Guidelines

Extracting Data with Database Queries

Follow these guidelines when extracting data with database queries:

Ask yourself the following questions about the data:
- How frequently will new data be available?
- How frequently will data be refreshed from the data source?
- How available and accessible will the data be?

When dealing with access to data:
- Ensure there are permissions in place for databases and tables.
- Prefer to set read-only access on the data.
- Coordinate when data will be accessed with maintenance, backups, and other downtime.

When extracting data:
- Use queries that are the most efficient for whatever task you're trying to perform.
- Use indices where possible for query criteria.

When dealing with large volumes of data:
- Keep in mind that queries involving larger datasets take longer to run and download.
- Limit the number of columns in a query for large datasets.
- Consider breaking up large datasets into multiple subsets, partitioning columns that are indexed.
- Track data that has already been collected so that only incremental data is retrieved.

Consolidating Data from Multiple Sources

Follow these guidelines when consolidating data from multiple sources:
Ask yourself the following questions when combining data.
- What common fields are there between the datasets, and how can you match them?
- Will you combine the datasets into an existing set or into a new set?
- How will you handle joins on missing data?
- How will you handle duplicated data?

Consider how data can be consolidated.

- Identify same or similar data formats that can be combined (e.g., currency/language conversion).
- Identify character set conversions (e.g., EBCDIC, ASCII, ISO Latin-8, and Unicode).
- Identify scaling factors between data points (e.g., weights in pounds vs. tons).

Consider how you'll get data from multiple locations into one location.

- Identify data stored on local file systems, corporate intranet, and cloud providers.

Parsing Data

Follow these guidelines when parsing data:

- Whenever possible, prefer to parse data from a well-structured format, rather than more permissible formats like HTML.
- Consider that binary formats are faster to parse and will take up less storage space, but they are much harder if not impossible for a human to read.
- Consider that binary formats may only be readable by the system or library that originally created it.

Handling Irregular and Unusable Data

Follow these guidelines when handling irregular and unusable data:
Keep in mind that irregular data is data that has inconsistencies between records; for example, varying records (record lengths, improperly ordered columns) or data formats (date formats, numeric representations, character sets).
Correct data with inconsistent columns between records before input. Automated systems can often correctly interpret when columns are omitted at the end of a record, but it becomes more difficult when cells are omitted from the middle of a record.
Consider the different ways you could handle irregular data:

- By utilities or applications within or outside conventional data science tools. These can perform simple transformations like processing strings.
- On the data source itself before it is exported to a data science environment. This can make it easier to correct data since the context of the original data is readily available.
- By manual processes; for example, manually reviewing data sourced from image scans or optical character recognition (OCR).

Consider that, often, the only way to handle unusable data is to recreate it from source data. This is because unusable data, by definition, cannot be transformed by conventional methods into something that can be read or interpreted by a data science workbench, or data which is not suitable for analysis and modeling.

Correcting Data Formats

Follow these guidelines when correcting data formats:

Correct Date Values

Consider that dates come in many formats. With some care and foreknowledge, they can be transformed into the required date format.
Consider that there are many formal and informal date formatting standards.

- ISO 8601 uses YYYY-MM-DD or YYYYYMMDD as the calendar date.

- Most data science tools and programming languages will read this date natively.
- Some platforms may not read ISO 8601-formatted dates. However, they will often understand dates in YYYY-MM-DD and YYYY-MON-DD and times in HH:MM:SS with and without an AM/PM period indicator.

Keep in mind that dates formatted with the year at the end of the string can be ambiguous. In the U.S., the date would be interpreted as month followed by day, whereas European convention is to interpret it as day followed by month.
Consider your knowledge of the source data when deciding how to transform dates.
For easiest processing, normalize dates in all data sources. This will simplify import and interpretation of data by a data science workbench.

Correct Numeric Values

Consider that numeric values can take many forms in data files. This includes both data type (float, integer) and also the representation of these values. Values may be represented by human-readable characters (ASCII `0`–`9` and digit group separators like commas and periods, and occasionally spaces or underscores) or raw binary values.
Keep in mind that some data science tools will ignore embedded digit group separators. It's still important to know the format of the numbers to know if `.` or `,` is the floating point indicator.
Keep in mind that numeric values can have a leading `+` or `-` to indicate sign.
Keep in mind that negative numbers can be surrounded by parentheses.
Keep in mind that binary values are not human readable.

- For example, the value `12,345,678` takes 10 characters in a data file. The equivalent binary representation could be encoded into three "byte" values `0xBC 0x61 0x43`. This uses the same storage space as three letters, in this case an unintuitive value of `¼aC`.

Deduplicating Data

Follow these guidelines when deduplicating data:

- Keep in mind that data that has been duplicated between different files can be difficult to discover and correct after it has been transformed.
- Be systematic when collecting data. Keep track of which dates have been exported to prevent exporting the same data twice.
- Deduplicate data if there is a value that is guaranteed to be unique within a data record. Some examples of this are a unique database record identifier, timestamp, or a combination of these and other indicators to determine uniqueness.
- Deduplicate records by using ad hoc scripts, third-party libraries, or applications.
- Deduplicate data as early as possible in the data pipeline to conserve storage space and redundant processing operations.
- Perform checks to ensure that no duplicate records exist. Often these checks require deep knowledge of the data being collected. This can be done by calculating different aggregates and verifying that the results are reasonable, such as comparing the count of records with the elapsed collection time.

Text Data Transformation Techniques

There are several other techniques for handling text data that you should be aware of. The following list uses a line from Walt Whitman's poetry collection *Leaves of Grass* as an example:

```
'The shadows, gleams, up under the leaves of the old sycamore-trees'
```

Bag-of-words

A bag of words is a list of each individual word in a document without respect to grammar, punctuation, or any other language component that may be extraneous to the word itself. It is usually represented in a collection object like an array or dictionary. In the case of the latter, the key is the word and the value is the number of times it appears in the document.

A bag of words using the example might produce a dictionary like:

```
{'the': 3, 'shadows': 1, 'gleams': 1, 'up': 1, 'under': 1, 'leaves': 1,
'of': 1, 'old': 1, 'sycamore': 1, 'trees': 1}
```

Tokenization

Tokenization partitions a document into smaller units. In fact, *n*-grams are a form of tokenization. There is also word tokenization, where a document is simply broken up into words, and character tokenization, where each character is partitioned.

An example of character tokenization applied to the word "gleams" and put into a list:

```
['g', 'l', 'e', 'a', 'm', 's']
```

Stop words

A stop word is any word that is very common in text and whose inclusion typically provides little to no value. Words like "the," "a," "an," and so on, are usually excised from a document and put into a list of stop words. When an algorithm operates on text, it can refer to this list and ignore any such words. Other than universally common words, a stop word can also be any word that appears very frequently in the document in question. So, a list of stop words might be created by taking the frequency of all words in a document and extracting the top results.

Considering the entire poetry collection rather than just a single line, the stop words from this line might be:

```
['the', 'up', 'under', 'of']
```

The word "leaves" is also very common, so it might be a candidate for a stop word. Though, because of its thematic importance, it may be best to leave it alone.

Stemming

Stemming removes the affix of a word in order to reduce the inflection of that word. In most cases this means a suffix, but stemming can also be applied to prefixes. The result of the operation is the word stem, the base form of the word itself. Suffixes include plurals like "s" and "es"; gerunds ("ing"); past tense ("ed"); adverbs ("ly"); and so on.

If run through a stemming program, the word "shadows" becomes:

```
'shadow'
```

However, stemming is somewhat limited. For the most part, it simply chops off the suffix without regard to the true dictionary root word. For example, stemming the irregular plural "leaves" produces:

```
'leav'
```

Lemmatization

Lemmatization addresses the shortcomings of stemming through morphology; that is, the analysis of word parts and structures and how they change. Lemmatization attempts to derive the *lemma*, or the canonical dictionary form of a word. The lemma can be derived through cutting off a suffix, like in stemming, but also through more intelligent methods, like analyzing word tense and grammatical mood.

When "leaves" is lemmatized, the outcome will be:

```
'leaf'
```

This is the base dictionary form of the word. However, lemmatization isn't perfect, as "leaves" in a different context might be a verb whose lemma is actually "leave"—to depart.

Transforming Data

Follow these guidelines when transforming data:

- Keep in mind that the goal of transforming tabular data is to make data formats uniform between columns and cells in a given column.
- Keep in mind that most transformations can be done with third-party tools. For example, ad hoc scripts in Python, R, or other languages; custom-written software; or even general-purpose worksheets like Excel with custom formulas or VBA macros.
- Include documentation, tests, and versioning for conversion scripts. It is likely that these conversion scripts will be used multiple times for long-running projects.
- Maintain and update conversion scripts as needed. They should account for changes to the format of incoming data.
- Ensure that dates have a common representation. It's common to use ISO 8601, but others are valid as well.
- Determine whether or not currency should have leading or trailing symbols.
- Add a column that indicates a currency code using a standard like ISO 4217.Remove symbols for grouping digits by thousands, millions, etc.
- Consider expanding scientific notation to use a floating point representation.
- Keep in mind that some numbers have a prefix or suffix indicating magnitude (e.g., "K" and "m" can mean thousands).
- Consider that most data science tools assume an ASCII character set. If the content is in a different character representation like EBCDIC or Unicode, the text should be transformed to ASCII. Some tools will automatically transform the characters when given a "from" and "to" character set. If this feature is not available, there are third-party applications or libraries available that can do these character set transformations.

Consider that strings can be surrounded by:

- Single or double straight quotes (`'` and `"`).
- Chevrons/guillemets (« and »).
- Single or double smart quotes (`‘ ’` and `“ ”`) inserted by some word processing programs.

Note: Most data science tools will recognize and process strings surrounded by straight quotes, but you may need to transform guillemets or smart quotes into straight quotes. Smart quotes can be particularly challenging to transform because they are often hard to distinguish from straight quotes.

Loading Data into Databases

Follow these guidelines when loading data into databases:

- Look at data files and the targeted database's schema to determine which database tables and columns the data will be loaded to.
- Decide where every piece of data in your data file will be loaded to.
- Load data with a custom-written program, data science tool, or a bulk-loading program like mysqldump for MySQL or SQL*Loader for Oracle Database.
- Consider writing a small program to load data if it will be loaded regularly.

- Collect and analyze missing data or erroneous data and decide if this will violate any data integrity rules and make data inconsistent.

Loading Data into DataFrames

Follow these guidelines when loading data into `DataFrame`s:

- Use one `DataFrame` for each dataset being loaded.
- Decide which columns are in common between datasets so data from different datasets can be combined.
- Combine `DataFrame`s after loading based on joined columns.
- Keep in mind that most data science tools read all data from a file into memory at once, potentially causing slowness or errors when files are very large.

Loading Data into Text Files

Follow these guidelines when loading data into text files:
Choose the file format that is best for your needs. Common text-based formats are CSV (comma-separated values), JSON (JavaScript Object Notation), and XLS/X (Excel).
Prefer CSV for the widest compatibility with all applications. Nearly every application and data science workbench supports importing and exporting CSV files.
Consider the advantages and disadvantages of CSV files:

- **Advantages**: near-universal compatibility for importing and exporting, excellent library and language support, easy to parse, human readable, easy to edit, and forgiving of some minor errors in formatting.
- **Disadvantages**: not very space efficient, only one dataset can be stored per CSV file, and support for embedded delimiters (quotes, commas, etc.) depends on the data science tool used.

Consider the advantages and disadvantages of JSON files:

- **Advantages**: human readable, easy to edit, good library and language support, many data sources store data in native JSON or can export in this format, structured so one file can source more than one dataset, and tools and libraries can enable querying like a database.
- **Disadvantages**: not space efficient, not as widely supported as CSV, and limited and non-standard support for binary data.

Consider the advantages and disadvantages of XLS/X files:

- **Advantages**: binary data is supported, very space efficient, and some formats automatically incorporate compression.
- **Disadvantages**: not human readable, and varying support for formats.

Store binary data for textual file formats (JSON, CSV) in Base64.
Keep in mind that smaller files are read from and written to faster, but add the complexity of maintaining multiple files.
When writing CSV files, store less than 1,048,576 rows and 16,384 columns, and keep each column to less than 255 characters to maintain compatibility with Excel.
Select the proper options when exporting data if the data includes Unicode/UTF-8 characters.

Configuring ETL Endpoints

Follow these guidelines when configuring ETL endpoints:

- Consider how data will be used when deciding on output formats.
- Find out how to send data to the ETL tool. Some tools support RESTful, textual, and/or proprietary formats and libraries.
- Obtain permissions to use endpoints and tools.
- Write custom applications for repeatability and to help with the automation of regular loads.
- Consider how often data needs to be sent.
- Determine how long a data load will take to complete.
- Work with the ETL provider or team to determine what data transformations will occur before data is sent to the ETL provider and what will be done by the ETL system.
- Keep in mind that working with ETL systems can be challenging because they perform complex transformations between enterprise systems, often requiring a dedicated team to support them.

Loading Data into Visualization Tools

Follow these guidelines when loading data into visualization tools:

- Keep in mind that some visualization tools are their own data science platforms, and may integrate with other data science tools.
- Only load the data to be visualized and any supporting data, as the volume of data can affect rendering or animation times.
- Consider what data needs to be aggregated before visualization.
- Look at the data formats required by the visualization tool and format appropriately.
- Format data as much as possible before sending it to the visualization tool.

Exploring Data Visualization Tools

While most of your work will be done in a programming language, you may find it easy to load the data into a tool that is specifically designed to generate neatly formatted visuals. You can search the web for data visualization tools to see if you can find any that might be beneficial to the project.

Open a new browser tab and navigate to your search engine of choice. Then, search for ***data visualization tools*** or search for a specific tool you're interested in, like Tableau or Microsoft Power BI. Open one or more results that look interesting to you, then briefly read the reviews and/or explanations of the tool(s).

Think about the following questions:

- **What types of features do these tools offer?**
- **How do they compare to other data visualization tools?**

Examining Data

Follow these guidelines when examining data:

- Spend a fair bit of time interacting with the data you will be working with.
- Use charts to begin to understand the data, but don't rely too heavily on them.

- Use multiple correlated charts of different fields to determine correlation between these fields.
- Examine a number of small portions of a dataset to understand relationships. Simply scrolling through the data is one technique to use, as is looking at every *n*th record.
- Resist the urge to start drawing conclusions from the data without a thorough manual review of the data.
- Bring the data into a data science tool or even Excel for ad hoc searching and plotting.

Exploring the Underlying Distribution of Data

Follow these guidelines when exploring the underlying distribution of data:

- Use histogram, scatter, and other plots to get a sense of the shape of the data. Plot lines for statistical measures like mean and median. One or two standard deviations can be included too.
- Look at the distribution of different fields as each may have a different distribution.
- Always cross-check assumptions about distributions by comparing and finding correlation with standardized distributions and calculated values like skew and kurtosis, as your initial perception of plots may not always be accurate.
- Prefer to use standard deviation over variance when explaining the variability of data. This keeps the results in the same scale as the data.
- Ensure you're using measures of central tendency that apply to the feature type you're exploring; e.g., don't try to obtain the mean of a categorical variable.

Analyzing Data Using Histograms

Follow these guidelines when analyzing data using histograms:

- Manually adjust the number of bins using observation and experimentation.
- Plot multiple groups of numerical data on the same chart to identify overlap between those groups.
- Use histograms to reveal multimodality in data.

Analyzing Data Using Box Plots and Violin Plots

Follow these guidelines when analyzing data using box plots and violin plots:

- Use a series of box plots to show many statistical measures (mean, median, min, max, etc.) across a number of datasets.
- Use violin plots to show an extra dimension of value densities that box plots can't show.
- Experiment with different colors for each violin plot.
- Create violin and box plots when there are multiple datasets to display at once, plotting them on the same axes.

Analyzing Data Using Scatter Plots, Line Plots, and Area Plots

Follow these guidelines when analyzing data using scatter plots, line plots, and area plots:

- Use labels for important data points only, as too many labels clutter plots.
- Consider using a color and pattern for each line on a line plot so plots are readable if rendered in black and white.
- Use scatter plots where neither axis is categorical data, one of the axes is a time or date, or both axes have irregular intervals.
- Use a line plot when one axis represents time or date.
- Consider an area plot instead of a line plot when multiple aspects of data need to be shown.
- Plot the largest area first (i.e., at the back) so that the smallest area is not obscured by other plots.

Analyzing Data Using Bar Charts

Follow these guidelines when analyzing data using bar charts:

- Use bar charts when at least one axis is categorical data or discrete data at regular intervals.
- Consider using bar charts when one of the axes has around twenty or fewer values, as too many categories can be difficult to analyze.
- Show multiple data sources on a bar chart if they share common categories.
- Use one color per dataset when displaying multiple datasets.
- Order bars from highest to lowest when there is no inherent ordering of the categories.

Analyzing Data Using Maps

Follow these guidelines when analyzing data using maps:

- Consider overlaying data on a topographical map when the data corresponds to physical features such as water levels, vegetation, etc.
- Avoid using too many colors or too many data overlays in your map.
- Add features to pan and zoom or add/remove data layers if presenting the map in an interactive format.
- Use gradients that make sense for the data being displayed, especially if related to a natural feature or physical phenomena, like blue hues for water, green for vegetation, white and blue for precipitation, etc.
- Add major cities and geographical landmarks to help users orient themselves.
- Use maps that relate to the data being presented, not as a background.

Using Visualizations to Analyze Data

Follow these guidelines when using visualizations to analyze data:

- Be mindful of the story you are trying to tell with your data. Visualizations should augment the story.
- Include axis labels, titles, and data legends.
- Scale axes appropriately with major and minor tick marks, and be sure to label tick marks and format tick mark labels.
- Avoid adding too much data that ends up being distracting, making it difficult to "see the forest for the trees."
- Generate enough visualizations that are necessary for revealing important patterns and ideas, but don't overwhelm your workload with too many visuals.

Handling Missing Values

Follow these guidelines when handling missing values:

- Recognize if the model/algorithm being used is tolerant of some missing data, as many algorithms do not behave well with missing data.
- Replace missing data in continuous values with an average, zero, min, or max depending on the characteristics of a given data series.
- Replace missing category data with the most frequently occurring category.
- Create a new "missing" category if many records are missing a category.
- Consider removing a record if there are a number of variables missing.Drop columns missing values for at least 70% of the records.
- When working with an ordered series, fill in missing records at the start of the series by carrying back the first data available, and fill in missing records at the end of the series by carrying the last data point forward.

Additional Transformation Functions

There are other ways to transform features besides normalization and standardization. Some of these transformation functions are described in the following table.

Transformation Function	**Description**
Log	This function calculates the log base 10 of *x*, log base *e* of *x* (`ln(x)`), or log base 2 of *x*, where *x* is a data example. The log transformation function helps to reduce skewness (particularly positive skewness) in non-normally distributed datasets. Note that it can only be used on positive numbers.
Cube root	This function raises each data example to a power of `1 / 3`.

Box–Cox	This function raises each data example to a power of some lambda (λ) value, subtracts 1, then divides that result by λ. This calculation happens for all values of λ (usually between −5 and 5) until the transformed data approximates a normal distribution. Note that whenever λ is 0, the function simply takes the log of the data example. Box–Cox, therefore, also helps to reduce skewness and obtain normally distributed data. It leverages both log transformations and power transformations. There is also a version that can be used to transform negative values.

Note: These are just a few common examples. Depending on the circumstances, various other functions may be appropriate, such as sigmoid, hyperbolic tangent, reciprocals, Yeo–Johnson, O'Brien, and so forth.

Guidelines for Applying Transformation Functions to Datasets

Follow these guidelines when applying transformation functions to datasets:

- Test different transformations with the model or algorithm used, as different data can behave differently for a given transformation.
- Recognize if transformations are required for the model or algorithm used.
- Use normalization when different variables need to be between a given range.
- Use standardization to preserve distributions when comparing variables with very different scales.

Data Encoding Methods

There are various methods for converting categorical features to numbers. Some of the most common encoding methods are listed in the following table.

Encoding Method	**Description**
Label encoding	Label encoding translates each unique value in a label (the feature you're interested in studying) into separate numbers. So, **`city_encoded`** would have `[0, 1, 2]` as possible values, where `0` is Berlin, `1` is London, and `2` is Rochester. Because machine learning algorithms may perceive an order or ranking to numbered categories, label encoding is most suitable for categories meant to imply a sequence or rank. That's why it's also called ordinal encoding. In this example, a machine learning algorithm might perceive that Rochester (with a value of `2`) is ranked the highest, which may not be what you intended.

One-hot encoding	If you don't want to imply a sequence or rank when you encode categorical labels, you can use one-hot encoding. With this method, you create a dummy column for each class of a categorical attribute. For example, you might create three columns named **`isBerlin`**, **`isLondon`**, and **`isRochester`**. The presence of each class is represented by `1` and its absence is represented by `0`. This ensures that the machine learning algorithm will give no class (Berlin, London, or Rochester, in this case) more value than the others. **Note**: One-hot encoding is sometimes conflated with the very similar technique of dummy encoding. However, dummy encoding creates *n* − 1 columns, whereas one-hot encoding creates *n* columns, where *n* is the number of unique values in the categorical variable.
Binary encoding	This encoding scheme converts each categorical value into an ordinal integer, then converts that into a binary digit. Each binary digit becomes a new column. Some information is lost in this conversion, but it's more efficient than one-hot encoding, because with binary encoding, you don't need one column for each possible value. So, binary encoding is preferred when the number of possible categorical values is high.
Effect encoding	This is similar to binary encoding, except the encoded values are either `−1`, `0`, or `1`. Effect encoding is a more advanced technique and is less common.
Frequency encoding	With this approach you calculate how frequently each class occurs within the training set, and you use this number as the code for that class, essentially determining the weight of that class within the dataset, and using that value as its code.
Target mean encoding	With this encoder, each class is encoded as a function of the mean of the target variable (i.e., the label in supervised learning). So the target variable must be numeric, whereas the variable being encoded is categorical.
Hash encoding	This scheme uses a hash encoding algorithm to map a particular text string to a number value. While the resulting hash value appears to be random, it is algorithmic, based on the characters in the text string, and will produce the same number value each time a particular text string is provided to it. This method can be a useful way to generate consistent codes from text values when there are hundreds or thousands of categories.
Base-*N* encoding	This encoding scheme chooses a number base to convert the categorical values into. Base-1 encoding is the same as one-hot encoding, and base-2 encoding is the same as binary encoding. Otherwise, this particular encoding style confers no additional advantages.

Guidelines for Encoding Data

Follow these guidelines when encoding data:

- Use integers when assigning labels or categories to discrete non-continuous values even when they appear to be numeric (e.g., ZIP codes).
- Convert categories with an inherent, natural ordering to a sequence using ordinals.
- Maintain a lookup table to generate the category names from the values when reporting or plotting.
- Use one-hot encoding when a machine learning algorithm calls for a vector of categories for each record.
- Consider hashing labels when there are a large number (100+) of categories in order to reduce the number of categories, such as ZIP codes or business merchant category codes. Performance and accuracy are often inversely correlated with the number of categories.

Discretizing Variables

Follow these guidelines when discretizing variables:

- Convert continuous variables to discrete ordinals for models or algorithms that cannot handle discrete data or that train faster with a limited number of ordinals for a variable.
- Determine how many bins are required based on the use of the data. Most machine learning algorithms work better with 5 to 10 bins.
- Use a uniform distribution when the shape and other characteristics of the distribution should be preserved.
- Choose a quantile distribution with the same number of samples in each bin when favored by the algorithm being used.
- Recognize that the loss in precision is often offset by the gain in training speed and/or performance of a model.

Splitting Features

Follow these guidelines when splitting features:

- Split on punctuation if available in the field.
- Split on whitespace only when data has regular patterns and each split item does not have embedded spaces. For example, whitespace shouldn't be used to split a data item of city name and state name because many cities and states like "St. Louis" and "New Hampshire" have embedded spaces.
- Use regular expressions when the data format is predictable but doesn't have delimiters.
- Take advantage of case differences, embedded numbers, punctuation, and other features when using regular expressions.
- Retain only the new fields that are required, such as country code and area code when splitting a phone number.
- Process features after splitting by cleaning up or applying transformation functions again as necessary.

Dimensionality Reduction Methods

You can select and extract features manually, but this becomes tedious in high-dimensional datasets. It's also prone to error, as you might not be making optimal decisions about what to reduce and how. Thankfully, there are many ways to automate the dimensionality reduction process. The following table provides an overview of some of the most common dimensionality reduction methods

Dimensionality Reduction Method	**Description**
Principal component analysis (PCA)	PCA performs a type of feature extraction by taking data that is in high dimensions and projecting that data into a space of equal or lower dimensions. It does this by selecting only the features that contribute to the greatest amount of linear variance in the dataset, while dropping the features that contribute very little to the variance.
Singular value decomposition (SVD)	SVD is similar to PCA; both of them decompose matrices of feature values in order to select a subset of features that better represent the data for modeling and analysis. In fact, many implementations of PCA actually use SVD to decompose matrices. SVD tends to work well on sparse matrices, i.e., matrices with mostly zeros.
t-distributed stochastic neighbor embedding (*t*-SNE)	*t*-SNE generates probability distributions of data pairs in high dimensions. Similar data examples are given a higher probability, whereas dissimilar examples are given a lower probability. Then, the process is repeated in lower dimensions until the divergence between higher and lower dimensions is minimized. Unlike PCA, *t*-SNE can retain non-linear variance.
Random forest	Random forests are a type of machine learning algorithm that is commonly used in classification and regression tasks, but they're also effective at feature selection. They can help you determine which features have the largest influence on the model's estimations by returning the weight of each feature as a percentage (out of 100). The higher the percentage, the more important a feature is to the estimator.
Forward feature selection	This method starts a model with no features from the dataset, then iteratively adds features until the performance of the model stops improving.
Backward feature elimination	This method starts a model with all features from the dataset, then iteratively removes features until the performance of the model stops improving.
Factor analysis	Factor analysis measures a latent variable; i.e., a variable that is not directly observed but which is identified through the relationship of other variables. These latent variables are selected as new features because they may exert more influence over a model than the observable variables.

Missing value ratio	This method removes any features that have a total number of missing values that exceed some predefined threshold. Missing values don't improve a model's performance, and, in some cases, they can actually hinder its performance. You can adjust the threshold to either increase or decrease the number of features that get removed.
Low-variance filter	This method removes any features that have a total variance lower than some predefined threshold. Low variance means the feature values don't change much between each data example, which doesn't help a model learn. As with missing value ratio, you can adjust the threshold as needed.
High-correlation filter	This method calculates the correlation coefficient between pairs of features and removes one of those features if the coefficient exceeds a predefined threshold. Correlated features provide similar information to a model, so they may be redundant for the learning process. Once again, you can tune the threshold as desired.

Performing Dimensionality Reduction

Follow these guidelines when performing dimensionality reduction:

- Remove a variable that has a high correlation with another variable.
- Alternatively, join multiple variables that are closely correlated into a single variable with hyphens or spaces.
- Review the impact to results of removed or reduced variables, as the impact is not always predictable.
- Try different methods of dimensionality reduction for a collection of variables.

Preprocessing Data

Follow these guidelines when preprocessing data:

- Automate as much of the process as possible.
- Create ad hoc scripts early in the research to more easily verify with different datasets the design decisions that were made.
- Rerun models and compare with prior executions of the same data to ensure that no code has changed.
- Consider using a single integrated platform for initial investigations before moving different parts to best-of-breed solutions.
- Allocate a generous amount of time when automating a manual process, especially when a number of different tools or data science platforms are being used.
- Consider implementing diagnostic and error reporting for automated processes.
- Recognize that feature engineering decisions will need to be revisited when results are analyzed or new data is received.
- Review data retention policies and identify how long data will need to be stored for verification or modeling purposes.
- Store versions or snapshots of code, programs, workbooks, or workbenches used in the modeling process.

Cross-Validation

Cross-validation (also called rotation estimation or out-of-sample testing) is a technique for partitioning data in order to improve a model's ability to generalize to new data. There are actually several cross-validation techniques, the most basic of which is the holdout method.

Cross-Validation Method	**Description**
k-fold cross-validation	The data is split into *k* groups (folds). One group is the test set, and the remaining groups form the training set. The model trains and then evaluates its performance. Then, the groups rotate: a different group is designated as the test set, and the rest are used in the training set. Once again, the model trains and evaluates its performance. This process repeats *k* times. Then, the average error across all of these trials is calculated. The advantage of this approach is that it minimizes variance, as every data point is used to both train and test at some point. However, because the training and testing must be done *k* times, this method adds overhead to both time and processing power. A practical rule of thumb is to set *k* between 5 and 10.
stratified *k*-fold cross-validation	Similar to *k*-fold cross-validation, this alternative helps to minimize variance and bias issues by ensuring that each train/test fold is a good representation of the data as a whole. In binary classification, if 30% of the data is in class 0 and 70% is in class 1, then 30% of the data in each fold will be in class 0, and 70% will be in class 1. The stratification method is therefore most suitable in cases of class imbalance.
Leave-*p*-out cross-validation (LPOCV)	The *k*-fold method, but with *k* equal to all data points in the set (*n*). So, $n - p$ data points are used in training, and *p* data points are used to test. This is repeated for all possible combinations of data that fit this split. Like with *k*-fold, the average error across these trials is then calculated.
Leave-one-out cross-validation (LOOCV)	The same as LPOCV when *p* is set to 1. It's common to do this because as *p* increases, the amount of trial combinations increases dramatically, leading to significant performance issues. LOOCV is commonly used to minimize bias in smaller datasets, though it may not perform well with larger datasets due to increased variance and performance concerns.

Guidelines for Training Machine Learning Models

Follow these guidelines when training machine learning models:

- **Select the right algorithm for the job**. One machine learning algorithm may be more effective at accomplishing a specific task than another. Or, consider selecting multiple algorithms and comparing the results to choose the right one.

- **Avoid unnecessary complexity**. In general, you should use the simplest or most computationally efficient method that meets your requirements.
- **Prioritize model generalization**. It's important for a model to be able to generalize well to new data. This usually means paying attention to factors like bias and variance and how they can restrict a model's ability to generalize.
- **Learn through experimentation**. Continue to tune your models by modifying factors like the content of datasets used and training and hyperparameters that apply to the relevant algorithms.
- **Select the right hardware**. Make sure the systems you use are appropriate for the tasks you need to accomplish, and they are appropriately configured.
- **Set realistic performance requirements**. You may not need the most skillful model that time and money can produce. Set your performance requirements for the model to align with business requirements. In some cases, you may not need a very high-performing model to get the actionable insights you require.
- **Improve the structure of your datasets**. Some algorithms take much longer to process datasets when data items have high standard deviations or columns are in vastly different ranges. Optimize datasets as needed for the types of tasks you need to perform on them.

Additional Hypothesis Testing Methods

Besides A/B tests, there are several other types of hypothesis tests you can conduct alongside machine learning.

Testing Method	Description
z-test	A *z*-test is used to compare the mean of two distributions when the standard deviation of a population is known. For example, among a population of bank customers, the mean credit score is 700. If you selected a sample of 100 customers from this population, the mean of this sample might be 712. You'd then compare this sample mean to the mean of some other random sampling of 100 customers (e.g., customers in a different market) to see if the increase in score is significant. In other words, the null hypothesis is that the selected 100 customers have comparable credit scores to a random sampling of customers. A *z*-test is performed by calculating the standard deviation of the sample, then using this to calculate the aforementioned *z*-score. The *z*-test is most applicable to larger sample sizes, typically above 30.
t-test	A *t*-test is an alternative to the *z*-test in that it compares the mean of two distributions in which the population standard deviation is *not* known. A *t*-test estimates the population standard deviation by incorporating the standard deviation of the sample. The *t*-test is most applicable to smaller sample sizes, typically below 30.

Analysis of variance (ANOVA)	An ANOVA test compares the mean of multiple distributions. In the standard approach, an ANOVA test evaluates the effect that a single independent variable has on three or more sample groups. For example, among a population of crops, you can test the effect of a specific type of insecticide. The null hypothesis would state that the specific insecticide has the same effect on one random sampling as it does on multiple other random samplings. ANOVA as a single test is more useful than conducting multiple *t*-tests for each sample, as ANOVA considers the variation within and between all of the samples. This minimizes the chance of errors.
Chi-squared test	A chi-squared (χ^2) test compares the effect of categorical variables. For example, you might categorize animals as being either mammals or not mammals. You also categorize animals as having fur or not having fur. A chi-squared test attempts to answer the question, "Does the presence of fur affect whether or not an animal is a mammal?" If yes, then the two categorical variables are said to be dependent. The null hypothesis of such a test is that the variables are independent—i.e., one variable has no significant effect on the other.

Guidelines for Testing a Hypothesis

Follow these guidelines when testing a hypothesis:

- Consider the accuracy of a model compared to a naïve prediction of all one class. As an example, for an illness predictor you can "predict" that a given individual has no illness and be over 99% accurate, as only a small fraction of a population will have any given illness.
- Keep in mind that the null hypothesis is that nothing has changed in a particular problem set, whereas the alternative hypothesis is that there is some meaningful change.
- Choose a hypothesis testing method that is most applicable to your problem domain and dataset.

For A/B tests:

- Focus on testing a hypothesis that is easily measurable.
- Ensure you have a large enough sample size for the test to be effective.
- Define a reasonable duration for conducting the test.
- Ensure your test audience is relevant to the hypothesis being tested.
- Avoid making changes in the middle of the test.

Training Logistic Regression Models

Follow these guidelines when training logistic regression models:

- Consider training a logistic regression model to address binary classification tasks.
- Consider using logistic regression on large datasets.
- Consider using logistic regression when avoiding overfitting is a priority.
- Determine if a classification problem may require a data example being placed into two or more classes (multi-label).
- For multi-label problems, consider training multiple binary classification models, unless the labels are correlated.

- Consider how a classification problem may require a data example being placed into one of three or more possible classes (multi-class).
- For multi-class problems, consider training a multinomial logistic regression model.

Training k-NN Models

Follow these guidelines when training *k*-NN models:

- Consider that lower *k* values make class boundaries more distinct while being less effective at minimizing noise, and vice versa.
- Consider making *k* odd for binary classification to avoid a tie vote.
- Consider using a bootstrapping method of selecting *k* by taking the square root of the number of examples in the dataset.
- Consider using cross-validation to help determine a good *k* value.
- Consider using *k*-NN over logistic regression when simplicity of implementation is key, and when working with smaller datasets.

Training SVM Classification Models

Follow these guidelines when training SVM classification models:

- Consider using an SVM model when the problem you are trying to solve is sensitive to outliers.
- Recognize that the goal of an SVM classification model is to widen the margins as much as feasible, while at the same time keeping data examples outside the margins.
- Tune the regularization hyperparameter to adjust the size of the margins.
- Consider that narrowing the margins too much to keep all examples outside those margins may lead to complications (e.g., overfitting).
- Consider softening the margins to avoid hard-margin overfitting issues.
- Recognize that softening the margins will likely place some examples within those margins, which is often a necessary tradeoff.
- Scale the data used to train an SVM model so the margins are calculated based on the distribution of the feature spaces and not the range of values.

Training Naïve Bayes Models

Follow these guidelines when training naïve Bayes models:

- Use naïve Bayes when variables have minimal or no dependency or relationship to each other.
- Consider using naïve Bayes when the speed of training and prediction generation are important.
- Encode categorical features (observations and outcomes) to integers.
- Consider discretizing variables that have a large number of distinct observations.
- Create a new feature for a combination of features that does not already exist in the dataset. For instance, if the predictor factors in weather and temperature, you probably won't have a combination for **`rainy`** and **`less than 40 degrees`**, so you can create a feature for **`rainy & temp < 40`** with an outcome of **`True`** or **`False`**.

CART Hyperparameters

The following are some of the major hyperparameters associated with CART.

Hyperparameter	Description
`max_depth`	Use this to specify how "deep" the tree should go, i.e., the number of decision nodes from root to farthest leaf, which relates to the number of times the data is split. If you don't specify a max depth, CART will keep splitting the dataset until *G* (purity) for all leaves is 0, or until all decision nodes are less than the value of `min_samples_split`. In their quest for achieving high levels of decision purity, decision trees tend to fall prey to overfitting, especially when the dataset has many features. The `max_depth` parameter is one way to regularize the training in order to avoid overfitting. There is not necessarily one ideal depth value for all scenarios; you must evaluate the model's performance to determine the ideal value for your specific task.
`min_samples_split`	Use this to specify how many samples (i.e., data examples) are required in order to split a decision node. By increasing this value, you constrain the model because each node must contain many examples before it can split. This can help minimize overfitting, but take care not to constrain the model too much, and thus underfit it.
`min_samples_leaf`	Use this to specify how many samples are required to be at a leaf node. Again, increasing this value can reduce overfitting but also has the chance to lead to underfitting.
`splitter`	The default value for this parameter is `"best"`, which calculates the Gini index as described previously. However, if you specify `splitter="random"`, the CART algorithm splits a feature at random and then calculates its Gini index. It repeats this process and identifies the split with the best Gini index. The advantages of this approach are that it requires less computational overhead and may be better at avoiding overfitting in certain cases. However, random splitting will not always be as good at making optimal decisions as the default setting.

Note: The exact names of the parameters may differ based on the library you're using; the names in the previous table are based on scikit-learn.

Training Classification Decision Trees and Ensemble Models

Follow these guidelines when training classification decision trees and ensemble models:

For decision trees:

- Consider using decision trees if you'd like to keep the model simple, easy to understand, and easy to demonstrate visually—especially to non-technical audiences.

- Consider that, depending on the nature of the dataset and the implementation of the decision tree algorithm, you may not need to prepare that data as much as you would with other algorithms.
- Consider that you may need to one-hot encode variables before using them with decision tree algorithms.
- Consider that you may need to discretize continuous variables before using them with decision tree algorithms.
- Be aware that decision trees are prone to overfitting.
- Perform pre-pruning by tuning various decision tree hyperparameters, like the maximum depth of the tree, to help reduce overfitting.
- Consider performing various post-pruning methods such as reduced error pruning to further reduce the complexity of trees and minimize overfitting.

For random forests:

- Consider using an ensemble learning method like random forests to improve your model's estimative skill and reduce overfitting.
- Use a bootstrap aggregation (bagging) technique with random forests to perform data sampling.
- Consider that the bagging process makes cross-validation unnecessary with random forests.
- Use most of the same pre-pruning hyperparameters in a random forest as you would on a single decision tree.
- Consider limiting the number of trees to grow in the forest to around a few hundred, as growing more may not be worth the extra training time.
- Use random forests for feature selection, culling the features that exhibit less importance and thereby reducing the dimensionality of the dataset.

For gradient boosting:

- Consider using gradient boosting methods over random forests if you have unbalanced datasets.
- Be aware of gradient boosting's tendency to overfit data that is noisy.
- Be aware that a gradient boosting model can take much longer to train than a random forest.

Tuning Classification Models

Follow these guidelines when tuning classification models:

- Use a search technique like grid search to find the optimal hyperparameters for a given model.
- Perform the search based on the metric(s) you're trying to optimize.
- Define a grid that includes the hyperparameters you want to try in combination.
- Prefer randomized search to cut down on search time, especially when there is a large field of values that you'd like to include in the search.
- Adjust the number of iterations in a randomized search, considering the tradeoff between search time and the quality of the results.
- Consider that Bayesian optimization can return results even faster than randomized search, but is more difficult to implement.

Evaluating Classification Models

Follow these guidelines when evaluating classification models:
Consider the significance of the four truth values for a classification problem: true positive, true negative, false positive, false negative.

- Use a confusion matrix to visualize these truth results for a classification problem.

For individual metrics:

- Consider that accuracy may not be useful in datasets with a class imbalance.
- Prefer precision and recall when the dataset has a class imbalance.
- Consider that recall may be better than precision when it's crucial that the model avoid false negatives (e.g., in predicting the presence of disease in patients).
- Consider that there is a tradeoff between precision and recall—as one increases, the other tends to decrease.
- Use F_1 score when neither precision nor recall is more important than the other.
- Prefer specificity when you need to maximize the amount of true negatives in a balanced dataset.
- Generate a ROC curve and its AUC to compare the true positive rate (TPR) with the false positive rate (FPR) for all thresholds.
- Generate learning curves to identify whether or not adding more data examples will improve a model's skill.
- Generate learning curves to identify whether a model is high in bias, high in variance, or both.

Overall:

- Consider that there is no objectively "correct" evaluation metric for all problems, or even for a given type of problem.
- Consider applying multiple metrics to the same problem.

Training Linear Regression Models

Follow these guidelines when training linear regression models:

- Use linear regression in supervised learning to predict the change in value of a numeric dependent variable in relation to one or more independent variables.
- Consider how linear regression can lead to more simplified and interpretable model results as compared to other forms of regression.
- Consider that not all data can easily be fit to a straight line.
- Keep in mind the shortcomings of the closed-form normal equation: that large datasets will lead to memory issues due to the computational cost of inverse matrix calculations, and that it is prone to overfitting.

Training Regression Trees and Ensemble Models

Follow these guidelines when training regression trees and ensemble models:

- Consider adding a bagging or other ensemble learner for better results and resistance to overfitting.
- Encode categorical data into numerical values.
- Use a bagged random forest to increase training speed with large datasets.

- Reduce the number of dimensions to increase training speed.

Training Forecasting Models

Follow these guidelines when training forecasting models:

- Consider splitting your data into 60% training and 40% testing data.
- Split time series data into segments instead of selecting values at random, with training data always earlier than test data.
- Reduce training time by using fewer categories or applying dimensionality reduction to the data.
- Consider using differences rather than raw values when forecasting time series data.
- Use multiple regions for testing time series data to evaluate overfitting of the training data.

Regularization Techniques

Regularization simplifies a model by applying the λ constraint hyperparameter. More specifically, this hyperparameter is part of a term that is added to the cost function. This term penalizes the model if its parameter values are too high, and therefore keeps the parameter values small. There are actually several techniques for applying regularization to a model, each of which uses a different regularization term. The three regularization techniques that have found the most success are as follows.

Regularization Technique	**Description**
Ridge regression	Uses a mathematical function called an ℓ_2 norm to implement its regularization term. The ℓ_2 norm is the sum of square coefficients, and the objective is to minimize it. This helps to keep the model parameter weights small, reducing overfitting. Ridge regression is suitable in datasets featuring a large number of features, each having at least some estimative power. This is because ridge regression helps to reduce overfitting without actually removing any of the features entirely.
Lasso regression	Uses the ℓ_1 norm to implement its regularization term. The ℓ_1 norm forces the coefficients of the least relevant features to 0—in other words, removing them from the model. Like with ridge regression, this helps the model avoid overfitting to the training data. As opposed to ridge regression, lasso regression is suitable in datasets that have only a small or moderate number of features that have moderate estimative power. Lasso regression is able to eliminate the rest of the features that have no significant effect on the data, which may lead to better model performance than keeping and reducing them, as in ridge regression.

Elastic net regression	Uses a weighted average of both ridge and lasso regression as part of its regularization term. It is therefore an attempt to leverage the best of both ℓ_1 and ℓ_2 norms. Along with *λ*, elastic net regression also uses the *α* ratio hyperparameter. The *α* value specifies which regression technique exerts more influence over the result, where values closer to 0 favor ridge regression and values closer to 1 favor lasso regression. Like lasso regression, elastic net regression tends to work well when there are only a small to moderate number of features that are actually relevant (though, depending on *α*, it can also equal ridge regression). Elastic net regression is typically preferable, as lasso regression may not perform optimally when the number of features far outnumber the training examples. Likewise, elastic net regression tends to perform better in situations where multiple features exhibit high correlation. Pure ridge regression may still be ideal if removing even a small number of features could impair the model's estimative skill.

In most cases, whichever specific technique you choose, it's a good idea to apply at least some form of regularization while training a linear regression model. However, you should only perform regularization during training; when you evaluate the model's performance after training, you must not use the regularization hyperparameter.
Note: *α* is the Greek letter alpha. In scikit-learn, *α* is used instead of *λ* (lambda) to refer to the regularization strength, and **`l1_ratio`** instead of alpha to refer to the weight of lasso vs. ridge in elastic net regularization.

Collinearity

Ridge regression addresses the issue of *collinearity*, in which two features exhibit a linear relationship—in other words, the features are either exactly or very closely related in terms of how they influence the dependent variable. Collinearity therefore makes it difficult to determine how each feature has an effect on the output independently. *Multicollinearity* refers to the same concept, but can extend to more than two features.

Tuning Regression Models

Follow these guidelines when tuning regression models:

- Use lower-order polynomials for linear regression to avoid overfitting the training data.
- Increase leaf size or tree size in tree-based regression models if overfitting is occurring.
- Tune the number of trees in a forest as well as leaf size, measuring overfitting with each change.
- Keep in mind that increasing leaf size for a decision tree can lead to diminished skill.
- Apply grid and/or randomized searches to evaluate a model across a range of hyperparameter values.

Evaluating Regression Models

Follow these guidelines when evaluating regression models:

- Consider that regression models are usually not evaluated based on how well they fit, but on how poorly they fit (the cost).
- Prefer using a cost function like mean squared error (MSE) over R^2 when evaluating the skill of a regression model.
- Prefer using MSE over mean absolute error (MAE).

- Consider using *root* MSE to put the results on the same scale as the data, making them more interpretable.
- Consider using *relative* squared error/absolute error to compare models that estimate values on different scales.

Training k-Means Clustering Models

Follow these guidelines when training *k*-means clustering models:

- Use *k*-means clustering for unsupervised learning to find groups of data points that share similarities. Examples: customer segmentation, categorizing images and video, categorizing alert data, etc.
- Apply *k*-means clustering to data that is spherical or overlapping in nature.
- Consider applying domain knowledge in determining *k*—you may know exactly or approximately how many clusters you need or is best for the problem.

Training Hierarchical Clustering Models

Follow these guidelines when training hierarchical clustering models:

- Like k-means clustering, use hierarchical clustering to find groups of data points that share similarities.
- Use hierarchical clustering on data that is well separated in shape, and does not overlap.
- Consider using hierarchical agglomerative clustering (HAC) over hierarchical divisive clustering (HDC) if simplicity is desired.
- Consider using HDC over HAC to maximize clustering skill.

Guidelines for Tuning Clustering Models

Follow these guidelines when tuning clustering models:

- Encode categorical data to integers before training a clustering model.
- Try different numbers of clusters, avoiding sizes that generate clusters with too many or too few data points.
- Plot values in different colors based on each point's cluster to visually determine the usefulness of clustering.
- Consider that centroids that are close to each other indicate the data may not be suitable for clustering.
- Evaluate a dataset with multiple clustering models to choose the optimal one.
- Use features with the lowest positive/negative correlation (closer to zero) for best clustering performance.

Evaluating Clustering Models

Follow these guidelines when evaluating clustering models:

For *k*-means clustering and latent class analysis (LCA):

- When domain knowledge isn't enough, use statistical analysis methods to help determine *k*.
- Use an elbow point graph to identify the value of *k* when the mean distance between a point and its centroid is no longer decreasing significantly.
- Use within- and between-cluster sum of squares (WCSS and BCSS) to evaluate the compactness of a cluster and how well separated it is from other clusters, respectively.

- Use a silhouette analysis to evaluate both compactness and separation at once.
- In silhouette analysis, consider choosing the *k* with the highest silhouette coefficient.

For hierarchical clustering:

- Use some of the same techniques as in *k*-means clustering to analyze clusters and determine the optimal number.
- Consider applying domain knowledge in determining the number of clusters to stop at.
- Create a dendrogram to analyze how data points are being clustered.

Overall:

- Consider that there is no objectively "correct" number of clusters for many unsupervised problems.
- Consider that different analysis methods may give you different results for an optimal number of clusters, and that one is not necessarily more "correct" than another.

Communicating Results to Stakeholders

Follow these guidelines when communicating results to stakeholders:

- **Don't drown your audience in a sea of numbers**. Adapt the presentation for different audiences you must address, including only information relevant to that audience, avoiding jargon, and explaining things in a way the audience will understand.
- **Use visuals strategically**. Use charts or diagrams when they more clearly illustrate patterns in the data than words or numbers, particularly where they support important points you have to make about your findings. Experiment with various chart types and formatting options as needed until you come up with a configuration that clearly communicates the point you're trying to make with little or no explanation.
- **Use formatting, labels, and legends**. When displaying numbers, use commas and decimal points to help viewers read the numbers. If values are really close in size, use more decimal places to show differences in values. Use currency symbols for money values. Include labels, legends, and other captions where they will help the reader understand what they're looking at. Align columns of numbers on the right, or at the decimal point so they are more easily compared.
- **Avoid suspense**. A good presentation flows well and tells a story, but not in the way a mystery novel tells a story. You will likely provide your audience with a lot of information to process, and they will have questions and concerns. Help them understand and buy in to your conclusions by anticipating the types of questions you think they will have, and sharing important points early on.
- **Provide context for details**. If you must tunnel down into details showing numbers, tables, and charts, be sure to connect them back to the main idea or point you're trying to communicate.
- **Be honest and transparent**. Be clear regarding how you obtained results. Don't hide significant data or results—even if they don't fully support your hypothesis, objectives, or proposed solution.
- **Identify key assumptions you have made**. This will provide context and a rationale for the results you've produced.
- **Check your work**. Although your machine learning models may be impeccable, the quality of your findings may be called into question if your presentations contain errors.
- **Invite feedback**. Make sure your audience has a chance to ask questions so you can provide clarification and promote buy-in.

- **Provide supplementary communication channels for those with different communication styles**. Some people may be hesitant to ask questions during a presentation, and may be more comfortable asking you follow-up questions one on one or through online messaging. Consider inviting them to share feedback through such side channels.
- **Provide supplementary documentation for those with different informational needs**. Some people may require more details or technical depth than others. While it may not be appropriate to bog down your presentation slides or report with details that are only of interest to a small portion of your audience, consider directing those who require more information to view supplemental documents that you've posted on the company intranet, a network directory, or some other secure location where they can view the additional details.
- **Test it before you deliver it**. Depending on the importance of the outcome of your presentation, consider testing it before you deliver it. Pass it by reviewers who can represent your target audience and will give you honest feedback. Ask them questions to assess how well the presentation gets important points across. Do a dry run of your delivery. Make revisions as needed.

Demonstrating Models in a Web App

Follow these guidelines when demonstrating models in a web app:

- Start small and simple, using an iterative approach to refine a web app into a more complex solution.
- Be aware that web apps can be complicated to create and maintain, especially those that have a mobile component.
- Consider engaging an IT or development organization to assist with web app design for an application projected to have a longer lifespan.
- Use a built-in utility or application to display results, conclusions, and charts, if provided with the data science tool.
- Consider prototyping a solution with a small group before it is enabled for everyone.
- Remember to tell a story with the data.
- Focus more on the information being presented than fancy features. Sometimes an interactive Jupyter Notebook will be sufficient for initial presentations.
- Consider where a web app will be deployed based on the availability required.
- Be aware of security issues for your web app. Consider any proprietary information that may be revealed, and the security of your web app platform and system. These are especially important if your web app is being exposed to the public.

Implementing and Testing Production Pipelines

Follow these guidelines when implementing and testing production pipelines:

- **Create software unit tests that validate the model every time new code is pushed out to the application**. Changes in code or data may adversely affect the model. Modular software design with unit tests that address performance of the model can help to mitigate this problem.
- **Protect the model from upstream changes to the data pipeline**. If possible, work with outside data providers to ensure that they will not make changes that will adversely affect

your model's data pipeline. Avoid depending on inputs that are likely to change over time. Provide a configuration layer at the front end of the data pipeline so you can easily adapt to changes in the input data over time.

- **Protect downstream consumers of the model**. When you are unaware of other processes that consume outputs produced by the model, changes to your model (such as retraining or changes to inputs and output) may adversely affect them. By implementing access controls, you will be aware of all consumers and will be able to provide them with proper notifications when changes are anticipated. Require that outside components request access to use the model and/or its outputs so you can track consumer processes.
- **Design for periodic retraining of the model**. Model performance typically decays over time as inputs evolve. It is generally a good idea to periodically retrain a model with recent data.
- **Provide a versioning system for various components used in the application**. Consider providing version numbers for components that may change over time, including model configuration and parameters, specific features used in the data pipeline, and training and validation datasets.
- **Put appropriate protections in place for the data pipeline and all outputs**. The organization must ensure that data is protected for security (the company's confidential data, for example) and privacy (user data protected by laws and regulations).