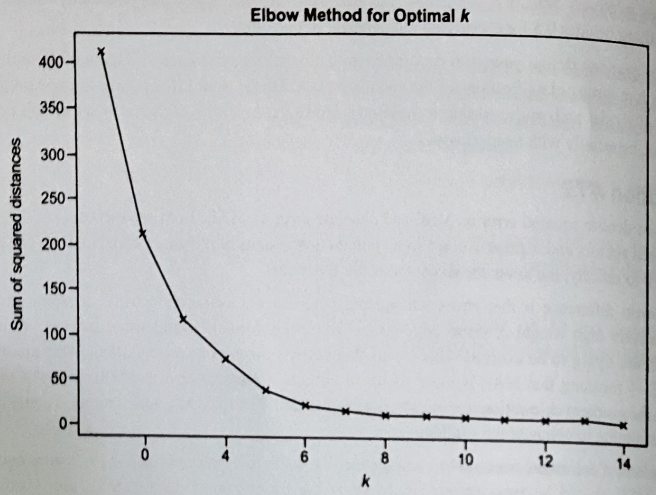
Note that the explained variation is quantified by the within-cluster sum of squared errors. To calculate this error metric, we look at, for each cluster, the total sum of squared errors (using Euclidean distance). A caveat to keep in mind: the assumption of a drop in variation may not necessarily be true — the y-axis may be continuously decreasing slowly (i.e., there is no significant drop).

Another popular alternative to determining k in k-means clustering is to apply the silhouette method, which aims to measure how similar points are in its cluster compared to other clusters. Concretely, it looks at:

$$\frac{(x-y)}{max(x,y)}$$

where x is the mean distance to the examples of the nearest cluster, and y is the mean distance to other examples in the same cluster. The coefficient varies between -1 and 1 for any given point. A value of 1 implies that the point is in the "right" cluster and vice versa for a score of -1. By plotting the score

on the y-axis versus k, we can get an idea for the optimal number of clusters based on this metric. Note that the metric used in the silhouette method is more computationally intensive to calculate for all points versus the elbow method.



Taking a step back, while both the elbow and silhouette methods serve their purpose, sometimes it helps to lean on your business intuition when choosing the number of clusters. For example, if you are clustering patients or customer groups, stakeholders and subject matter experts should have a hunch concerning how many groups they expect to see in the data. Additionally, you can visualize the features for the different groups and assess whether they are indeed behaving similarly. There is no perfect method for picking k, because if there were, it would be a supervised problem and not an unsupervised one.

Solution #7.4

Investigating outliers is often the first step in understanding how to treat them. Once you understand the nature of why the outliers occurred, there are several possible methods we can use:

- Add regularization: reduces variance, for example L1 or L2 regularization.
- Try different models: can use a model that is more robust to outliers. For example, tree-based models (random forests, gradient boosting) are generally less affected by outliers than linear models.
- Winsorize data: cap the data at various arbitrary thresholds. For example, at a 90% winsorization, we can take the top and bottom 5% of values and set them to the 95th and 5th percentile of values, respectively.
- Transform data: for example, do a log transformation when the response variable follows an exponential distribution or is right skewed.
- Change the error metric to be more robust: for example, for MSE, change it to MAE or Huber loss.

- Remove outliers: only do this if you're certain that the outliers are true anomalies not worth incorporating into the model. This should be the last consideration, since dropping data means losing information on the variability in the data.

Solution #7.5

There will be two primary problems when running a regression if several of the predictor variables are correlated. The first is that the coefficient estimates and signs will vary dramatically, depending on what particular variables you included in the model. Certain coefficients may even have confidence intervals that include 0 (meaning it is difficult to tell whether an increase in that X value is associated with an increase or decrease in Y or not), and hence results will not be statistically significant. The second is that the resulting p-values will be misleading. For instance, an important variable might have a high p-value and so be deemed as statistically insignificant even though it is actually important. It is as if the effect of the correlated features were "split" between them, leading to uncertainty about which features are actually relevant to the model.

You can deal with this problem by either removing or combining the correlated predictors. To effectively remove one of the predictors, it is best to understand the causes of the correlation (i.e., did you include extraneous predictors such as X and $2X$ or are there some latent variables underlying one or more of the ones you have included that affect both? To combine predictors, it is possible to include interaction terms (the product of the two that are correlated). Additionally, you could also (1) center the data and (2) try to obtain a larger size of sample, thereby giving you narrower confidence intervals. Lastly, you can apply regularization methods (such as in ridge regression).

Solution #7.6

Random forests are used since individual decision trees are usually prone to overfitting. Not only can these utilize multiple decision trees and then average their decisions, but they can be used for either classification or regression. There are a few main ways in which they allow for stronger out-of-sample prediction than do individual decision trees.

- As in other ensemble models, using a large set of trees created in a resample of the data (bootstrap aggregation) will lead to a model yielding more consistent results. More specifically, and in contrast to decision trees, it leads to diversity in training data for each tree and so contributes to better results in terms of bias-variance trade-off (particularly with respect to variance).
- Using only $m < p$ features at each split helps to de-correlate the decision trees, thereby avoiding having very important features always appearing at the first splits of the trees (which would happen on standalone trees due to the nature of information gain).
- They're fairly easy to implement and fast to run.
- They can produce very interpretable feature-importance values, thereby improving model understandability and feature selection.
- Points A and B above are the main keys for improving upon single decision trees.

Solution #7.7

Step 1: Clarify the Missing Data

Since these types of problems are generally context dependent, it's best to start your answer with clarifying questions. For example,

- Is the amount of missing values uniform by feature?

- Are these missing values numerical or categorical?
- How many features with missing data are there?
- Is there a pattern in the types of transactions that have a lot of missing data?

It would also be useful to think about why the data is missing, because this affects how you'd impute the data. Missing data is commonly classified as:

- ***Missing completely at random (MCAR):*** the probability of being missing is the same for all classes
- ***Missing at random (MAR):*** the probability of being missing is the same within groups defined by the observed data
- ***Not missing at random (NMAR):*** if the data is not MCAR and not MAR

Step 2: Establish a Baseline

One reason to ask these questions is because a good answer would consider that the missing data may not actually be a problem. What if the missing data was in transactions that were almost never fraud? What if the missing data is mostly in columns whose data features don't have good predictive value? For example, if you were missing the IP-address derived geolocation of the person making the payment, that would likely be bad for model performance. On the other hand, if you were missing the user's middle name, it likely wouldn't have any bearing on whether the transaction was fraud or not. Even simpler yet, can a baseline model be built that meets the business goals, without having to deal with any missing data?

Step 3: Impute Missing Data

If the baseline model indicates that dealing with the missing data is worth it, one technique we could use is imputation. For continuous features, we can start by using the mean or median for missing values within any feature. However, the downside to this approach is that it does not factor in any of the other features and correlations between them — it is unlikely that two transactions in differing locations for different category codes would have the same transaction price. An alternative could be to use a nearest neighbors method to estimate any given feature based on the other features available.

Step 4: Check Performance with Imputed Data

With these modeled features, we can then run a set of classification algorithms to predict fraud or not fraud. With this imputation technique, we can also cross-validate to check whether performance increases by including the imputed data, relative to just the original data. Note that a performance increase is only expected if the feature contains valuable information (for those rows/entries that have it). If this isn't the case, and you won't see a significant impact as a result, it may be easiest to drop the existing missing data before training the model.

Step 5: Other Approaches for Missing Data

Finally, thinking outside the box, is it possible to use a third-party dataset to fill in some of the missing information? Suppose a common missing piece of information was the type of business that a person paid. But, let's say we have the business's address on file. Could we use the address against a business listings dataset to infer the type of business the transaction was made at? Lastly, note that some models are capable of dealing with missing data, without requiring imputation.

Solution #7.8

There are several possible ways to improve the performance of a logistic regression:

- ***Normalizing features:*** The features should be normalized such that particular weights do not dominate within the model.
- ***Adding additional features:*** Depending on the problem, it may simply be the case that there aren't enough useful features. In general, logistic regression is high bias, so adding more features should be helpful.
- ***Addressing outliers:*** Identify and decide whether to retain or remove them.
- ***Selecting variables:*** See if any features have introduced too much noise into the process.
- ***Cross validation and hyperparameter tuning:*** Using k-fold cross validation along with hyperparameter tuning (for example, introducing a penalty term for regularization purposes) should help improve the model.
- The classes may not be linearly separable (logistic regression produces linear decision boundaries), and, therefore, it would be worth looking into SVMs, tree-based approaches, or neural networks instead.

Solution #7.9

For regular regression, recall we have the following for our least squares estimator:

$$\beta = (X^T X)^{-1} X^T y$$

So if we double the data, then we are using: $\begin{pmatrix} X \\ X \end{pmatrix}, \begin{pmatrix} Y \\ Y \end{pmatrix}$

instead of and respectively. Then plugging this into our estimator from above, we get:

$$\beta = \left(\begin{pmatrix} X \\ X \end{pmatrix}^T \begin{pmatrix} X \\ X \end{pmatrix} \right)^{-1} \begin{pmatrix} X \\ X \end{pmatrix}^T \begin{pmatrix} Y \\ Y \end{pmatrix}$$

Simplifying yields:

$$\beta = (2X^T X)^{-1} 2X^T y$$

Therefore, we see that the coefficient remains unchanged.

Solution #7.10

In both gradient boosting and random forests, an ensemble of decision trees are used. Additionally, both are flexible models and don't need much data preprocessing.

However, there are two main differences. The first main difference is that, in gradient boosting, trees are built one at a time, such that successive weak learners learn from the mistakes of preceding weak learners. In random forests, the trees are built independently at the same time.

The second difference is in the output: gradient boosting combines the results of the weak learners with each successive iteration, whereas, in random forests, the trees are combined at the end (through either averaging or majority).

Because of these structural differences, gradient boosting is often more prone to overfitting than are random forests due to their focus on mistakes over training iterations and the lack of independence in tree building. Additionally, gradient boosting hyperparameters are harder to tune than those of

random forests. Lastly, gradient boosting may take longer to train than random forests because the trees of the latter are built sequentially. In real-life applications, gradient boosting generally excels when used on unbalanced datasets (fraud detection, for example), whereas random forests generally excel at multi-class object detection with noisy data (computer vision, for example).

Solution #7.11

Because "accurate enough" is subjective, it's best to ask the interviewer clarifying questions before addressing the lack of training data. To stand out, you can also proactively mention ways to source more training data at the end of your answer.

Step 1: Clarify What "Good" ETA Means

To determine how accurate the ETA model needs to be, first ask what the ETA prediction will be used for. The level of accuracy needed in ETA predictions might be higher for the order-driver matching algorithm than what DoorDash needs to display to the customer in the app. Also, consider if your ETA estimate under-promises and over-delivers. Maybe that's okay — customers would likely be happy that the delivery arrived faster than expected. At the same time, high ETA estimates across the board may lead people to say, "Screw it, I'll just go to the store and pick it up myself." By considering the context around how the ETA predictions will be used, you'll be one step closer to understanding what a good-enough ETA model looks like.

One data-driven approach to establishing how accurate your ETA model needs to be involves looking at ETA models in similar markets. From data in other locations, we could better understand the economic impact of both over and underestimated ETAs. Tying the model output to its business impact can help DoorDash decide if investing money into solving this problem is even warranted in the first place.

Step 2: Assess Baseline ETA Performance

After you understand what "good-enough" ETA means, it's best practice to next see how a baseline model, trained on the beta 10,000 deliveries made, performs. A baseline model can be something as simple as the estimated driving time plus the average preparation time (conditional on the restaurant and time of day). Since predicting the ETAs is a regression problem, potential metrics we can use to assess this baseline model include root mean square error (RMSE), MAE, R^2, etc.

Step 3: Determine How More Data Improves Accuracy

Say that we use R^2 as the main metric. One way to gauge the benefit from having more training data would be to build learning curves. A learning curve depicts how the accuracy changes when we train a model on a progressively larger percentage of the data. For example, say that with 25% of the data, we get an R^2 of 0.5, with 50% of the data we get an R^2 of 0.65, and with 75% of the data we get an R^2 of 0.67. Note that the improvement drops off significantly between the use of 50% and 75% of the training data. The point at which the drop-off starts to become a problem is the signal that we should look into reevaluating the features rather than simply adding more training data.

This process is analogous to looking at the learning curves discussed earlier in this chapter, which look at how the training error and validation error change over the number of iterations — here, instead, we are looking at how the model performance changes over the amount of training data used.

Step 4: In Case Performance Isn't "Good Enough"

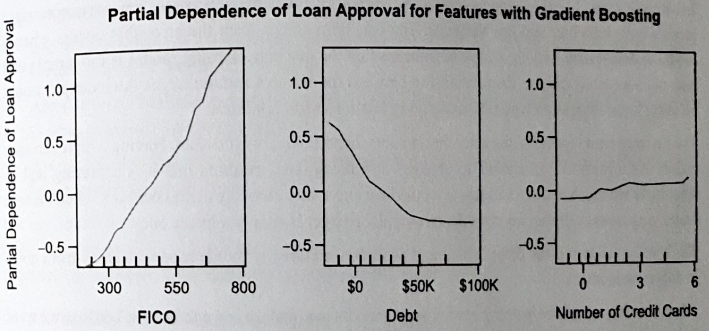
If after learning curves you realize that you don't have sufficient data to build an accurate enough model, the interviewer would likely shift the discussion to dealing with this lack of data. Or, if

you are feeling like an overachiever with a can-do attitude, you could proactively bring up these discussion points:

- Are there too few features? If so, you want to look into adding features like marketplace supply and demand indicators, traffic patterns on the road at the time of the delivery, etc.
- Are there too many features? If there are almost as many or more features than data points, then our model will be prone to overfitting and we should look into either dimensionality reduction or feature selection techniques.
- Can different models be used that handle smaller training datasets better?
- Is it possible to acquire more data in a cost-effective way?
- Is the less accurate ETA model a true launch blocker? If we launched in the new market, which generates more training data, can the ETA model be retrained?

Solution #7.12

Without looking at features, we could look at partial dependence plots (also called response curves) to assess how any one feature affects the model's decision. A partial dependence plot shows the marginal effect of a feature on the predicted target of a machine learning model. So, after the model is fit, we can take all the features and start plotting them individually against the loan approval/rejection, while keeping all the other features constant.



For example, if we believe that FICO score has a strong relationship to the predicted probability of loan rejection, then we can plot the loan approvals and rejections as we adjust the FICO score from low to high. Thus, we can get an idea of how features impact the model without explicitly looking at feature weights, and supply reasons for rejection accordingly.

As a concrete example, consider having four applicants: 1, 2, 3, and 4. Assume that the features include annual income, current debt, number of credit cards, and FICO score. Suppose we have the following situation:

1. $100,000 income, $10,000 debt, 2 credit cards, and FICO score of 700.
2. $100,000 income, $10,000 debt, 2 credit cards, and FICO score of 720.
3. $100,000 income, $10,000 debt, 2 credit cards, and FICO score of 600.
4. $100,000 income, $10,000 debt, 2 credit cards, and FICO score of 650.

If 3 and 4 were rejected but 1 and 2 were accepted, then we can intuitively reason that a lower FICO score was the reason the model made the rejections. This is because the remaining features are equal, so the model chose to reject 3 and 4 "all-else-equal" versus 1 and 2.

Solution #7.13

To find synonyms, we can first find word embeddings through a corpus of words. Word2vec is a popular algorithm for doing so. It produces vectors for words based on the words' contexts. Vectors that are closer in Euclidean distance are meant to represent words that are also closer in context and meaning. The word embeddings that are thus generated are weights on the resulting vectors. The distance between these vectors can be used to measure similarity, for example, via cosine similarity or some other similar measure.

Once we have these word embeddings, we can then run an algorithm such as *K*-means clustering to identify clusters within word embeddings or run a *K*-nearest neighbor algorithm to find a particular word for which we want to find synonyms. However, some edge cases exist, since word2vec can produce similar vectors even in the case of antonyms; consider the words "hot" and "cold," for example, which have opposite meanings but appear in many similar contexts (related to temperature or in a Katy Perry song).

Solution #7.14

The bias-variance trade-off is expressed as the following: Total model error = Bias + Variance + Irreducible error. Flexible models tend to have low bias and high variance, whereas more rigid models have high bias and low variance. The bias term comes from the error that occurs when a model underfits data. Having a high bias means that the machine learning model is too simple and may not adequately capture the relationship between the features and the target. An example would be using linear regression when the underlying relationship is nonlinear.

The variance term represents the error that occurs when a model overfits data. Having a high variance means that a model is susceptible to changes in training data, meaning that it is capturing and so reacting to too much noise. An example would be using a very complex neural network when the true underlying relationship between the features and the target is simply a linear one.

The irreducible term is the error that cannot be addressed directly by the model, such as from noise in data measurement.

When creating a machine learning model, we want both bias and variance to be low, because we want to be able to have a model that predicts well but that also doesn't change much when it is fed new data. The key point here is to prevent overfitting and, at the same time, to attempt to retain sufficient accuracy.

Solution #7.15

Cross validation is a technique used to assess the performance of an algorithm in several resamples/subsamples of training data. It consists of running the algorithm on subsamples of the training data, such as the original data less some of the observations comprising the training data, and evaluating model performance on the portion of the data that was not present in the subsample. This process is repeated many times for the different subsamples, and the results are combined at the end. This step is very important in ML because it reveals the quality and consistency of the model's true performance.

The process is as follows:

1. Randomly shuffle data into k equally-sized blocks (folds).
2. For each i in fold 1...k, train the model on all the data except for fold i, and evaluate the validation error using block i.
3. Average the k validation errors from step 2 to get an estimate of the true error.

This process aids in accomplishing the following: (1) avoiding training and testing on the same subsets of data points, which would lead to overfitting, and (2) avoiding using a dedicated validation set, with which no training can be done. The second of these points is particularly important in cases where very little training data is available or the data collection process is expensive. One drawback of this process, however, is that roughly k times more computation is needed than using a dedicated holdout validation set. In practice, cross validation works very well for smaller datasets.

Solution #7.16

Step 1: Clarify Lead Scoring Requirements

Lead scoring is the process of assigning numerical scores for any leads (potential customers) in a business. Lead scores can be based on a variety of attributes, and help sales and marketing teams prioritize leads to try and convert them to customers.

As always, it's smart to ask the interviewer clarifying questions. In this case, learning more about the requirements for the lead scoring algorithm is critical. Questions to ask include:

- Are we building this for our own company's sales leads? Or, are we building an extensible version as part of the Salesforce product?
- Are there any business requirements behind the lead scoring (i.e., does it need to be easy to explain internally and/or externally)?
- Are we running this algorithm only on companies in our sales database (CRM), or looking at a larger landscape of all companies?

For our solution, we'll assume the interviewer means we want to develop a leading scoring model to be used internally — that means using the company's internal sales data to predict whether a prospective company will purchase a Salesforce product.

Step 2: Explain the Features You'd Use

Some elements which should influence whether a prospective company turns into a customer:

- ***Firmographic Data:*** What type of company is this? Industry? Amount of revenue? Employee count?
- ***Marketing Activity:*** Have they interacted with marketing materials, like clicking on links within email marketing campaigns? Have employees from that company downloaded whitepapers, read case studies, or clicked on ads? If so, how much activity has there been recently?
- ***Sales Activity:*** Has the prospective company interacted with sales? How many sales meetings took place, and how recently did the last one take place?
- ***Deal Details:*** What products are being bought? Some might be harder to close than others. How many seats (licenses) are being bought? What's the size of the deal? What's the contract length?

After selecting features, it is good to conduct the standard set of feature engineering best practices. Note that the model will only be as good as the data and judgement in feature engineering applied — in practice, many companies that predict lead scoring can face issues with missing data or lack of relevant data.

Step 3: Explain Models You'd Use

Lead scoring can be done through building a binary classifier that predicts the probability of a lead converting. In terms of model selection, logistic regression offers a straightforward solution with an easily interpretable result: the resulting log-odds is a probability score for, in this case, purchasing a particular item. However, it cannot capture complex interaction effects between different variables and could also be numerically unstable under certain conditions (i.e., correlated covariates and a relatively small user base).

An alternative to logistic regression would be to use a more complex model, such as a neural network or an SVM. Both are great for dealing with high-dimensional data and with capturing the complex interactions that logistic regression cannot. However, unlike logistic regression, neither is easy to explain, and both generally require a large amount of data to perform well.

A suitable compromise is tree-based models, such as random forests or XGBoost, which typically perform well. With tree-based models, the features that have the highest influence on predictions are readily perceived, a characteristic that could be very useful in this particular case.

Step 4: Model Deployment Nuances

Lastly, it is important to monitor for feature shifts and/or model degradations. As the product line and customer base changes over time, models trained on old data may not be as relevant. For a mature company like Salesforce, for example, it's very likely that companies signing up now aren't exactly like the companies that signed up with Salesforce 5 or 10 years ago. That's why it's important to monitor feature drift and continuously update the model.

Solution #7.17

Collaborative filtering would be a commonly used method for creating a music recommendation algorithm. Such algorithms use data on what feedback users have provided on certain items (songs in this case) in order to decide recommendations. For example, a well-known use case is for movie recommendation on Netflix. However, there are several differences compared to the Netflix case:

- Feedback for music does not have a 1-to-5 rating scale as Netflix does for its movies.
- Music may be subject to repeated consumption; that is, people may watch a movie once or twice but will listen to a song many times over.
- Music has a wider variety (i.e., niche music).
- The scale of music catalog items is much larger than movies (i.e., there are many more songs than movies).

Therefore, a user-song matrix (or a user-artist matrix) would constitute the data for this issue, with the rows of the dataset being users and the columns various songs. However, in considering the first point above, since explicit ratings are lacking, we can employ a binary system to count the number of times a song is streamed and store this count.

We can then proceed with matrix factorization. Say there are M songs and N users in the matrix, which we will label R. Then, we want to solve: $R = PQ^T$

where user preferences are captured by the vectors: $r_{ui} = q_i^T p_u$

Various methods can be used for this matrix factorization; a common one is alternating least squares (ALS), and, since the scale of the data is large, this would likely be done through distributed computing. Once the latent user and song vectors are discovered, then the above dot product will be able to predict the relevance of a particular song to a user. This process can be used directly for recommendation at the user level, where we sort by relevance prediction on songs that the user has not yet streamed. In addition, the vectors given above can be employed in such tasks as assessing similarity between different users and different songs using a method such as *kNN* (*K*-nearest neighbors).

Solution #7.18

Mathematically, a convex function f satisfies the following for any two points x and y in the domain of f: $f((1-\alpha)x + \alpha y) \leq (1-\alpha)f(x) + \alpha f(y)$, $0 \leq \alpha \leq 1$

That is, the line segment from x to y lies above the function graph of f for any points x and y. Convexity matters because it has implications about the nature of minima in f. Stated more specifically, any local minimum of f is also a global minimum.

Neural networks provide a significant example of non-convex problems in machine learning. At a high level, this is because neural networks are universal function approximators, meaning that they can (with a sufficient number of neurons) approximate any function arbitrarily well. Because not all functions are convex (convex functions cannot approximate non-convex ones), by definition, they must be non-convex. In particular, the cost function for a neural network has a number of local minima; you could interchange parameters of different nodes in various layers and still obtain exactly the same cost function output (all inputs/outputs the same, but with nodes swapped). Therefore, there is no particular global minima, so neural networks cannot be convex.

Solution #7.19

Because information gain is based on entropy, we'll discuss entropy first.

The formula for entropy is $Entropy = \sum_{i=1} kP(Y=k)\log P(Y=k)$

The equation above yields the amount of entropy present and shows exactly how homogeneous a sample is (based on the attribute being split). Consider a case where $k = 2$. Let a and b be two outputs/labels that we are trying to classify. Given these values, the formula considers the proportion of values in the sample that are a and the proportion that are b, with the sample being split on a different attribute.

A completely homogeneous sample will have an entropy of 0. For instance, if a given attribute has values a and b, then the entropy of splitting on that given attribute would be

$$Entropy = -1 * \text{Log}_2(1) - 0 * \text{Log}_2(0) = 0$$

whereas a completely split (50%-50%) would result in an entropy of 1. A lower entropy means a more homogeneous sample.

Information gain is based on the decrease in entropy after splitting on an attribute.

$$IG(X_j, Y) = H(Y) - H(Y|X_j)$$

This concept is better explained with a simple numerical example. Consider the above case again with $k = 2$. Let's say there are 5 instances of value a and 5 instances of value b. Then, we decide to split on some attribute X. When $X = 1$, there are 5 a's and 1 b, whereas when $x = 0$. there are 4 b's and 0 a's.

Now, by splitting on X, we have two classes: $X = 1$ and $X = 0$. However, by splitting on this attribute, we now have $X = 1$, which has 5 a's and 1 b, while $X = 0$ has 4 b's and 0 a's.

$$\text{Entropy(After)} = \left(\frac{4}{10}\right) * \text{Entropy}(X=0) - \left(\frac{6}{10}\right) * \text{Entropy}(X=1)$$

The entropy value for $X = 0$ is 0, since the sample is homogeneous (all b's, no a's).

$$\text{Entropy}(X=0) = 0$$

$$\text{Entropy}(X=1) = -\left(\frac{1}{6}\right) * Log_2\left(\frac{1}{6}\right) - \left(\frac{5}{6}\right) * Log_2\left(\frac{5}{6}\right) = .65$$

Plug these into the entropy(after) formula to obtain the following:

$$\text{Entropy(After)} = \left(\frac{4}{10}\right) * 0 - \left(\frac{6}{10}\right) * .65 = .39$$

Finally, we can go back to our original formula and obtain information gain: $IG = 1 - .39 = .61$

These results make intuitive sense, since, ideally, we want to split on an attribute that splits the output perfectly. Therefore, we ideally want to split on something that is homogeneous with regards to the output, and this something would thus have an entropy equal to 0.

Solution #7.20

In machine learning, L_1 and L_2 penalization are both regularization methods that prevent overfitting by coercing the coefficients of a regression model towards zero. The difference between the two methods is the form of the penalization applied to the loss function. For a regular regression model, assume the loss function is given by L. Using L_1 regularization, the least absolute shrinkage and selection operator, or Lasso, adds the absolute value of the coefficients as a penalty term, whereas ridge regression uses L_2 regularization, that is, adding the squared magnitude of the coefficients as the penalty term.

The loss function for the two are thus the following:

$$\text{Loss}(L_1) = L + \lambda \mid w_i \mid \qquad \text{Loss}(L_2) = L + \lambda \mid w_i^2 \mid$$

where the loss function L is the sum of errors squared, given by the following, where $f(x)$ is the model of interest — for example, a linear regression with p predictors:

$$L = \sum_{i=1}^{n} \left(y_i - f(x_i)\right)^2 = \sum_{i=1}^{n} \left(y_i - \sum_{j=1}^{p} (x_0 w_j)\right)^2 \text{ for linear regression}$$

If we run gradient descent on the weights w, L_1 regularization forces any weight closer to 0, irrespective of its magnitude, whereas with L_2 regularization, the rate at which the weight approaches 0 becomes slower as the weight approaches 0. Because of this, L_1 is more likely to "zero" out particular weights and hence completely remove certain features from the model, leading to models of increased sparseness.

Solution #7.21

The gradient descent algorithm takes small steps in the direction of steepest descent to optimize a particular objective function. The size of the "steps" the algorithm takes are proportional to the negative gradient of the function at the current value of the parameter being sought. The stochastic version of the algorithm, SGD, uses an approximation of the nonstochastic gradient descent algorithm instead of the function's actual gradient. This estimate is done by using only one randomly selected sample at each step to evaluate the derivative of the function, making this version of the algorithm much faster and more attractive for situations involving lots of data. SGD is also useful when redundancy in the data is present (i.e., observations that are very similar).

Assume function f at some point x and at time t. Then, the gradient descent algorithm will update x as follows until it reaches convergence:

$$x^{t+1} = x^t - \alpha_t \nabla f(x^t)$$

That is, we calculate the negative of the gradient of f and scale that by some constant and move in that direction at the end of each iteration.

Since many loss functions are decomposable into the sum of individual functions, then the gradient step can be broken down into addition of discrete, separate gradients. However, for very large datasets, this process can be computationally intensive, and the algorithm might become stuck at local minima or at saddle points.

Therefore, we use the stochastic gradient descent algorithm to obtain an unbiased estimate of the true gradient without going through all data points by uniformly selecting a point at random and performing a gradient update then and there.

The estimate is therefore unbiased since we have the following:

$$\nabla f(x) = \frac{1}{n} \sum_{i=1}^{n} \nabla f_i(x)$$

Since the data are assumed to be i.i.d., for the SGD, the expectation of $g(x)$ is: $E[g(x)] = \nabla f(x)$ where $g(x)$ is the stochastic gradient descent.

Solution #7.22

Recall that the ROC curve plots the true positive rate versus the false positive rate. If all scores change simultaneously, then none of the actual classifications change (since thresholds are adjusted), leading to the same true positive and false positive rates, since only the relative ordering of the scores matters. Therefore, taking a square root would not cause any change to the ROC curve because the relative ordering has been maintained. If one application had a score of X and another a score of Y, and if $Y > X$, then $\sqrt{Y} > \sqrt{X}$ still. Only the model thresholds would change.

In contrast, any function that is not monotonically increasing would change the ROC curve, since the relative ordering would not be maintained. Some simple examples are the following:

$f(x) = -x$, $f(x) = -x^2$, or a stepwise function.

Solution #7.23

We have: $X \sim N(\mu, \sigma^2)$, and entropy for a continuous random variable is given by the following:

$$H(x) = -\int_{-\infty}^{\infty} p(x) \log p(x) dx$$

For a Gaussian, we have the following: $p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(x-\mu)}{2\sigma^2}}$

Substituting into the above equation yields

$$H(x) = -\int_{-\infty}^{\infty} p(x) \log \sigma\sqrt{2\pi} dx - \int_{-\infty}^{\infty} p(x) \left(-\frac{(x-\mu)}{2\sigma^2}\right) \log(e) dx$$

where the first term equals $-\log \sigma\sqrt{2\pi} \int_{-\infty}^{\infty} p(x) dx = -\log \sigma\sqrt{2\pi}$

since the integral evaluates to 1 (by the definition of a probability density function). The second term is given by:

$$\frac{1}{2\sigma^2}\int_{-\infty}^{\infty} p(x)(x-\mu)^2\,dx = \frac{\sigma^2}{2\sigma^2} = \frac{1}{2}$$

since the inner term is the expression for the variance. The entropy is therefore as follows:

$$H(x) = \frac{1}{2} + \log\sigma\sqrt{2\pi}$$

Solution #7.24

The standard approach is (1) to construct a large dataset with the variable of interest (purchase or not) and relevant covariates (age, gender, income, etc.) for a sample of platform users and (2) to build a model to calculate the probability of purchase of each item. Propensity models are a form of binary classifier, so any model that can accomplish this could be used to estimate a customer's propensity to buy the product.

In selecting a model, logistic regression offers a straightforward solution with an easily interpretable result: the resulting log-odds is a probability score for, in this case, purchasing a particular item. However, it cannot capture complex interaction effects between different variables and could also be numerically unstable under certain conditions (i.e., correlated covariates and a relatively small user base).

An alternative to logistic regression would be to use a more complex model, such as a neural network or an SVM. Both are great with dealing with high-dimensional data and with capturing the complex interactions that logistic regression cannot. However, unlike logistic regression, neither is easy to explain, and both generally require a large amount of data to perform well.

A good compromise is tree-based models, such as random forests, which are typically highly accurate and are easily understandable. With tree-based models, the features which have the highest influence on predictions are readily perceived, a characteristic that could be very useful in this particular case.

Solution #7.25

Both Gaussian naive Bayes (GNB) and logistic regression can be used for classification. The two models each have advantages and disadvantages, which provide the answer as to which to choose under what circumstances. These are discussed below, along with their similarities and differences:

Advantages:

1. GNB requires only a small number of observations to be adequately trained; it is also easy to use and reasonably fast to implement; interpretation of the results produced by GNB can also be highly useful.
2. Logistic regression has a simple interpretation in terms of class probabilities, and it allows inferences to be made about features (i.e., variables) and identification of the most relevant of these with respect to prediction.

Disadvantages:

1. By assuming features (i.e., variables) to be independent, GNB can be wrongly employed in problems where that does not hold true, a very common occurrence.
2. Not highly flexible, logistic regression may fail to capture interactions between features and so may lose prediction power. This lack of flexibility can also lead to overfitting if very little data are available for training.

Differences:

1. Since logistic regression directly learns $P(Y|X)$, it is a discriminative classifier, whereas GNB directly estimates $P(Y)$ and $P(X|Y)$ and so is a generative classifier.

2. Logistic regression requires an optimization setup (where weights cannot be learned directly through counts), whereas GNB requires no such setup.

Similarities:

1. Both methods are linear decision functions generated from training data.
2. GNB's implied $P(Y|X)$ is the same as that of logistic regression (but with particular parameters).

Given these advantages and disadvantages, logistic regression would be preferable assuming training provided data size is not an issue, since the assumption of conditional independence breaks down if features are correlated. However, in cases where training data are limited or the data-generating process includes strong priors, using GNB may be preferable.

Solution #7.26

Assume we have k clusters and n sample points: $x_1 \ldots x_n$; $\mu_1 \ldots \mu_k$

The loss function then consists of minimizing total error using a squared L_2 norm (since it is a good way to measure distance) for all points within a given cluster:

$$L = \sum_{j=1}^{k} \sum_{x_i \in S_j} \|x_i - \mu_j\|^2$$

Taking the derivatives yields the following: $\frac{\partial L}{\partial \mu_k} = \frac{\partial}{\partial \mu_k} \sum_{x_i \in S_k} (x_i - \mu_k)^T (x_i - \mu_k) = \sum_{x_i \in S_k} 2(x_i - \mu_k)$

For batch gradient descent, the update step is then given by the following:

$$\mu_k = \mu_k + \epsilon \sum_{x_i \in S_k} 2(x_i - \mu_k)$$

However, for stochastic gradient descent, the update step is given by the following:

$$\mu_k = \mu_k + \epsilon(x_i - \mu_k)$$

Solution #7.27

The idea behind the kernel trick is that data that cannot be separated by a hyperplane in its current dimensionality can actually be linearly separable by projecting it onto a higher dimensional space.

$$k(x, y) = \phi(x)^T \phi(y)$$

and we can take any data and map that data to a higher dimension through a variety of functions ϕ. However, if ϕ is difficult to compute, then we have a problem — instead, it is desirable if we can compute the value of k without blowing up the computation.

For instance, say we have two examples and want to map them to a quadratic space. We have the following:

$$\phi(x_1, x_2) = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_2^2 \\ x_1 x_2 \end{bmatrix}$$

and we can use the following: $k(x, y) = (1 + x^T y)^2 = \phi(x)^T \phi(y)$

If we now change $n = 2$ (quadratic) to arbitrary n, we can have arbitrarily complex ϕ. As long as we perform computations in the original feature space (without a feature transformation), then we avoid the long compute time while still mapping our data to a higher dimension!

In terms of which kernel to choose, we can choose between linear and nonlinear kernels, and these will be for linear and nonlinear problems, respectively. For linear problems, we can try a linear or logistic kernel. For nonlinear problems, we can try either radial basis function (RBF) or Gaussian kernels.

In real-life problems, domain knowledge can be handy — in the absence of such knowledge, the above defaults are probably good starting points. We could also try many kernels, and set up a hyperparameter search (a grid search, for example) and compare different kernels to one another. Based on the loss function at hand, or certain performance metrics (accuracy, F1, AUC of the ROC curve, etc.), we can determine which kernel is appropriate.

Solution #7.28

Assume we have some dataset X consisting of n i.i.d observations: $x_1, ..., x_n$

Our likelihood function is then $p(X|\mu,\sigma^2) = \prod_{i=1}^{n} N(x_i|\mu,\sigma^2)$ where $N(x_i|\mu,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{\frac{(x_i-\mu)}{2\sigma^2}}$

and therefore the log-likelihood is given by:

$$\log p(X|\mu,\sigma^2) = \sum_{i=1}^{n} \log N(x_i|\mu,\sigma^2) = -\frac{1}{2\sigma^2}\sum_{i=1}^{n}(x_i-\mu)^2 - \frac{n}{2}\log\sigma^2 - \frac{n}{2}\log\pi$$

Taking the derivative of the log-likelihood with respect to μ and setting the result to 0 yields the following:

$$\frac{d\log p(X|\mu,\sigma^2)}{d\mu} = \frac{1}{\sigma^2}\sum_{i=1}^{n}(x_i-\mu) = 0$$

Simplifying the result yields: $\sum_{i=1}^{n} x_i = \sum_{i=1}^{n} \hat{\mu} = n\hat{\mu}$, and therefore the maximum likelihood estimate for μ is given by:

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i$$

To obtain the variance, we take the derivative of the log likelihood with respect to σ^2 and set the result equal to 0

$$\frac{d\log p(X|\mu,\sigma^2)}{d\sigma^2} = \frac{1}{2\sigma^4}\sum_{i=1}^{n}(x_i-\mu)^2 - \frac{n}{2\sigma^2} = 0$$

Simplifying yields the following: $\frac{1}{2\sigma^4}\sum_{i=1}^{n}(x_i-\mu)^2 = \frac{n}{2\sigma^2}$

$$\sum_{i=1}^{n}(x_i-\mu) = n\sigma^2$$

The maximum likelihood estimate for the variance is thus given by the following:

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i-\hat{\mu})^2$$

Solution #7.29

The GMM model assumes a Gaussian probability distribution function, across K classes:

$$p(x) = \sum_{k=1}^{K} \pi_k N(x|\mu_k,\Sigma_k)$$

where the π coefficients are the mixing coefficients on the clusters and are normalized so that they sum to 1.

The posterior probabilities for each cluster are given by Bayes' rule and can be interpreted as "what is the probability of being in class k given the data x":

$$z_k = p(k|x) = \frac{p(k)\,p(x|k)}{p(x)}$$

and hence: $z_k = p(k|x) = \frac{\pi_k N(x|\mu_k,\Sigma_k)}{\sum_{k=1}^{K}\pi_k N(x|\mu_k,\Sigma_k)}$

The unknown set of parameters θ consists of the mean and variance parameters for each of the K classes, along with the K coefficients. The likelihood is therefore given by:

$$p(\theta|X) = \prod_{i=1}^{n} p(x) = \prod_{i=1}^{n}\sum_{k=1}^{K}\pi_k N(x|\mu_k,\Sigma_k)$$

and therefore the log-likelihood is $\log p(\theta|X) = \sum_{i=1}^{n}\log\sum_{k=1}^{K}\pi_k N(x|\mu_k,\Sigma_k)$

The parameters can be calculated iteratively using expectation-maximization and the information above. After the model has been trained, for any new transaction we can then calculate the posterior probabilities of any new transactions over the K classes as above. If the posterior probabilities calculated are low, then the transaction most likely does not belong to any of the K classes, so we can deem it to be fraudulent.

Solution #7.30

Step 1: Clarify What Churn Is & Why It's Important

First, it is important to clarify with your interviewer what churn means. Generally, the word "churn" defines the process of a platform's loss of users over time.

To determine what qualifies as a churned user at Robinhood, it's helpful to first follow the money and understand how Robinhood monetizes. One primary way is by trading activity — whether it is through their Robinhood Gold offering or order flow sold to market makers like Citadel. Thus, a cancellation of their Robinhood Gold membership or a long period of no trading activity could constitute churn. The other way Robinhood monetizes is through a user's account balance. By collecting interest on uninvested cash and making stock loans to counterparties, Robinhood is incentivized to have user's manage a large portfolio on the platform. As such, a negligible account balance or portfolio maintained over a period of time — say a quarter — could constitute a churned user.

Churn is a big deal, because even a small monthly churn can compound quickly over time: consider that a 2% monthly churn translates to almost a 27% yearly churn. Since it is much more expensive to acquire new customers than to retain existing ones, businesses with high churn rates will need to continually dedicate more financial resources to support new customer acquisition, which is costly,

and therefore to be avoided if possible. So, if Robinhood is to stay ahead of WeBull, Coinbase, and TD Ameritrade, predicting who will churn, and then helping these at-risk users, is beneficial.

After you've worked with your interviewer to clarify what churn is in this context, and why it's important to mitigate, be sure to ask the obvious question: how is my model output going to be used? If it's not clear how the model will be used for the business, then even if the model has great predictive power, it is not useful in practice.

Step 2: Modeling Considerations

Any classification algorithm could be used to model whether a particular customer would be in the churned or active state. However, models that produce probabilities (e.g., logistic regression) would be preferable if the interest is in the probability of the customer's loss rather than simply a final prediction about whether the customer will be lost or not.

Another key consideration when picking a model in this instance would be model explainability. This is because company representatives likely want to understand the main reasons for churn to support marketing campaigns or customer support programs. In this case, interpretable models such as logistic regression, decision trees, or random forests should be used. However, if by talking with the interviewer you learn that it's okay to simply detect churn, and that explainability isn't required, then less interpretable models like neural networks and SVMs can work.

Step 3: Features We'd Use to Model Churn

Some feature ideas include:

- ***Raw Account Balance:*** Is the portfolio value close to the threshold where it doesn't make sense to check the app anymore (say under $10)?
- ***Account Balance Trend:*** Is there a negative trend — they used to have $20k in their account, but have steadily been withdrawing money out of the account?
- ***Experienced Heavy Losses:*** Maybe they recently lost a ton of money trading Dogecoin, making them want to quit investing and rethink their trading strategies (and their life).
- ***Recent Usage Patterns:*** Maybe they used to be a Daily Active User, but recently have started logging in less and less — a sign that the app isn't as important any more.
- ***User Demographics:*** Standard user profile variables like age, gender, and location can also be used to model churn.

It's also wise to collaborate with business users to see their perspectives and to look for basic heuristics they might use that can be factored into the model. For example, maybe the customer support team has some insights into signals that indicate a user will churn out.

After running the model, it is good to double-check the results to see if the feature importance roughly matches what we would intuitively expect; for example, it is unlikely that a higher balance would result in a higher likelihood of churn.

Step 4: Deploying the Churn Model

We want to make sure the various metrics of interest (confusion matrix, ROC curve, F1 score, etc.) are satisfactory during offline training before deploying the model in production. As with any prediction task, it is important to monitor model performance and adjust features as necessary whenever there is new data or feedback from customer-facing teams. This helps prevent model degradation, which is a common problem in real-world ML systems. We'd also continuously conduct error analysis by

looking at where the model is wrong, in order to keep refining the model. Finally, we'd also make sure to A/B test our model to validate its impact.

Solution #7.31

In matrix form, we assume Y is distributed as multivariate Gaussian: $Y \sim N(X\beta, \sigma^2 I)$

The likelihood of Y given above is $L(\beta,\sigma^2) = (2\sigma^2\pi)^{-\frac{N}{2}} \exp\left(-\frac{1}{2\sigma^2}(X\beta - Y)^T(X\beta - Y)\right)$

Of which we can take the log in order to optimize:

$$\log L(\beta,\sigma^2) = -\frac{N}{2}\log(2\sigma^2\pi) - \frac{1}{2\sigma^2}(X\beta - Y)^T(X\beta - Y)$$

Note that, when taking a derivative with respect to β, the first term is a constant, so we can ignore it, making our optimization problem as follows:

$$\arg\max_{\beta} -\frac{1}{2\sigma^2}(X\beta - Y)^T(X\beta - Y)$$

We can ignore the constant and flip the sign to rewrite as the following: $\arg\min_{\beta}(X\beta - Y)^T(X\beta - Y)$

which is exactly equivalent to minimizing the sum of the squared residuals.

Solution #7.32

PCA aims to reconstruct data into a lower dimensional setting, and so it creates a small number of linear combinations of a vector x (assume it to be p dimensional) to explain the variance within x. More specifically, we want to find the vector w of weights such that we can define the following linear combination:

$$y_i = w_i^T x = \sum_{j=1}^{p} w_{ij} x_j$$

subject to the constraint that w is orthonormal and that the following is true:

$$y_i \text{ is uncorrelated with } y_j,\ var(y_i) \text{ is maximized}$$

Hence, we perform the following procedure, in which we first find $y_1 = w_1^T x$ with maximal variance, meaning that the scores are obtained by orthogonally projecting the data onto the first principal direction, w_1. We then find $y_2 = w_2^T x_1$ is uncorrelated with y_1 and has maximal variance, and we continue this procedure iteratively until ending with the kth dimension such that

$$y_1, \ldots y_k \text{ explain the majority of variance, } k << p$$

To solve, note that we have the following for the variance of each y, utilizing the covariance matrix of x: $var(y_i) = w_i^T\ var(x) w_i = w_i^T\ \Sigma w_i$

Without any constraints, we could choose arbitrary weights to maximize this variance, and hence we will normalize by assuming orthonormality of w, which guarantees the following: $w_i^T w_i = 1$

We now have a constrained maximization problem where we can use Lagrange multipliers. Specifically, we have the function $w_i^T \Sigma w_i - \lambda_i(w_i^T w_i - 1) = 0$, which we differentiate with respect to w to solve the optimization problem:

$$\frac{d}{dw_i} w_i^T \Sigma w_i - \lambda_i(w_i^T w_i - 1) = \Sigma w_i - \lambda_i(w_i) = 0$$

Simplifying, we see that: $\Sigma w_i = \lambda_i(w_i)$

This is the result of an eigen-decomposition, whereby w is the eigenvector of the covariance matrix and λ is this vector's associated eigenvalue. Noting that we want to maximize the variance for each y, we pick: $w_i^T \Sigma w_i = w_i^T \lambda_i w_i = \lambda_i w_i^T w_i = \lambda_i$ to be as large as possible. Hence, we choose the first eigenvalue to be the first principal component, the second largest eigenvalue to be the second principal component, and so on.

Solution #7.33

Logistic regression aims to classify X into one of k classes by calculating the following:

$$\log \frac{P(C=i|X=x)}{P(C=K|X=x)} = \beta_{i0} + \beta_i^T x$$

Therefore, the model is equivalent to the following, where the denominator normalizes the numerator over the k classes:

$$P(C=k|X=x) = \frac{e^{\beta_{k0}+\beta_k^T x}}{\sum_{i=1}^{K} e^{\beta_{i0}+\beta_i^T x}}$$

The log-likelihood over N observations in general is the following:

$$L(\beta|X,C) = \sum_{i=1}^{N} \log P\left(C=k|X=x_i,\beta\right)$$

Use the following notation to denote classes 1 and 2 for the two-class case:

$$y_i = 1 \text{ if the class is 1, otherwise 0}$$

Then we have the following: $P(C = 2 \mid X = x_i, \theta) = 1 - P(C = 1 \mid X = x_i, \theta)$

Using the following notation, $P(C = 1 \mid X = x_i, \theta) = p(x_i)$ such that the log-likelihood can be written as follows:

$$L(\beta) = \sum_{i=1}^{N} \left(y_i \log p(x_i) + (1-y_i)\log\left(1-p(x_i)\right)\right)$$

Simplifying yields the following:

$$L(\beta) = \sum_{i=1}^{N} \log\left(1-p(x_i)\right) + \sum_{i=1}^{N} y_i \log \frac{p(x_i)}{1-p(x_i)}$$

Substituting for the probabilities yields the following:

$$L(\beta) = \sum_{i=1}^{N} \log\left(1-e^{\beta_0+\beta_1 x_i}\right) + \sum_{i=1}^{N} y_i(\beta_0+\beta_1 x_i)$$

To maximize this log-likelihood, take the derivative and set it equal to 0

$$\frac{\partial L(\beta)}{\partial \beta} = \sum_{i=1}^{N} \left(x_i\left(y_i - p(x_i)\right)\right) = 0$$

We note that: $\frac{\partial}{\partial \beta} \sum_{i=1}^{N} \log\left(1-e^{\beta_0+\beta_1 x_i}\right) = \frac{\partial}{\partial \beta} \sum_{i=1}^{N} -\log\left(1+e^{\beta_0+\beta_1 x_i}\right) = \sum_{i=1}^{N} -\frac{e^{\beta_0+\beta_1 x_i}}{1+e^{\beta_0+\beta_1 x_i}}$

which is equivalent to the latter half of the above expression: $-\sum_{i=1}^{N} p(x_i)$

The solutions to these equations are not closed form, however, and, hence, the above should be iterated until convergence.

Solution #7.34

Step 1: Clarify Details of Discover Weekly

First we can ask some clarifying questions:

- What is the goal of the algorithm?
- Do we recommend just songs, or do we also include podcasts?
- Is our goal to recommend new music to a user, and push their musical boundaries? Or is it to just give them the music they'll want to listen to the most, so they spend more time on Spotify? Said more generally, how do we think about the trade-off of exploration versus exploitation?
- What are the various service-level agreements to consider (e.g., does this playlist need to change week to week if the user doesn't listen to it?)
- Do new users get a Discover Weekly playlist?

Step 2: Describe What Data Features You'd Use

The core features will be user-song interactions. This is because users' behaviors and reactions to various songs should be the strongest signal for whether or not they enjoy a song. This approach is similar to the well-known use case for movie recommendations on Netflix, with several notable differences:

- Feedback for music does not have a 1-to-5 rating scale as Netflix does for its movies.
- Music may be subject to repeated consumption (i.e., people may watch a movie once or twice but will listen to a song many times).
- Music has a wider variety (i.e., niche music).
- The scale of music catalog items is much larger than movies (i.e., there are many more songs than movies).

There is also a variety of other features outside of user-song interactions that could be interesting to consider. For example, we have plenty of metadata about the song (the artist, the album, the playlists that include that song) that could be factored in. Additionally, potential audio features in the songs themselves (tempo, speechiness, instruments used) could be used. And finally, demographic information (age, gender, location, etc.) can also impact music listening preferences — people living in the same region are more likely to have similar tastes than someone living on the other side of the globe.

Step 3: Explain Collaborative Filtering Model Setup

There are two types of recommendation systems in general: collaborative filtering (recommending songs that similar users prefer) and content-based recommendation (recommending similar types of songs). Our answer will use collaborative filtering.

Collaborative filtering uses data from feedback users have provided on certain items (in this case, songs) in order to decide recommendations. Therefore, a user-song matrix (or a user-artist matrix) would constitute the dataset at hand, with the rows of the dataset being users and the columns various songs. However, as discussed in the prior section, since explicit song ratings are lacking, we can proxy liking a song by using the number of times a user streamed it. This song play count is stored for every entry in the user-song matrix.

The output of collaborative filtering is a latent user matrix and a song matrix. Using vectors from these matrices, a dot product denotes the relevance of a particular song to a particular user. This process can be used directly for recommendation at the user level, where we sort by relevance scores on songs that the user has not yet streamed. You can also use these vectors to assess similarity between different users and different songs using a method such as *kNN* (*K*-nearest neighbors).

Step 4: Additional Considerations

Also relevant to this discussion are the potential pros and cons of collaborative filtering. For example, one pro is that you can run it in a scalable manner to find correlations behind user-song interactions. On the flip side, one con is the "cold start" problem, where an existing base of data is needed for any given user.

Another important consideration is scale. Since Spotify has hundreds of millions of users, the Discover Weekly algorithm could be updated in batch for various users at different times to help speed up data processing and model training.

Another consideration is the dynamic nature of the problem; the influx of new users and songs, along with fast-changing music trends, would necessitate constant retraining.

Lastly, it is important to consider how you can measure and track the impact of this system over time — collaborative filtering doesn't come with clear metrics of performance. Ideally, you'd use an A/B test to find that users with the improved recommendations had increased engagement on the platform (as measured by time spent listening, for example).

Solution #7.35

We are attempting to solve for $Var(\hat{\beta})$

Recall that the parameter estimates have the following closed-form solution in matrix form:

$$\hat{\beta} = (X^T X)^{-1} X^T y$$

To derive the variance of the estimates, recall that for any given random variable X:

$$Var(X) = E[X^2] - E[X]^2$$

Therefore, we have the following: $Var(\hat{\beta}) = E(\hat{\beta}^2) - E[\hat{\beta}]^2$

We can evaluate the second term since we can assume the parameter estimates are unbiased.

Therefore, $E[\hat{\beta}] = \beta$

$$Var(\hat{\beta}) = E(\hat{\beta}^2) - \beta^2$$

Substituting into the closed-form solution yields the following: $Var(\hat{\beta}) = E[((X^T X)^{-1} X^T y)^2] - \beta^2$

Since least squares assumes that: $y = X\beta + \epsilon$ where $\epsilon \sim N(0, \sigma^2)$, we have the following:

$$Var(\hat{\beta}) = E[((X^T X)^{-1} X^T (X\beta + \epsilon))^2] - \beta^2$$

$$Var(\hat{\beta}) = E[(X^T X)^{-1} X^T (X\beta) + (X^T X)^{-1} X^T \epsilon)^2] - \beta^2$$

Note that: $(X^T X)^{-1} X^T X = 1$

So simplifying yields: $Var(\hat{\beta}) = E[(\beta + (X^T X)^{-1} X^T \epsilon)^2] - \beta^2$

$$Var(\hat{\beta}) = \beta^2 + E[((X^T X)^{-1} X^T \epsilon)^2] - \beta^2$$

where the middle terms were canceled since the expectation of the error term is 0. Canceling out the first and last squared terms and simplifying the middle part yields the following:

$$Var(\hat{\beta}) = E[(X^T X)^{-1} X^T X (X^T X)^{-1} \epsilon^2] = (X^T X)^{-1} E[\epsilon^2] = (X^T X)^{-1} \sigma^2$$