Google Data Engineering Cheatsheet

Compiled by Maverick Lin (http://mavericklin.com)

Last Updated August 4, 2018

What is Data Engineering?

Data engineering enables data-driven decision making by collecting, transforming, and visualizing data. A data engineer designs, builds, maintains, and troubleshoots data processing systems with a particular emphasis on the security, reliability, fault-tolerance, scalability, fidelity, and efficiency of such systems.

A data engineer also analyzes data to gain insight into business outcomes, builds statistical models to support decision-making, and creates machine learning models to automate and simplify key business processes.

Key Points

- Build/maintain data structures and databases
- Design data processing systems
- Analyze data and enable machine learning
- Design for reliability
- Visualize data and advocate policy
- Model business processes for analysis
- Design for security and compliance

Google Compute Platform (GCP)

GCP is a collection of Google computing resources, which are offered via *services*. Data engineering services include Compute, Storage, Big Data, and Machine Learning.

The 4 ways to interact with GCP include the console, command-line-interface (CLI), API, and mobile app.

The GCP resource hierarchy is organized as follows. All resources (VMs, storage buckets, etc) are organized into **projects**. These projects *may* be organized into **folders**, which can contain other folders. All folders and projects; can be brought together under an organization node. Project folders and organization nodes are where policies can be defined. Policies are inherited downstream and dictate who can access what resources. Every resource must belong to a project and every must have a billing account associated with it.

Advantages: Performance (fast solutions), Pricing (sub-hour billing, sustained use discounts, custom machine types), PaaS Solutions, Robust Infrastructure

Hadoop Overview

Hadoop

Data can no longer fit in memory on one machine (monolithic), so a new way of computing was devised using many computers to process the data (distributed). Such a group is called a cluster, which makes up server farms. All of these servers have to be coordinated in the following ways: partition data, coordinate computing tasks, handle fault tolerance/recovery, and allocate capacity to process.

Hadoop is an open source *distributed* processing framework that manages data processing and storage for big data applications running in clustered systems. It is comprised of 3 main components:

- **Hadoop Distributed File System (HDFS)**: a distributed file system that provides high-throughput access to application data by partitioning data across many machines
- **YARN**: framework for job scheduling and cluster resource management (task coordination)
- **MapReduce**: YARN-based system for parallel processing of large data sets on multiple machines

HDFS

Each disk on a different machine in a cluster is comprised of 1 master node; the rest are data nodes. The **master node** manages the overall file system by storing the directory structure and metadata of the files. The **data nodes** physically store the data. Large files are broken up/distributed across multiple machines, which are replicated across 3 machines to provide fault tolerance.

MapReduce

Parallel programming paradigm which allows for processing of huge amounts of data by running processes on multiple machines. Defining a MapReduce job requires two stages: map and reduce.

- **Map**: operation to be performed in parallel on small portions of the dataset. the output is a key-value pair $< K, V >$
- **Reduce**: operation to combine the results of Map

YARN- Yet Another Resource Negotiator

Coordinates tasks running on the cluster and assigns new nodes in case of failure. Comprised of 2 subcomponents: the resource manager and the node manager. The **resource manager** runs on a single master node and schedules tasks across nodes. The **node manager** runs on all other nodes and manages tasks on the individual node.

Hadoop Ecosystem

An entire ecosystem of tools have emerged around Hadoop, which are based on interacting with HDFS.

Hive: data warehouse software built o top of Hadoop that facilitates reading, writing, and managing large datasets residing in distributed storage using SQL-like queries (HiveQL). Hive abstracts away underlying MapReduce jobs and returns HDFS in the form of tables (not HDFS).
Pig: high level scripting language (Pig Latin) that enables writing complex data transformations. It pulls unstructured/incomplete data from sources, cleans it, and places it in a database/data warehouses. Pig performs ETL into data warehouse while Hive queries from data warehouse to perform analysis (GCP: DataFlow).
Spark: framework for writing fast, distributed programs for data processing and analysis. Spark solves similar problems as Hadoop MapReduce but with a fast in-memory approach. It is an unified engine that supports SQL queries, streaming data, machine learning and graph processing. Can operate separately from Hadoop but integrates well with Hadoop. Data is processed using Resilient Distributed Datasets (RDDs), which are immutable, lazily evaluated, and tracks lineage. ;
Hbase: non-relational, NoSQL, column-oriented database management system that runs on top of HDFS. Well suited for sparse data sets (GCP: BigTable)
Flink/Kafka: stream processing framework. Batch streaming is for bounded, finite datasets, with periodic updates, and delayed processing. Stream processing is for unbounded datasets, with continuous updates, and immediate processing. Stream data and stream processing must be decoupled via a message queue. Can group streaming data (windows) using tumbling (non-overlapping time), sliding (overlapping time), or session (session gap) windows.
Beam: programming model to define and execute data processing pipelines, including ETL, batch and stream (continuous) processing. After building the pipeline, it is executed by one of Beam's distributed processing backends (Apache Apex, Apache Flink, Apache Spark, and Google Cloud Dataflow). Modeled as a Directed Acyclic Graph (DAG).
Oozie: workflow scheduler system to manage Hadoop jobs
Sqoop: transferring framework to transfer large amounts of data into HDFS from relational databases (MySQL)

IAM

Identity Access Management (IAM)

Access management service to manage different members of the platform- who has what access for which resource.

Each member has roles and permissions to allow them access to perform their duties on the platform. 3 member types: Google account (single person, gmail account), service account (non-person, application), and Google Group (multiple people). Roles are a set of specific permissions for members. Cannot assign permissions to user directly, must grant roles.

If you grant a member access on a higher hierarchy level, that member will have access to all levels below that hierarchy level as well. You cannot be restricted a lower level. The policy is a union of assigned and inherited policies.

Primitive Roles: Owner (full access to resources, manage roles), Editor (edit access to resources, change or add), Viewer (read access to resources)
Predefined Roles: finer-grained access control than primitive roles, predefined by Google Cloud
Custom Roles

Best Practice: use predefined roles when they exist (over primitive). Follow the principle of least privileged favors.

Stackdriver

GCP's monitoring, logging, and diagnostics solution. Provides insights to health, performance, and availability of applications.
Main Functions

- **Debugger**: inspect state of app in real time without stopping/slowing down e.g. code behavior
- **Error Reporting**: counts, analyzes, aggregates crashes in cloud services
- **Monitoring**: overview of performance, uptime and heath of cloud services (metrics, events, metadata)
- **Alerting**: create policies to notify you when health and uptime check results exceed a certain limit
- **Tracing**: tracks how requests propagate through applications/receive near real-time performance results, latency reports of VMs
- **Logging**: store, search, monitor and analyze log data and events from GCP

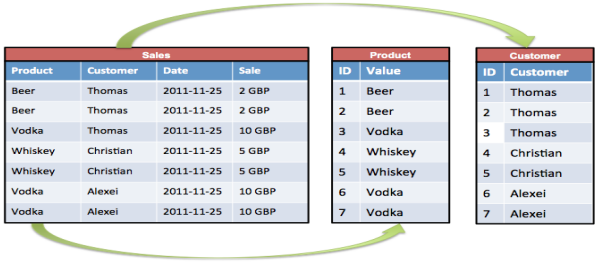
Key Concepts

OLAP vs. OLTP

Online Analytical Processing (OLAP): primary objective is data analysis. It is an online analysis and data retrieving process, characterized by a large volume of data and complex queries, uses data warehouses.
Online Transaction Processing (OLTP): primary objective is data processing, manages database modification, characterized by large numbers of short online transactions, simple queries, and traditional DBMS.

Row vs. Columnar Database

Row Format: stores data by row
Column Format: stores data tables by column rather than by row, which is suitable for analytical query processing and data warehouses



Sales

Product	Customer	Date	Sale
Beer	Thomas	2011-11-25	2 GBP
Beer	Thomas	2011-11-25	2 GBP
Vodka	Thomas	2011-11-25	10 GBP
Whiskey	Christian	2011-11-25	5 GBP
Whiskey	Christian	2011-11-25	5 GBP
Vodka	Alexei	2011-11-25	10 GBP
Vodka	Alexei	2011-11-25	10 GBP

Product

ID	Value
1	Beer
2	Beer
3	Vodka
4	Whiskey
5	Whiskey
6	Vodka
7	Vodka

Customer

ID	Customer
1	Thomas
2	Thomas
3	Thomas
4	Christian
5	Christian
6	Alexei
7	Alexei

IaaS, Paas, SaaS

IaaS: gives you the infrastructure pieces (VMs) but you have to maintain/join together the different infrastructure pieces for your application to work. Most flexible option.
PaaS: gives you all the infrastructure pieces already joined so you just have to deploy source code on the platform for your application to work. PaaS solutions are managed services/no-ops (highly available/reliable) and serverless/autoscaling (elastic). Less flexible than IaaS

Fully Managed, Hotspotting

Compute Choices

Google App Engine

Flexible, serverless platform for building highly available applications. Ideal when you want to focus on writing and developing code and do not want to manage servers, clusters, or infrastructures.
Use Cases: web sites, mobile app and gaming backends, RESTful APIs, IoT apps.

Google Kubernetes (Container) Engine

Logical infrastructure powered by Kubernetes, an open-source container orchestration system. Ideal for managing containers in production, increase velocity and operatability, and don't have OS dependencies.
Use Cases: containerized workloads, cloud-native distributed systems, hybrid applications.

Google Compute Engine (IaaS)

Virtual Machines (VMs) running in Google's global data center. Ideal for when you need complete control over your infrastructure and direct access to high-performance hardward or need OS-level changes.
Use Cases: any workload requiring a specific OS or OS configuration, currently deployed and on-premises software that you want to run in the cloud.

Summary: AppEngine is the PaaS option- serverless and ops free. ComputeEngine is the IaaS option- fully controllable down to OS level. Kubernetes Engine is in the middle- clusters of machines running Kuberenetes and hosting containers.

Additional Notes

You can also mix and match multiple compute options.
Preemptible Instances: instances that run at a much lower price but may be terminated at any time, self-terminate after 24 hours. ideal for interruptible workloads
Snapshots: used for backups of disks
Images: VM OS (Ubuntu, CentOS)

Storage

Persistent Disk Fully-managed block storage (SSDs) that is suitable for VMs/containers. Good for snapshots of data backups/sharing read-only data across VMs.

Cloud Storage Infinitely scalable, fully-managed and highly reliable object/blob storage. Good for data blobs: images, pictures, videos. Cannot query by content.

To use Cloud Storage, you create buckets to store data and the location can be specified. Bucket names are globally unique. There a 4 storage classes:

- **Multi-Regional**: frequent access from anywhere in the world. Use for "hot data"
- **Regional**: high local performance for region
- **Nearline**: storage for data accessed less than once a month (archival)
- **Coldline**: less than once a year (archival)

CloudSQL and Cloud Spanner- Relational DBs

Cloud SQL

Fully-managed relational database service (supports MySQL/PostgreSQL). Use for relational data: tables, rows and columns, and super structured data. SQL compatible and can update fields. Not scalable (small storage- GBs). Good for web frameworks and OLTP workloads (not OLAP). Can use **Cloud Storage Transfer Service** or **Transfer Appliance** to data into Cloud Storage (from AWS, local, another bucket). Use gsutil if copying files over from on-premise.

Cloud Spanner

Google-proprietary offering, more advanced than Cloud SQL. Mission-critical, relational database. Supports horizontal scaling. Combines benefits of of relational and non-relational databases.

- **Ideal**: relational, structured, and semi-structured data that requires high availability, strong consistency, and transactional reads and writes
- **Avoid**: data is not relational or structured, want an open source RDBMS, strong consistency and high availability is unnecessary

Cloud Spanner Data Model

A database can contain 1+ tables. Tables look like relational database tables. Data is strongly typed: must define a schema for each database and that schema must specify the data types of each column of each table.
Parent-Child Relationships: Can optionally define relationships between tables to physically co-locate their rows for efficient retrieval (data locality: physically storing 1+ rows of a table with a row from another table.

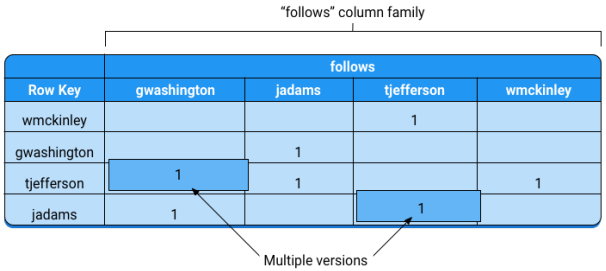
BigTable

Columnar database ideal applications that need high throughput, low latencies, and scalability (IoT, user analytics, time-series data, graph data) for non-structured key/value data (each value is < 10 MB). A single value in each row is indexed and this value is known as the row key. Does not support SQL queries. Zonal service. Not good for data less than 1TB of data or items greater than 10MB. Ideal at handling large amounts of data (TB/PB) for long periods of time.

Data Model

4-Dimensional: row key, column family (table name), column name, timestamp.

- Row key uniquely identifies an entity and columns contain individual values for each row.
- Similar columns are grouped into column families.
- Each column is identified by a combination of the column family and a column qualifier, which is a unique name within the column family.



Load Balancing Automatically manages splitting, merging, and rebalancing. The master process balances workload/data volume within clusters. The master splits busier/larger tablets in half and merges less-accessed/smaller tablets together, redistributing them between nodes.
Best write performance can be achieved by using row keys that do not follow a predictable order and grouping related rows so they are adjacent to one another, which results in more efficient multiple row reads at the same time.

Security Security can be managed at the project and instance level. Does not support table-level, row-level, column-level, or cell-level security restrictions.

Other Storage Options

- Need SQL for OLTP: CloudSQL/Cloud Spanner.
- Need interactive querying for OLAP: BigQuery.
- Need to store blobs larger than 10 MB: Cloud Storage.
- Need to store structured objects in a document database, with ACID and SQL-like queries: Cloud Datastore.

BigTable Part II

Designing Your Schema

Designing a Bigtable schema is different than designing a schema for a RDMS. Important considerations:

- Each table has only one index, the row key (4 KB)
- Rows are **sorted lexicographically by row key**.
- All operations are **atomic** (ACID) at row level.
- Both reads and writes should be distributed evenly
- Try to keep all info for an entity in a single row
- Related entities should be stored in adjacent rows
- Try to store **10 MB in a single cell** (max 100 MB) and **100 MB in a single row** (256 MB)
- Supports max of 1,000 tables in each instance.
- Choose row keys that don't follow predictable order
- Can use up to around 100 column families
- Column Qualifiers: can create as many as you need in each row, but should avoid splitting data across more column qualifiers than necessary (16 KB)
- **Tables are sparse**. Empty columns don't take up any space. Can create large number of columns, even if most columns are empty in most rows.
- Field promotion (shift a column as part of the row key) and salting (remainder of division of hash of timestamp plus row key) are ways to help design row keys.

For time-series data, use tall/narrow tables.
Denormalize- prefer multiple tall and narrow tables

BigQuery

Scalable, fully-managed Data Warehouse with extremely fast SQL queries. Allows querying for massive volumes of data at fast speeds. Good for OLAP workloads (petabyte-scale), Big Data exploration and processing, and reporting via Business Intelligence (BI) tools. Supports SQL querying for non-relational data. Relatively cheap to store, but costly for querying/processing. Good for analyzing historical data.

Data Model

Data tables are organized into units called datasets, which are sets of tables and views. A table must belong to dataset and a datatset must belong to a porject. Tables contain records with rows and columns (fields).
You can load data into BigQuery via two options: batch loading (free) and streaming (costly).

Security

BigQuery uses IAM to manage access to resources. The three types of resources in BigQuery are organizations, projects, and datasets. Security can be applied at the project and dataset level, but not table or view level.

Views

A view is a virtual table defined by a SQL query. When you create a view, you query it in the same way you query a table. **Authorized views allow you to share query results with particular users/groups without giving them access to underlying data.** When a user queries the view, the query results contain data only from the tables and fields specified in the query that defines the view.

Billing

Billing is based on **storage** (amount of data stored), **querying** (amount of data/number of bytes processed by query), and **streaming inserts**. Storage options are active and long-term (modified or not past 90 days). Query options are on-demand and flat-rate.
Query costs are based on how much data you read/process, so if you only read a section of a table (partition), your costs will be reduced. Any charges occurred are billed to the attached billing account. Exporting/importing/copying data is free.

BigQuery Part II

Partitioned tables

Special tables that are divided into partitions based on a column or partition key. Data is stored on different directories and specific queries will only run on slices of data, which improves query performance and reduces costs. Note that the partitions will not be of the same size. BigQuery automatically does this.
Each partitioned table can have up to 2,500 partitions (2500 days or a few years). The daily limit is 2,000 partition updates per table, per day. The rate limit: 50 partition updates every 10 seconds.

Two types of partitioned tables:

- **Ingestion Time**: Tables partitioned based on the data's ingestion (load) date or arrival date. Each partitioned table will have pseudocolumn _PARTITIONTIME, or time data was loaded into table. Pseudocolumns are reserved for the table and cannot be used by the user.
- **Partitioned Tables**: Tables that are partitioned based on a TIMESTAMP or DATE column.

Windowing: window functions increase the efficiency and reduce the complexity of queries that analyze partitions (windows) of a dataset by providing complex operations without the need for many intermediate calculations. They reduce the need for intermediate tables to store temporary data

Bucketing

Like partitioning, but each split/partition should be the same size and is based on the hash function of a column. Each bucket is a separate file, which makes for more efficient sampling and joining data.

Querying

After loading data into BigQuery, you can query using Standard SQL (preferred) or Legacy SQL (old). Query jobs are actions executed asynchronously to load, export, query, or copy data. Results can be saved to permanent (store) or temporary (cache) tables. 2 types of queries:

- **Interactive**: query is executed immediately, counts toward daily/concurrent usage (default)
- **Batch**: batches of queries are queued and the query starts when idle resources are available, only counts for daily and switches to interactive if idle for 24 hours

Wildcard Tables

Used if you want to union all similar tables with similar names. '*' (e.g. project.dataset.Table*)

Optimizing BigQuery

Controlling Costs

- Avoid SELECT * (full scan), select only columns needed (SELECT * EXCEPT)
- Sample data using preview options for free
- Preview queries to estimate costs (dryrun)
- Use max bytes billed to limit query costs
- Don't use LIMIT clause to limit costs (still full scan)
- Monitor costs using dashboards and audit logs
- Partition data by date
- Break query results into stages
- Use default table expiration to delete unneeded data
- Use streaming inserts wisely
- Set hard limit on bytes (members) processed per day

Query Performance

Generally, queries that do less work perform better.

Input Data/Data Sources

- Avoid SELECT *
- Prune partitioned queries (for time-partitioned table, use PARTITIONTIME pseudo column to filter partitions)
- Denormalize data (use nested and repeated fields)
- Use external data sources appropriately
- Avoid excessive wildcard tables

SQL Anti-Patterns

- Avoid self-joins., use window functions (perform calculations across many table rows related to current row)
- Partition/Skew: avoid unequally sized partitions, or when a value occurs more often than any other value
- Cross-Join: avoid joins that generate more outputs than inputs (pre-aggregate data or use window function)
Update/Insert Single Row/Column: avoid point-specific DML, instead batch updates and inserts

Managing Query Outputs

- Avoid repeated joins and using the same subqueries
- Writing large sets has performance/cost impacts. Use filters or LIMIT clause. 128MB limit for cached results
- Use LIMIT clause for large sorts (Resources Exceeded)

Optimizing Query Computation

- Avoid repeatedly transforming data via SQL queries
- Avoid JavaScript user-defined functions
- Use approximate aggregation functions (approx count)
- Order query operations to maximize performance. Use ORDER BY only in outermost query, push complex operations to end of the query.
- For queries that join data from multiple tables, optimize join patterns. Start with the largest table.

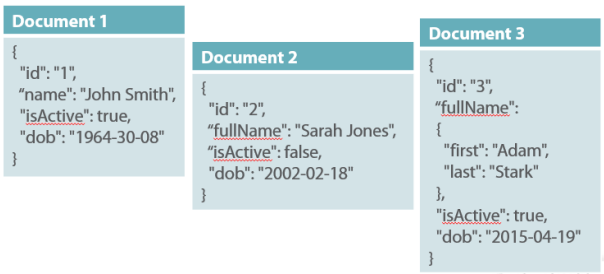
DataStore

NoSQL document database that automatically handles sharding and replication (highly available, scalable and durable). Supports ACID transactions, SQL-like queries. Query execution depends on size of returned result, not size of dataset. Ideal for "needle in a haystack"operation and applications that rely on highly available structured data at scale

Data Model

Data objects in Datastore are known as entities. An entity has one or more named properties, each of which can have one or more values. Each entity in has a key that uniquely identifies it. You can fetch an individual entity using the entity's key, or query one or more entities based on the entities' keys or property values.

Ideal for highly structured data at scale: product catalogs, customer experience based on users past activities/preferences, game states. Don't use if you need extremely low latency or analytics (complex joins, etc).



DataProc

Fully-managed cloud service for running Spark and Hadoop clusters. Provides access to Hadoop cluster on GCP and Hadoop-ecosystem tools (Pig, Hive, and Spark). Can be used to implement ETL warehouse solution.

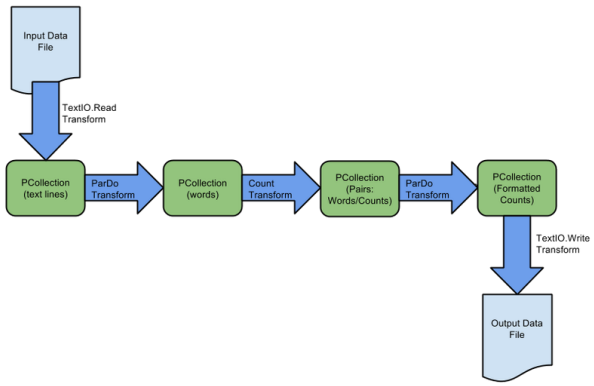
Preferred if migrating existing on-premise Hadoop or Spark infrastructure to GCP without redevelopment effort. Dataflow is preferred for a new development.

DataFlow

Managed service for developing and executing data processing patterns (ETL) (based on Apache Beam) for **streaming** and **batch** data. Preferred for new Hadoop or Spark infrastructure development. Usually site between front-end and back-end storage solutions.

Concepts

Pipeline: encapsulates series of computations that accepts input data from external sources, transforms data to provide some useful intelligence, and produce output
PCollections: abstraction that represents a potentially distributed, multi-element data set, that acts as the pipeline's data. PCollection objects represent input, intermediate, and output data. The edges of the pipeline.
Transforms: operations in pipeline. A transform takes a PCollection(s) as input, performs an operation that you specify on each element in that collection, and produces a new output PCollection. Uses the "what/where/when/how"model. Nodes in the pipeline. Composite transforms are multiple transforms: combining, mapping, shuffling, reducing, or statistical analysis.
Pipeline I/O: the source/sink, where the data flows in and out. Supports read and write transforms for a number of common data storage types, as well as custom.



Windowing

Windowing a PCollection divides the elements into windows based on the associated event time for each element. Especially useful for PCollections with unbounded size, since it allows operating on a sub-group (mini-batches).

Triggers

Allows specifying a trigger to control when (in processing time) results for the given window can be produced. If unspecified, the default behavior is to trigger first when the watermark passes the end of the window, and then trigger again every time there is late arriving data.

Pub/Sub

Asynchronous messaging service that decouples senders and receivers. Allows for secure and highly available communication between independently written applications. A publisher app creates and sends messages to a *topic*. Subscriber applications create a subscription to a topic to receive messages from it. Communication can be one-to-many (fan-out), many-to-one (fan-in), and many-to-many. Gaurantees at least once delivery before deletion from queue.

Scenarios

- Balancing workloads in network clusters- queue can efficiently distribute tasks
- Implementing asynchronous workflows
- Data streaming from various processes or devices
- Reliability improvement- in case zone failure
- Distributing event notifications
- Refreshing distributed caches
- Logging to multiple systems

Benefits/Features

Unified messaging, global presence, push- and pull-style subscriptions, replicated storage and guaranteed at-least-once message delivery, encryption of data at rest/transit, easy-to-use REST/JSON API

Data Model

Topic, **Subscription**, **Message** (combination of data and attributes that a publisher sends to a topic and is eventually delivered to subscribers), **Message Attribute** (key-value pair that a publisher can define for a message)

Message Flow

- Publisher creates a topic in the Cloud Pub/Sub service and sends messages to the topic.
- Messages are persisted in a message store until they are delivered and acknowledged by subscribers.
- The Pub/Sub service forwards messages from a topic to all of its subscriptions, individually. Each subscription receives messages either by pushing/pulling.
- The subscriber receives pending messages from its subscription and acknowledges message.
- When a message is acknowledged by the subscriber, it is removed from the subscription's message queue.

ML Engine

Managed infrastructure of GCP with the power and flexibility of TensorFlow. Can use it to train ML models at scale and host trained models to make predictions about new data in the cloud. Supported frameworks include Tensorflow, scikit-learn and XGBoost.

ML Workflow

Evaluate Problem: What is the problem and is ML the best approach? How will you measure model's success?
Choosing Development Environment: Supports Python 2 and 3 and supports TF, scikit-learn, XGBoost as frameworks.
Data Preparation and Exploration: Involves gathering, cleaning, transforming, exploring, splitting, and preprocessing data. Also includes feature engineering.
Model Training/Testing: Provide access to train/test data and train them in batches. Evaluate progress/results and adjust the model as needed. Export/save trained model (250 MB or smaller to deploy in ML Engine).
Hyperparameter Tuning: hyperparameters are variables that govern the training process itself, not related to the training data itself. Usually constant during training.
Prediction: host your trained ML models in the cloud and use the Cloud ML prediction service to infer target values for new data

ML APIs

Speech-to-Text: speech-to-text conversion
Text-to-Speech: text-to-speech conversion
Translation: dynamically translate between languages
Vision: derive insight (objects/text) from images
Natural Language: extract information (sentiment, intent, entity, and syntax) about the text: people, places..
Video Intelligence: extract metadata from videos

Cloud Datalab

Interactive tool (run on an instance) to explore, analyze, transform and visualize data and build machine learning models. Built on Jupyter. Datalab is free but may incus costs based on usages of other services.

Data Studio

Turns your data into informative dashboards and reports. Updates to data will automatiaclly update in dashbaord. **Query cache** remembers the queries (requests for data) issued by the components in a report (lightning bolt)-turn off for data that changes frequently, want to prioritize freshness over performance, or using a data source that incurs usage costs (e.g. BigQuery). **Prefetch cache** predicts data that could be requested by analyzing the dimensions, metrics, filters, and date range properties and controls on the report.

ML Concepts/Terminology

Features: input data used by the ML model
Feature Engineering: transforming input features to be more useful for the models. e.g. mapping categories to buckets, normalizing between -1 and 1, removing null
Train/Eval/Test: training is data used to optimize the model, evaluation is used to asses the model on new data during training, test is used to provide the final result
Classification/Regression: regression is prediction a number (e.g. housing price), classification is prediction from a set of categories(e.g. predicting red/blue/green)
Linear Regression: predicts an output by multiplying and summing input features with weights and biases
Logistic Regression: similar to linear regression but predicts a probability
Neural Network: composed of neurons (simple building blocks that actually "learn"), contains activation functions that makes it possible to predict non-linear outputs
Activation Functions: mathematical functions that introduce non-linearity to a network e.g. RELU, tanh
Sigmoid Function: function that maps very negative numbers to a number very close to 0, huge numbers close to 1, and 0 to .5. Useful for predicting probabilities
Gradient Descent/Backpropagation: fundamental loss optimizer algorithms, of which the other optimizers are usually based. Backpropagation is similar to gradient descent but for neural nets
Optimizer: operation that changes the weights and biases to reduce loss e.g. Adagrad or Adam
Weights / Biases: weights are values that the input features are multiplied by to predict an output value. Biases are the value of the output given a weight of 0.
Converge: algorithm that converges will eventually reach an optimal answer, even if very slowly. An algorithm that doesn't converge may never reach an optimal answer.
Learning Rate: rate at which optimizers change weights and biases. High learning rate generally trains faster but risks not converging, whereas a lower rate trains slower
Overfitting: model performs great on the input data but poorly on the test data (combat by dropout, early stopping, or reduce # of nodes or layers)
Bias/Variance: how much output is determined by the features. more variance often can mean overfitting, more bias can mean a bad model
Regularization: variety of approaches to reduce overfitting, including adding the weights to the loss function, randomly dropping layers (dropout)
Ensemble Learning: training multiple models with different parameters to solve the same problem
Embeddings: mapping from discrete objects, such as words, to vectors of real numbers. useful because classifiers/neural networks work well on vectors of real numbers

TensorFlow

Tensorflow is an open source software library for numerical computation using data flow graphs. Everything in TF is a graph, where nodes represent operations on data and edges represent the data. Phase 1 of TF is building up a computation graph and phase 2 is executing it. It is also distributed, meaning it can run on either a cluster of machines or just a single machine.

Tensors

In a graph, tensors are the edges and are multidimensional data arrays that flow through the graph. Central unit of data in TF and consists of a set of primitive values shaped into an array of any number of dimensions.

A tensor is characterized by its rank (# dimensions in tensor), shape (# of dimensions and size of each dimension), data type (data type of each element in tensor).

Placeholders and Variables

Variables: best way to represent shared, persistent state manipulated by your program. These are the parameters of the ML model are altered/trained during the training process. Training variables.
Placeholders: way to specify inputs into a graph that hold the place for a Tensor that will be fed at runtime. They are assigned once, do not change after. Input nodes

Popular Architectures

Linear Classifier: takes input features and combines them with weights and biases to predict output value
DNNClassifier: deep neural net, contains intermediate layers of nodes that represent "hidden features" and activation functions to represent non-linearity
ConvNets: convolutional neural nets. popular for image classification.
Transfer Learning: use existing trained models as starting points and add additional layers for the specific use case. idea is that highly trained existing models know general features that serve as a good starting point for training a small network on specific examples
RNN: recurrent neural nets, designed for handling a sequence of inputs that have "memory" of the sequence. LSTMs are a fancy version of RNNs, popular for NLP
GAN: general adversarial neural net, one model creates fake examples, and another model is served both fake example and real examples and is asked to distinguish
Wide and Deep: combines linear classifiers with deep neural net classifiers, "wide"linear parts represent memorizing specific examples and "deep" parts represent understanding high level features

Case Study: Flowlogistic I

Company Overview

Flowlogistic is a top logistics and supply chain provider. They help businesses throughout the world manage their resources and transport them to their final destination. The company has grown rapidly, expanding their offerings to include rail, truck, aircraft, and oceanic shipping.

Company Background

The company started as a regional trucking company, and then expanded into other logistics markets. Because they have not updated their infrastructure, managing and tracking orders and shipments has become a bottleneck. To improve operations, Flowlogistic developed proprietary technology for tracking shipments in real time at the parcel level. However, they are unable to deploy it because their technology stack, based on Apache Kafka, cannot support the processing volume. In addition, Flowlogistic wants to further analyze their orders and shipments to determine how best to deploy their resources.

Solution Concept

Flowlogistic wants to implement two concepts in the cloud:

- Use their proprietary technology in a real-time inventory-tracking system that indicates the location of their loads.
- Perform analytics on all their orders and shipment logs (structured and unstructured data) to determine how best to deploy resources, which customers to target, and which markets to expand into. They also want to use predictive analytics to learn earlier when a shipment will be delayed.

Business Requirements

-Build a reliable and reproducible environment with scaled parity of production
-Aggregate data in a centralized Data Lake for analysis
-Perform predictive analytics on future shipments
-Accurately track every shipment worldwide
-Improve business agility and speed of innovation through rapid provisioning of new resources
-Analyze/optimize architecture cloud performance
-Migrate fully to the cloud if all other requirements met

Technical Requirements

-Handle both streaming and batch data
-Migrate existing Hadoop workloads
-Ensure architecture is scalable and elastic to meet the changing demands of the company
-Use managed services whenever possible
-Encrypt data in flight and at rest
-Connect a VPN between the production data center and cloud environment

Case Study: Flowlogistic II

Existing Technical Environment

Flowlogistic architecture resides in a single data center:

- Databases:
  - 8 physical servers in 2 clusters
    - SQL Server: inventory, user/static data
  - 3 physical servers
    - Cassandra: metadata, tracking messages
  - 10 Kafka servers: tracking message aggregation and batch insert
- Application servers: customer front end, middleware for order/customs
  - 60 virtual machines across 20 physical servers
    - Tomcat: Java services
    - Nginx: static content
    - Batch servers
- Storage appliances
  - iSCSI for virtual machine (VM) hosts
  - Fibre Channel storage area network (FC SAN): SQL server storage
  - Network-attached storage (NAS): image storage, logs, backups
- 10 Apache Hadoop / Spark Servers
  - Core Data Lake
  - Data analysis workloads
- 20 miscellaneous servers
  - Jenkins, monitoring, bastion hosts, security scanners, billing software

CEO Statement

We have grown so quickly that our inability to upgrade our infrastructure is really hampering further growth/efficiency. We are efficient at moving shipments around the world, but we are inefficient at moving data around. We need to organize our information so we can more easily understand where our customers are and what they are shipping.

CTO Statement

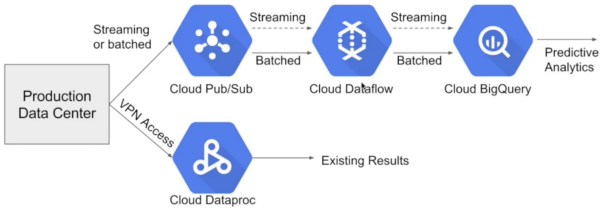
IT has never been a priority for us, so as our data has grown, we have not invested enough in our technology. I have a good staff to manage IT, but they are so busy managing our infrastructure that I cannot get them to do the things that really matter, such as organizing our data, building the analytics, and figuring out how to implement the CFO's tracking technology.

CFO Statement

Part of our competitive advantage is that we penalize ourselves for late shipments/deliveries. Knowing where our shipments are at all times has a direct correlation to our bottom line and profitability. Additionally, I don't want to commit capital to building out a server environment.

Flowlogistic Potential Solution

1. Cloud Dataproc handles the existing workloads and produces results as before (using a VPN).
2. At the same time, data is received through the data center, via either stream or batch, and sent to Pub/Sub.
3. Pub/Sub encrypts the data in transit and at rest.
4. Data is fed into Dataflow either as stream/batch data.
5. Dataflow processes the data and sends the cleaned data to BigQuery (again either as stream or batch).
6. Data can then be queried from BigQuery and predictive analysis can begin (using ML Engine, etc..)



Note: All services are fully managed., easily scalable, and can handle streaming/batch data. All technical requirements are met.

Case Study: MJTelco I

Company Overview

MJTelco is a startup that plans to build networks in rapidly growing, underserved markets around the world. The company has patents for innovative optical communications hardware. Based on these patents, they can create many reliable, high-speed backbone links with inexpensive hardware.

Company Background

Founded by experienced telecom executives, MJTelco uses technologies originally developed to overcome communications challenges in space. Fundamental to their operation, they need to create a distributed data infrastructure that drives real-time analysis and incorporates machine learning to continuously optimize their topologies. Because their hardware is inexpensive, they plan to overdeploy the network allowing them to account for the impact of dynamic regional politics on location availability and cost.

Their management and operations teams are situated all around the globe creating many-to-many relationship between data consumers and providers in their system. After careful consideration, they decided public cloud is the perfect environment to support their needs.

Solution Concept

MJTelco is running a successful proof-of-concept (PoC) project in its labs. They have two primary needs:

- Scale and harden their PoC to support significantly more data flows generated when they ramp to more than 50,000 installations.
- Refine their machine-learning cycles to verify and improve the dynamic models they use to control topology definitions. MJTelco will also use three separate operating environments – development/test, staging, and production – to meet the needs of running experiments, deploying new features, and serving production customers.

Business Requirements

-Scale up their production environment with minimal cost, instantiating resources when and where needed in an unpredictable, distributed telecom user community.
-Ensure security of their proprietary data to protect their leading-edge machine learning and analysis.
-Provide reliable and timely access to data for analysis from distributed research workers.
-Maintain isolated environments that support rapid iteration of their machine-learning models without affecting their customers.

Case Study: MJTelco II

Technical Requirements

-Ensure secure and efficient transport and storage of telemetry data.
-Rapidly scale instances to support between 10,000 and 100,000 data providers with multiple flows each.
-Allow analysis and presentation against data tables tracking up to 2 years of data storing approximately 100m records/day. Support rapid iteration of monitoring infrastructure focused on awareness of data pipeline problems both in telemetry flows and in production learning cycles.

CEO Statement

Our business model relies on our patents, analytics and dynamic machine learning. Our inexpensive hardware is organized to be highly reliable, which gives us cost advantages. We need to quickly stabilize our large distributed data pipelines to meet our reliability and capacity commitments.

CTO Statement

Our public cloud services must operate as advertised. We need resources that scale and keep our data secure. We also need environments in which our data scientists can carefully study and quickly adapt our models. Because we rely on automation to process our data, we also need our development and test environments to work as we iterate.

CFO Statement

This project is too large for us to maintain the hardware and software required for the data and analysis. Also, we cannot afford to staff an operations team to monitor so many data feeds, so we will rely on automation and infrastructure. Google Cloud's machine learning will allow our quantitative researchers to work on our high-value problems instead of problems with our data pipelines.

Choosing a Storage Option

Need	Open Source	GCP Solution
Compute, Block Storage	Persistent Disks, SSD	Persistent Disks, SSD
Media, Blob Storage	Filesystem, HDFS	Cloud Storage
SQL Interface on File Data	Hive	BigQuery
Document DB, NoSQL	CouchDB, MongoDB	DataStore
Fast Scanning NoSQL	HBase, Cassandra	BigTable
OLTP	RDBMS-MySQL	CloudSQL, Cloud Spanner
OLAP	Hive	BigQuery

Cloud Storage: unstructured data (blob)
CloudSQL: OLTP, SQL, structured and relational data, no need for horizontal scaling
Cloud Spanner: OLTP, SQL, structured and relational data, need for horizontal scaling, between RDBMS/big data
Cloud Datastore: NoSQL, document data, key-value structured but non-relational data (XML, HTML, query depends on size of result (not dataset), fast to read/slow to write
BigTable: NoSQL, key-value data, columnar, good for sparse data, sensitive to hot spotting, high throughput and scalability for non-structured key/value data, where each value is typically no larger than 10 MB

Solutions

Data Lifecycle

At each stage, GCP offers multiple services to manage your data.

1. **Ingest**: first stage is to pull in the raw data, such as streaming data from devices, on-premises batch data, application logs, or mobile-app user events and analytics
2. **Store**: after the data has been retrieved, it needs to be stored in a format that is durable and can be easily accessed
3. **Process and Analyze**: the data is transformed from raw form into actionable information
4. **Explore and Visualize**: convert the results of the analysis into a format that is easy to draw insights from and to share with colleagues and peers

Ingest

There are a number of approaches to collect raw data, based on the data's size, source, and latency.

- **Application**: data from application events, log files or user events, typically collected in a push model, where the application calls an API to send the data to storage (Stackdriver Logging, Pub/Sub, CloudSQL, Datastore, Bigtable, Spanner)
- **Streaming**: data consists of a continuous stream of small, asynchronous messages. Common uses include telemetry, or collecting data from geographically dispersed devices (IoT) and user events and analytics (Pub/Sub)
- **Batch**: large amounts of data are stored in a set of files that are transferred to storage in bulk. common use cases include scientific workloads, backups, migration (Storage, Transfer Service, Appliance)

Solutions- Part II

Storage

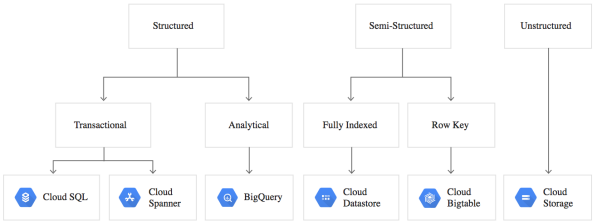
Cloud Storage: durable and highly-available object storage for structured and unstructured data

Cloud SQL: fully managed, cloud RDBMS that offers both MySQL and PostgreSQL engines with built-in support for replication, for low-latency, transactional, relational database workloads. supports RDBMS workloads up to 10 TB (storing financial transactions, user credentials, customer orders)

BigTable: managed, high-performance NoSQL database service designed for terabyte- to petabyte-scale workloads. suitable for large-scale, high-throughput workloads such as advertising technology or IoT data infrastructure. does not support multi-row transactions, SQL queries or joins, consider Cloud SQL or Cloud Datastore instead

Cloud Spanner: fully managed relational database service for mission-critical OLTP applications. horizontally scalable, and built for strong consistency, high availability, and global scale. supports schemas, ACID transactions, and SQL queries (use for retail and global supply chain, ad tech, financial services)

BigQuery: stores large quantities of data for query and analysis instead of transactional processing



Exploration and Visualization Data exploration and visualization to better understand the results of the processing and analysis.

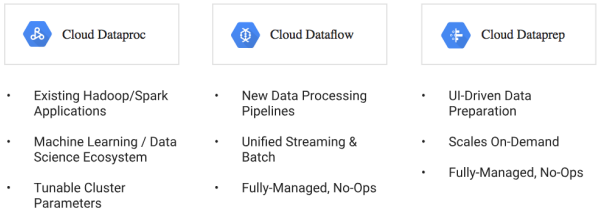
- **Cloud Datalab**: interactive web-based tool that you can use to explore, analyze and visualize data built on top of Jupyter notebooks. runs on a VM and automatically saved to persistent disks and can be stored in GC Source Repo (git repo)
- **Data Studio**: drag-and-drop report builder that you can use to visualize data into reports and dashboards that can then be shared with others, backed by live data, that can be shared and updated easily. data sources can be data files, Google Sheets, Cloud SQL, and BigQuery. supports **query** and **prefetch cache**: query remembers previous queries and if data not found, goes to prefetch cache, which predicts data that could be requested (disable if data changes frequently/using data source incurs charges)

Solutions- Part III

Process and Analyze

In order to derive business value and insights from data, you must transform and analyze it. This requires a processing framework that can either analyze the data directly or prepare the data for downstream analysis, as well as tools to analyze and understand processing results.

- **Processing**: data from source systems is cleansed, normalized, and processed across multiple machines, and stored in analytical systems
- **Analysis**: processed data is stored in systems that allow for ad-hoc querying and exploration
- **Understanding**: based on analytical results, data is used to train and test automated machine-learning models



Processing

- **Cloud Dataproc**: migrate your existing Hadoop or Spark deployments to a fully-managed service that automates cluster creation, simplifies configuration and management of your cluster, has built-in monitoring and utilization reports, and can be shutdown when not in use
- **Cloud Dataflow**: designed to simplify big data for both streaming and batch workloads, focus on filtering, aggregating, and transforming your data
- **Cloud Dataprep**: service for visually exploring, cleaning, and preparing data for analysis. can transform data of any size stored in CSV, JSON, or relational-table formats

Analyzing and Querying

- **BigQuery**: query using SQL, all data encrypted, user analysis, device and operational metrics, business intelligence
- **Task-Specific ML**: Vision, Speech, Natural Language, Translation, Video Intelligence
- **ML Engine**: managed platform you can use to run custom machine learning models at scale

streaming data: use Pub/Sub and Dataflow in combination

VIP Cheatsheet: Statistics

Shervine Amidi

August 13, 2018

Parameter estimation

❒ **Random sample** – A random sample is a collection of n random variables $X_1, ..., X_n$ that are independent and identically distributed with X.

❒ **Estimator** – An estimator $\hat{\theta}$ is a function of the data that is used to infer the value of an unknown parameter θ in a statistical model.

❒ **Bias** – The bias of an estimator $\hat{\theta}$ is defined as being the difference between the expected value of the distribution of $\hat{\theta}$ and the true value, i.e.:

$$\boxed{\text{Bias}(\hat{\theta}) = E[\hat{\theta}] - \theta}$$

Remark: an estimator is said to be unbiased when we have $E[\hat{\theta}] = \theta$.

❒ **Sample mean and variance** – The sample mean and the sample variance of a random sample are used to estimate the true mean μ and the true variance σ^2 of a distribution, are noted $\overline{X}$ and s^2 respectively, and are such that:

$$\boxed{\overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i} \quad \text{and} \quad \boxed{s^2 = \hat{\sigma}^2 = \frac{1}{n-1}\sum_{i=1}^{n}(X_i - \overline{X})^2}$$

❒ **Central Limit Theorem** – Let us have a random sample $X_1, ..., X_n$ following a given distribution with mean μ and variance σ^2, then we have:

$$\boxed{\overline{X} \underset{n\to+\infty}{\sim} \mathcal{N}\left(\mu, \frac{\sigma}{\sqrt{n}}\right)}$$

Confidence intervals

❒ **Confidence level** – A confidence interval $CI_{1-\alpha}$ with confidence level $1-\alpha$ of a true parameter θ is such that $1-\alpha$ of the time, the true value is contained in the confidence interval:

$$\boxed{P(\theta \in CI_{1-\alpha}) = 1 - \alpha}$$

❒ **Confidence interval for the mean** – When determining a confidence interval for the mean μ, different test statistics have to be computed depending on which case we are in. The following table sums it up:

Distribution	Sample size	σ^2	Statistic	$1-\alpha$ confidence interval
$X_i \sim \mathcal{N}(\mu, \sigma)$	any	known	$\frac{\overline{X}-\mu}{\frac{\sigma}{\sqrt{n}}} \sim \mathcal{N}(0,1)$	$\left[\overline{X} - z_{\frac{\alpha}{2}}\frac{\sigma}{\sqrt{n}}, \overline{X} + z_{\frac{\alpha}{2}}\frac{\sigma}{\sqrt{n}}\right]$
	small	unknown	$\frac{\overline{X}-\mu}{\frac{s}{\sqrt{n}}} \sim t_{n-1}$	$\left[\overline{X} - t_{\frac{\alpha}{2}}\frac{s}{\sqrt{n}}, \overline{X} + t_{\frac{\alpha}{2}}\frac{s}{\sqrt{n}}\right]$
$X_i \sim$ any	large	known	$\frac{\overline{X}-\mu}{\frac{\sigma}{\sqrt{n}}} \sim \mathcal{N}(0,1)$	$\left[\overline{X} - z_{\frac{\alpha}{2}}\frac{\sigma}{\sqrt{n}}, \overline{X} + z_{\frac{\alpha}{2}}\frac{\sigma}{\sqrt{n}}\right]$
		unknown	$\frac{\overline{X}-\mu}{\frac{s}{\sqrt{n}}} \sim \mathcal{N}(0,1)$	$\left[\overline{X} - z_{\frac{\alpha}{2}}\frac{s}{\sqrt{n}}, \overline{X} + z_{\frac{\alpha}{2}}\frac{s}{\sqrt{n}}\right]$
$X_i \sim$ any	small	any	Go home!	Go home!

❒ **Confidence interval for the variance** – The single-line table below sums up the test statistic to compute when determining the confidence interval for the variance.

Distribution	Sample size	μ	Statistic	$1-\alpha$ confidence interval
$X_i \sim \mathcal{N}(\mu,\sigma)$	any	any	$\frac{s^2(n-1)}{\sigma^2} \sim \chi^2_{n-1}$	$\left[\frac{s^2(n-1)}{\chi^2_2}, \frac{s^2(n-1)}{\chi^2_1}\right]$

Hypothesis testing

❒ **Errors** – In a hypothesis test, we note α and β the type I and type II errors respectively. By noting T the test statistic and R the rejection region, we have:

$$\boxed{\alpha = P(T \in R | H_0 \text{ true})} \quad \text{and} \quad \boxed{\beta = P(T \notin R | H_1 \text{ true})}$$

❒ **p-value** – In a hypothesis test, the p-value is the probability under the null hypothesis of having a test statistic T at least as extreme as the one that we observed T_0. In particular, when T follows a zero-centered symmetric distribution under H_0, we have:

Case	Left-sided	Right-sided	Two-sided
p-value	$P(T \leqslant T_0 \| H_0 \text{ true})$	$P(T \geqslant T_0 \| H_0 \text{ true})$	$P(\|T\| \geqslant \|T_0\| \| H_0 \text{ true})$

❒ **Sign test** – The sign test is a non-parametric test used to determine whether the median of a sample is equal to the hypothesized median. By noting V the number of samples falling to the right of the hypothesized median, we have:

Statistic when $np < 5$	Statistic when $np \geqslant 5$
$V \underset{H_0}{\sim} \mathcal{B}\left(n, p = \frac{1}{2}\right)$	$Z = \frac{V - \frac{n}{2}}{\frac{\sqrt{n}}{2}} \underset{H_0}{\sim} \mathcal{N}(0,1)$

❒ **Testing for the difference in two means** – The table below sums up the test statistic to compute when performing a hypothesis test where the null hypothesis is:

$$H_0 \quad : \quad \mu_X - \mu_Y = \delta$$

Distribution of X_i, Y_i	n_X, n_Y	σ_X^2, σ_Y^2	Statistic
Normal	any	known	$\frac{(\overline{X}-\overline{Y})-\delta}{\sqrt{\frac{\sigma_X^2}{n_X}+\frac{\sigma_Y^2}{n_Y}}} \underset{H_0}{\sim} \mathcal{N}(0,1)$
	large	unknown	$\frac{(\overline{X}-\overline{Y})-\delta}{\sqrt{\frac{s_X^2}{n_X}+\frac{s_Y^2}{n_Y}}} \underset{H_0}{\sim} \mathcal{N}(0,1)$
	small	unknown $\sigma_X = \sigma_Y$	$\frac{(\overline{X}-\overline{Y})-\delta}{s\sqrt{\frac{1}{n_X}+\frac{1}{n_Y}}} \underset{H_0}{\sim} t_{n_X+n_Y-2}$
Normal, paired $D_i = X_i - Y_i$	any $n_X = n_Y$	unknown	$\frac{\overline{D}-\delta}{\frac{s_D}{\sqrt{n}}} \underset{H_0}{\sim} t_{n-1}$

❒ **χ^2 goodness of fit test** – By noting k the number of bins, n the total number of samples, p_i the probability of success in each bin and Y_i the associated number of samples, we can use the test statistic T defined below to test whether or not there is a good fit. If $np_i \geqslant 5$, we have:

$$\boxed{T = \sum_{i=1}^{k} \frac{(Y_i - np_i)^2}{np_i} \underset{H_0}{\sim} \chi^2_{df}} \quad \text{with} \quad \boxed{df = (k-1) - \#(\text{estimated parameters})}$$

❒ **Test for arbitrary trends** – Given a sequence, the test for arbitrary trends is a non-parametric test, whose aim is to determine whether the data suggest the presence of an increasing trend:

$$\boxed{H_0 : \text{no trend}} \quad \text{versus} \quad \boxed{H_1 : \text{there is an increasing trend}}$$

If we note x the number of transpositions in the sequence, the p-value is computed as:

$$\boxed{p\text{-value} = P(T \leqslant x)}$$

Regression analysis

In the following section, we will note $(x_1, Y_1), ..., (x_n, Y_n)$ a collection of n data points.

❒ **Simple linear model** – Let X be a deterministic variable and Y a dependent random variable. In the context of a simple linear model, we assume that Y is linked to X via the regression coefficients α, β and a random variable $e \sim \mathcal{N}(0,\sigma)$, where e is referred as the error. We estimate Y, α, β by $\hat{Y}, A, B$ and have:

$$\boxed{Y = \alpha + \beta X + e} \quad \text{and} \quad \boxed{\hat{Y}_i = A + Bx_i}$$

❒ **Notations** – Given n data points (x_i, Y_i), we define S_{XY},S_{XX} and S_{YY} as follows:

$$\boxed{S_{XY} = \sum_{i=1}^{n}(x_i - \overline{x})(Y_i - \overline{Y}) \quad \text{and} \quad S_{XX} = \sum_{i=1}^{n}(x_i - \overline{x})^2 \quad \text{and} \quad S_{YY} = \sum_{i=1}^{n}(Y_i - \overline{Y})^2}$$

❒ **Sum of squared errors** – By keeping the same notations, we define the sum of squared errors, also known as SSE, as follows:

$$\boxed{SSE = \sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 = \sum_{i=1}^{n}(Y_i - (A + Bx_i))^2 = S_{YY} - BS_{XY}}$$

❒ **Least-squares estimates** – When estimating the coefficients α, β with the least-squares method which is done by minimizing the SSE, we obtain the estimates A, B defined as follows:

$$\boxed{A = \overline{Y} - \frac{S_{XY}}{S_{XX}}\overline{x} \quad \text{and} \quad B = \frac{S_{XY}}{S_{XX}}}$$

❒ **Key results** – When σ is unknown, this parameter is estimated by the unbiased estimator s^2 defined as follows:

$$\boxed{s^2 = \frac{S_{YY} - BS_{XY}}{n-2}} \quad \text{and we have} \quad \boxed{\frac{s^2(n-2)}{\sigma^2} \sim \chi^2_{n-2}}$$

The table below sums up the properties surrounding the least-squares estimates A, B when σ is known or not:

Coeff	σ	Statistic	$1-\alpha$ confidence interval
α	known	$\frac{A-\alpha}{\sigma\sqrt{\frac{1}{n}+\frac{\overline{X}^2}{S_{XX}}}} \sim \mathcal{N}(0,1)$	$\left[A - z_{\frac{\alpha}{2}}\sigma\sqrt{\frac{1}{n}+\frac{\overline{X}^2}{S_{XX}}}, A + z_{\frac{\alpha}{2}}\sigma\sqrt{\frac{1}{n}+\frac{\overline{X}^2}{S_{XX}}}\right]$
	unknown	$\frac{A-\alpha}{s\sqrt{\frac{1}{n}+\frac{\overline{X}^2}{S_{XX}}}} \sim t_{n-2}$	$\left[A - t_{\frac{\alpha}{2}}s\sqrt{\frac{1}{n}+\frac{\overline{X}^2}{S_{XX}}}, A + t_{\frac{\alpha}{2}}s\sqrt{\frac{1}{n}+\frac{\overline{X}^2}{S_{XX}}}\right]$
β	known	$\frac{B-\beta}{\frac{\sigma}{\sqrt{S_{XX}}}} \sim \mathcal{N}(0,1)$	$\left[B - z_{\frac{\alpha}{2}}\frac{\sigma}{\sqrt{S_{XX}}}, B + z_{\frac{\alpha}{2}}\frac{\sigma}{\sqrt{S_{XX}}}\right]$
	unknown	$\frac{B-\beta}{\frac{s}{\sqrt{S_{XX}}}} \sim t_{n-2}$	$\left[B - t_{\frac{\alpha}{2}}\frac{s}{\sqrt{S_{XX}}}, B + t_{\frac{\alpha}{2}}\frac{s}{\sqrt{S_{XX}}}\right]$

Correlation analysis

❒ **Sample correlation coefficient** – The correlation coefficient is in practice estimated by the sample correlation coefficient, often noted r or $\hat{\rho}$, which is defined as:

$$\boxed{r = \hat{\rho} = \frac{S_{XY}}{\sqrt{S_{XX}S_{YY}}}} \quad \text{with} \quad \boxed{\frac{r\sqrt{n-2}}{\sqrt{1-r^2}} \underset{H_0}{\sim} t_{n-2}} \quad \text{for } H_0 : \rho = 0$$

❒ **Correlation properties** – By noting $V_1 = V - \frac{z_{\frac{\alpha}{2}}}{\sqrt{n-3}}$, $V_2 = V + \frac{z_{\frac{\alpha}{2}}}{\sqrt{n-3}}$ with $V = \frac{1}{2}\ln\left(\frac{1+r}{1-r}\right)$, the table below sums up the key results surrounding the correlation coefficient estimate:

Sample size	Standardized statistic	$1-\alpha$ confidence interval for ρ
large	$\frac{V - \frac{1}{2}\ln\left(\frac{1+\rho}{1-\rho}\right)}{\frac{1}{\sqrt{n-3}}} \underset{n\gg 1}{\sim} \mathcal{N}(0,1)$	$\left[\frac{e^{2V_1}-1}{e^{2V_1}+1}, \frac{e^{2V_2}-1}{e^{2V_2}+1}\right]$

Data Science Cheatsheet

Compiled by Maverick Lin (http://mavericklin.com)

Last Updated August 13, 2018

What is Data Science?

Multi-disciplinary field that brings together concepts from computer science, statistics/machine learning, and data analysis to understand and extract insights from the ever-increasing amounts of data.

Two paradigms of data research.

1. **Hypothesis-Driven:** Given a problem, what kind of data do we need to help solve it?
2. **Data-Driven:** Given some data, what interesting problems can be solved with it?

The heart of data science is to always ask questions. Always be curious about the world.

1. What can we learn from this data?
2. What actions can we take once we find whatever it is we are looking for?

Types of Data

Structured: Data that has predefined structures. e.g. tables, spreadsheets, or relational databases.
Unstructured Data: Data with no predefined structure, comes in any size or form, cannot be easily stored in tables. e.g. blobs of text, images, audio
Quantitative Data: Numerical. e.g. height, weight
Categorical Data: Data that can be labeled or divided into groups. e.g. race, sex, hair color.
Big Data: Massive datasets, or data that contains greater *variety* arriving in increasing *volumes* and with ever-higher *velocity* (3 Vs). Cannot fit in the memory of a single machine.

Data Sources/Fomats

Most Common Data Formats CSV, XML, SQL, JSON, Protocol Buffers
Data Sources Companies/Proprietary Data, APIs, Government, Academic, Web Scraping/Crawling

Main Types of Problems

Two problems arise repeatedly in data science.
Classification: Assigning something to a discrete set of possibilities. e.g. spam or non-spam, Democrat or Republican, blood type (A, B, AB, O)
Regression: Predicting a numerical value. e.g. someone's income, next year GDP, stock price

Probability Overview

Probability theory provides a framework for reasoning about likelihood of events.

Terminology

Experiment: procedure that yields one of a possible set of outcomes e.g. repeatedly tossing a die or coin
Sample Space S: set of possible outcomes of an experiment e.g. if tossing a die, S = (1,2,3,4,5,6
Event E: set of outcomes of an experiment e.g. event that a roll is 5, or the event that sum of 2 rolls is 7
Probability of an Outcome s or P(s): number that satisfies 2 properties

1. for each outcome s, $0 \leq \text{P(s)} \leq 1$
2. $\sum \text{p(s)} = 1$

Probability of Event E: sum of the probabilities of the outcomes of the experiment: $\text{p(E)} = \sum_{s \subset E} \text{p(s)}$
Random Variable V: numerical function on the outcomes of a probability space
Expected Value of Random Variable V: $\text{E(V)} = \sum_{s \subset S} \text{p(s)} * \text{V(s)}$

Independence, Conditional, Compound

Independent Events: A and B are independent iff:

$$\text{P(A} \cap \text{B)} = \text{P(A)P(B)}$$
$$\text{P}(A|B) = \text{P(A)}$$
$$\text{P}(B|A) = \text{P(B)}$$

Conditional Probability: $\text{P}(A|B) = \text{P(A,B)/P(B)}$
Bayes Theorem: $\text{P}(A|B) = \text{P}(B|A)\text{P(A)/P(B)}$
Joint Probability: $\text{P(A,B)} = \text{P}(B|A)\text{P(A)}$
Marginal Probability: P(A)

Probability Distributions

Probability Density Function (PDF) Gives the probability that a rv takes on the value x: $p_X(x) = P(X = x)$
Cumulative Density Function (CDF) Gives the probability that a random variable is less than or equal to x: $F_X(x) = P(X \leq x)$
Note: The PDF and the CDF of a given random variable contain exactly the same information.

Descriptive Statistics

Provides a way of capturing a given data set or sample. There are two main types: **centrality** and **variability** measures.

Centrality

Arithmetic Mean Useful to characterize symmetric distributions without outliers $\mu_X = \frac{1}{n}\sum x$
Geometric Mean Useful for averaging ratios. Always less than arithmetic mean $= \sqrt[n]{a_1 a_2 ... a_3}$
Median Exact middle value among a dataset. Useful for skewed distribution or data with outliers.
Mode Most frequent element in a dataset.

Variability

Standard Deviation Measures the squares differences between the individual elements and the mean

$$\sigma = \sqrt{\frac{\sum_{i=1}^{N}(x_i - \overline{x})^2}{N-1}}$$

Variance $\text{V} = \sigma^2$

Interpreting Variance

Variance is an inherent part of the universe. It is impossible to obtain the same results after repeated observations of the same event due to random noise/error. Variance can be explained away by attributing to sampling or measurement errors. Other times, the variance is due to the random fluctuations of the universe.

Correlation Analysis

Correlation coefficients r(X,Y) is a statistic that measures the degree that Y is a function of X and vice versa. Correlation values range from -1 to 1, where 1 means fully correlated, -1 means negatively-correlated, and 0 means no correlation.
Pearson Coefficient Measures the degree of the relationship between linearly related variables

$$\text{r} = \frac{Cov(X,Y)}{\sigma(X)\sigma(Y)}$$

Spearman Rank Coefficient Computed on ranks and depicts monotonic relationships

Note: Correlation does not imply causation!

Data Cleaning

Data Cleaning is the process of turning raw data into a clean and analyzable data set. "Garbage in, garbage out." Make sure garbage doesn't get put in.

Errors vs. Artifacts

1. **Errors:** information that is lost during acquisition and can never be recovered e.g. power outage, crashed servers
2. **Artifacts:** systematic problems that arise from the data cleaning process. these problems can be corrected but we must first discover them

Data Compatibility

Data compatibility problems arise when merging datasets. Make sure you are comparing "apples to apples" and not "apples to oranges". Main types of conversions/unifications:

- **units** (metric vs. imperial)
- **numbers** (decimals vs. integers),
- **names** (John Smith vs. Smith, John),
- **time/dates** (UNIX vs. UTC vs. GMT),
- **currency** (currency type, inflation-adjusted, dividends)

Data Imputation

Process of dealing with missing values. The proper methods depend on the type of data we are working with. General methods include:

- Drop all records containing missing data
- Heuristic-Based: make a reasonable guess based on knowledge of the underlying domain
- Mean Value: fill in missing data with the mean
- Random Value
- Nearest Neighbor: fill in missing data using similar data points
- Interpolation: use a method like linear regression to predict the value of the missing data

Outlier Detection

Outliers can interfere with analysis and often arise from mistakes during data collection. It makes sense to run a "sanity check".

Miscellaneous

Lowercasing, removing non-alphanumeric, repairing, unidecode, removing unknown characters

Note: When cleaning data, always maintain both the raw data and the cleaned version(s). The raw data should be kept intact and preserved for future use. Any type of data cleaning/analysis should be done on a copy of the raw data.

Feature Engineering

Feature engineering is the process of using domain knowledge to create features or input variables that help machine learning algorithms perform better. Done correctly, it can help increase the predictive power of your models. Feature engineering is more of an art than science. FE is one of the most important steps in creating a good model. As Andrew Ng puts it:

"Coming up with features is difficult, time-consuming, requires expert knowledge. 'Applied machine learning' is basically feature engineering."

Continuous Data

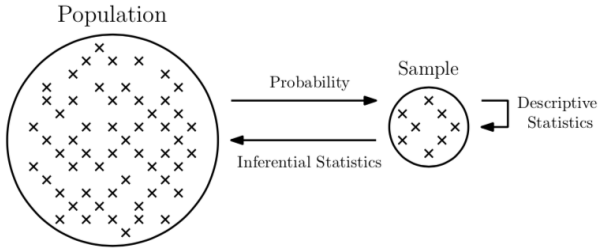
Raw Measures: data that hasn't been transformed yet
Rounding: sometimes precision is noise; round to nearest integer, decimal etc..
Scaling: log, z-score, minmax scale
Imputation: fill in missing values using mean, median, model output, etc..
Binning: transforming numeric features into categorical ones (or binned) e.g. values between 1-10 belong to A, between 10-20 belong to B, etc.
Interactions: interactions between features: e.g. subtraction, addition, multiplication, statistical test
Statistical: log/power transform (helps turn skewed distributions more normal), Box-Cox
Row Statistics: number of NaN's, 0's, negative values, max, min, etc
Dimensionality Reduction: using PCA, clustering, factor analysis etc

Discrete Data

Encoding: since some ML algorithms cannot work on categorical data, we need to turn categorical data into numerical data or vectors
Ordinal Values: convert each distinct feature into a random number (e.g. [r,g,b] becomes [1,2,3])
One-Hot Encoding: each of the m features becomes a vector of length m with containing only one 1 (e.g. [r, g, b] becomes [[1,0,0],[0,1,0],[0,0,1]])
Feature Hashing Scheme: turns arbitrary features into indices in a vector or matrix
Embeddings: if using words, convert words to vectors (word embeddings)

Statistical Analysis

Process of statistical reasoning: there is an underlying population of possible things we can potentially observe and only a small subset of them are actually sampled (ideally at random). Probability theory describes what properties our sample should have given the properties of the population, but ***statistical inference*** allows us to deduce what the full population is like after analyzing the sample.



Sampling From Distributions

Inverse Transform Sampling Sampling points from a given probability distribution is sometimes necessary to run simulations or whether your data fits a particular distribution. The general technique is called *inverse transform sampling* or Smirnov transform. First draw a random number p between [0,1]. Compute value x such that the CDF equals p: $F_X(x) = p$. Use x as the value to be the random value drawn from the distribution described by $F_X(x)$.

Monte Carlo Sampling In higher dimensions, correctly sampling from a given distribution becomes more tricky. Generally want to use Monte Carlo methods, which typically follow these rules: define a domain of possible inputs, generate random inputs from a probability distribution over the domain, perform a deterministic calculation, and analyze the results.

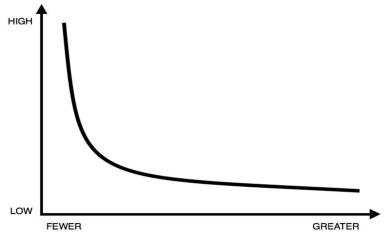
Classic Statistical Distributions

Binomial Distribution (Discrete)
Assume X is distributed Bin(n,p). X is the number of "successes" that we will achieve in n independent trials, where each trial is either a success or failure and each success occurs with the same probability p and each failure occurs with probability q=1-p.
PDF: $P(X = x) = \binom{n}{k} p^x (1-p)^{n-x}$
EV: $\mu = np$ Variance = npq

Normal/Gaussian Distribution (Continuous)
Assume X in distributed $\mathcal{N}(\mu, \sigma^2)$. It is a bell-shaped and symmetric distribution. Bulk of the values lie close to the mean and no value is too extreme. Generalization of the binomial distribution as n $\to \infty$.
PDF: $P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2}$
EV: μ Variance: σ^2
Implications: 68%-95%-99% rule. 68% of probability mass fall within 1σ of the mean, 95% within 2σ, and 99.7% within 3σ.

Poisson Distribution (Discrete)
Assume X is distributed Pois(λ). Poisson expresses the probability of a given number of events occurring in a fixed interval of time/space if these events occur independently and with a known constant rate λ.
PDF: $P(x) = \frac{e^{-\lambda}\lambda^x}{x!}$ EV: λ Variance = λ

Power Law Distributions (Discrete)
Many data distributions have much longer tails than the normal or Poisson distributions. In other words, the change in one quantity varies as a *power* of another quantity. It helps measure the inequality in the world. e.g. wealth, word frequency and Pareto Principle (80/20 Rule)
PDF: P(X=x) = $cx^{-\alpha}$, where α is the law's exponent and c is the normalizing constant



Modeling- Overview

Modeling is the process of incorporating information into a tool which can forecast and make predictions. Usually, we are dealing with statistical modeling where we want to analyze relationships between variables. Formally, we want to estimate a function $f(X)$ such that:

$$Y = f(X) + \epsilon$$

where X = $(X_1, X_2, ...X_p)$ represents the input variables, Y represents the output variable, and ϵ represents random error.

Statistical learning is set of approaches for estimating this $f(X)$.

Why Estimate f(X)?

Prediction: once we have a good estimate $\hat{f}(X)$, we can use it to make predictions on new data. We treat $\hat{f}$ as a black box, since we only care about the accuracy of the predictions, not why or how it works.
Inference: we want to understand the relationship between X and Y. We can no longer treat $\hat{f}$ as a black box since we want to understand how Y changes with respect to X = $(X_1, X_2, ...X_p)$

More About ϵ

The error term ϵ is composed of the reducible and irreducible error, which will prevent us from ever obtaining a perfect $\hat{f}$ estimate.

- **Reducible**: error that can potentially be reduced by using the most appropriate statistical learning technique to estimate f. The goal is to minimize the reducible error.
- **Irreducible**: error that cannot be reduced no matter how well we estimate f. Irreducible error is unknown and unmeasurable and will always be an upper bound for ϵ.

Note: There will always be trade-offs between model flexibility (prediction) and model interpretability (inference). This is just another case of the bias-variance trade-off. Typically, as flexibility increases, interpretability decreases. Much of statistical learning/modeling is finding a way to balance the two.

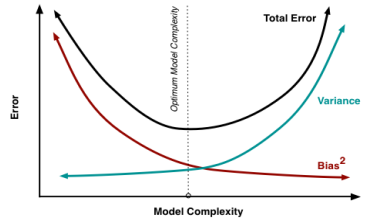
Modeling- Philosophies

Modeling is the process of incorporating information into a tool which can forecast and make predictions. Designing and validating models is important, as well as evaluating the performance of models. Note that the best forecasting model may not be the most accurate one.

Philosophies of Modeling

Occam's Razor Philosophical principle that the simplest explanation is the best explanation. In modeling, if we are given two models that predicts equally well, we should choose the simpler one. Choosing the more complex one can often result in overfitting.
Bias Variance Trade-Off Inherent part of predictive modeling, where models with lower bias will have higher variance and vice versa. Goal is to achieve low bias and low variance.

- ***Bias***: error from incorrect assumptions to make target function easier to learn (high bias $\to$ missing relevant relations or underfitting)
- ***Variance***: error from sensitivity to fluctuations in the dataset, or how much the target estimate would differ if different training data was used (high variance $\to$ modeling noise or overfitting)



No Free Lunch Theorem No single machine learning algorithm is better than all the others on all problems. It is common to try multiple models and find one that works best for a particular problem.

Thinking Like Nate Silver

1. Think Probabilistically Probabilistic forecasts are more meaningful than concrete statements and should be reported as probability distributions (including σ along with mean prediction μ.
2. Incorporate New Information Use live models, which continually updates using new information. To update, use Bayesian reasoning to calculate how probabilities change in response to new evidence.
3. Look For Consensus Forecast Use multiple distinct sources of evidence. Ssome models operate this way, such as boosting and bagging, which uses large number of weak classifiers to produce a strong one.

Modeling- Taxonomy

There are many different types of models. It is important to understand the trade-offs and when to use a certain type of model.

Parametric vs. Nonparametric

- **Parametric**: models that first make an assumption about a function form, or shape, of f (linear). Then fits the model. This reduces estimating f to just estimating set of parameters, but if our assumption was wrong, will lead to bad results.
- **Non-Parametric**: models that don't make any assumptions about f, which allows them to fit a wider range of shapes; but may lead to overfitting

Supervised vs. Unsupervised

- **Supervised**: models that fit input variables $x_i = (x_1, x_2, ...x_n)$ to a known output variables $y_i = (y_1, y_2, ...y_n)$
- **Unsupervised**: models that take in input variables $x_i = (x_1, x_2, ...x_n)$, but they do not have an associated output to supervise the training. The goal is understand relationships between the variables or observations.

Blackbox vs. Descriptive

- **Blackbox**: models that make decisions, but we do not know what happens "under the hood" e.g. deep learning, neural networks
- **Descriptive**: models that provide insight into *why* they make their decisions e.g. linear regression, decision trees

First-Principle vs. Data-Driven

- **First-Principle**: models based on a prior belief of how the system under investigation works, incorporates domain knowledge (ad-hoc)
- **Data-Driven**: models based on observed correlations between input and output variables

Deterministic vs. Stochastic

- **Deterministic**: models that produce a single "prediction" e.g. yes or no, true or false
- **Stochastic**: models that produce probability distributions over possible events

Flat vs. Hierarchical

- **Flat**: models that solve problems on a single level, no notion of subproblems
- **Hierarchical**: models that solve several different nested subproblems

Modeling- Evaluation Metrics

Need to determine how good our model is. Best way to assess models is out-of-sample predictions (data points your model has never seen).

Classification

	Predicted Yes	Predicted No
Actual Yes	True Positives (TP)	False Negatives (FN)
Actual No	False Positives (FP)	True Negatives (TN)

Accuracy: ratio of correct predictions over total predictions. Misleading when class sizes are substantially different. $accuracy = \frac{TP+TN}{TP+TN+FN+FP}$
Precision: how often the classifier is correct when it predicts positive: $precision = \frac{TP}{TP+FP}$
Recall: how often the classifier is correct for all positive instances: $recall = \frac{TP}{TP+FN}$
F-Score: single measurement to describe performance: $\mathrm{F} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}}$
ROC Curves: plots true positive rates and false positive rates for various thresholds, or where the model determines if a data point is positive or negative (e.g. if >0.8, classify as positive). Best possible area under the ROC curve (AUC) is 1, while random is 0.5, or the main diagonal line.

Regression

Errors are defined as the difference between a prediction y′ and the actual result y.
Absolute Error: $\Delta = y\prime - y$
Squared Error: $\Delta^2 = (y\prime - y)^2$
Mean-Squared Error: $MSE = \frac{1}{n}\sum_{i=1}^{n}(y\prime_i - y_i)^2$
Root Mean-Squared Error: $\mathrm{RMSD} = \sqrt{MSE}$
Absolute Error Distribution: Plot absolute error distribution: should be symmetric, centered around 0, bell-shaped, and contain rare extreme outliers.

Modeling- Evaluation Environment

Evaluation metrics provides use with the tools to estimate errors, but what should be the process to obtain the best estimate? Resampling involves repeatedly drawing samples from a training set and refitting a model to each sample, which provides us with additional information compared to fitting the model once, such as obtaining a better estimate for the test error.

Key Concepts

Training Data: data used to fit your models or the set used for learning
Validation Data: data used to tune the parameters of a model
Test Data: data used to evaluate how good your model is. Ideally your model should never touch this data until final testing/evaluation

Cross Validation

Class of methods that estimate test error by holding out a subset of training data from the fitting process.
Validation Set: split data into training set and validation set. Train model on training and estimate test error using validation. e.g. 80-20 split
Leave-One-Out CV (LOOCV): split data into training set and validation set, but the validation set consists of 1 observation. Then repeat n-1 times until all observations have been used as validation. Test erro is the average of these n test error estimates.
k-Fold CV: randomly divide data into k groups (folds) of approximately equal size. First fold is used as validation and the rest as training. Then repeat k times and find average of the k estimates.

Bootstrapping

Methods that rely on random sampling with replacement. Bootstrapping helps with quantifying uncertainty associated with a given estimate or model.

Amplifying Small Data Sets

What can we do it we don't have enough data?

- **Create Negative Examples**- e.g. classifying presidential candidates, most people would be unqualified so label most as unqualified
- **Synthetic Data**- create additional data by adding noise to the real data

Linear Regression

Linear regression is a simple and useful tool for predicting a quantitative response. The relationship between input variables $X = (X_1, X_2, ...X_p)$ and output variable Y takes the form:

$$Y \approx \beta_0 + \beta_1 X_1 + ... + \beta_p X_p + \epsilon$$

$\beta_0 ... \beta_p$ are the unknown coefficients (parameters) which we are trying to determine. The best coefficients will lead us to the best "fit", which can be found by minimizing the *residual sum squares* (RSS), or the sum of the differences between the actual ith value and the predicted ith value. $\text{RSS} = \sum_{i=1}^{n} e_i$, where $e_i = y_i - \hat{y_i}$

How to find best fit?

Matrix Form: We can solve the closed-form equation for coefficient vector w: $w = (X^T X)^{-1} X^T Y$. X represents the input data and Y represents the output data. This method is used for smaller matrices, since inverting a matrix is computationally expensive.

Gradient Descent: First-order optimization algorithm. We can find the minimum of a *convex* function by starting at an arbitrary point and repeatedly take steps in the downward direction, which can be found by taking the negative direction of the gradient. After several iterations, we will eventually converge to the minimum. In our case, the minimum corresponds to the coefficients with the minimum error, or the best line of fit. The learning rate α determines the size of the steps we take in the downward direction.

Gradient descent algorithm in two dimensions. Repeat until convergence.

1. $w_0^{t+1} := w_0^t - \alpha \frac{\partial}{\partial w_0} J(w_0, w_1)$
2. $w_1^{t+1} := w_1^t - \alpha \frac{\partial}{\partial w_1} J(w_0, w_1)$

For non-convex functions, gradient descent no longer guarantees an optimal solutions since there may be local minimas. Instead, we should run the algorithm from different starting points and use the best local minima we find for the solution.

Stochastic Gradient Descent: instead of taking a step after sampling the *entire* training set, we take a small batch of training data at random to determine our next step. Computationally more efficient and may lead to faster convergence.

Linear Regression II

Improving Linear Regression

Subset/Feature Selection: approach involves identifying a subset of the p predictors that we believe to be best related to the response. Then we fit model using the reduced set of variables.

- Best, Forward, and Backward Subset Selection

Shrinkage/Regularization: all variables are used, but estimated coefficients are shrunken towards zero relative to the least squares estimate. λ represents the tuning parameter- as λ increases, flexibility decreases $\rightarrow$ decreased variance but increased bias. The tuning parameter is key in determining the sweet spot between under and over-fitting. In addition, while Ridge will always produce a model with p variables, Lasso can force coefficients to be equal to zero.

- Lasso (L1): min RSS + $\lambda \sum_{j=1}^{p} |\beta_j|$
- Ridge (L2): min RSS + $\lambda \sum_{j=1}^{p} \beta_j^2$

Dimension Reduction: projecting p predictors into a M-dimensional subspace, where M < p. This is achieved by computing M different linear combinations of the variables. Can use PCA.

Miscellaneous: Removing outliers, feature scaling, removing multicollinearity (correlated variables)

Evaluating Model Accuracy

Residual Standard Error (RSE): $\text{RSE} = \sqrt{\frac{1}{n-2} RSS}$. Generally, the smaller the better.

R^2: Measure of fit that represents the proportion of variance explained, or the *variability in Y that can be explained using X*. It takes on a value between 0 and 1. Generally the higher the better. $R^2 = 1 - \frac{RSS}{TSS}$, where Total Sum of Squares (TSS) $= \sum (y_i - \bar{y})^2$

Evaluating Coefficient Estimates

Standard Error (SE) of the coefficients can be used to perform hypothesis tests on the coefficients:

H_0: No relationship between X and Y, H_a: Some relationship exists. A p-value can be obtained and can be interpreted as follows: a small p-value indicates that a relationship between the predictor (X) and the response (Y) exists. Typical p-value cutoffs are around 5 or 1 %.

Logistic Regression

Logistic regression is used for classification, where the response variable is categorical rather than numerical.

The model works by predicting the probability that Y belongs to a particular category by first fitting the data to a linear regression model, which is then passed to the logistic function (below). The logistic function will always produce a S-shaped curve, so regardless of X, we can always obtain a sensible answer (between 0 and 1). If the probability is above a certain predetermined threshold (e.g. P(Yes) > 0.5), then the model will predict Yes.

$$p(X) = \frac{e^{\beta_0 + \beta_1 X_1 + ... + \beta_p X_p}}{1 + e^{\beta_0 + \beta_1 X_1 + ... + \beta_p X_p}}$$

How to find best coefficients?

Maximum Likelihood: The coefficients $\beta_0 ... \beta_p$ are unknown and must be estimated from the training data. We seek estimates for $\beta_0 ... \beta_p$ such that the predicted probability $\hat{p}(x_i)$ of each observation is a number close to one if its observed in a certain class and close to zero otherwise. This is done by maximizing the likelihood function:

$$l(\beta_0, \beta_1) = \prod_{i:y_i=1} p(x_i) \prod_{i':y_{i'}=1} (1 - p(x_i))$$

Potential Issues

Imbalanced Classes: imbalance in classes in training data lead to poor classifiers. It can result in a lot of false positives and also lead to few training data. Solutions include forcing balanced data by removing observations from the larger class, replicate data from the smaller class, or heavily weigh the training examples toward instances of the larger class.

Multi-Class Classification: the more classes you try to predict, the harder it will be for the the classifier to be effective. It is possible with logistic regression, but another approach, such as Linear Discriminant Analysis (LDA), may prove better.

Distance/Network Methods

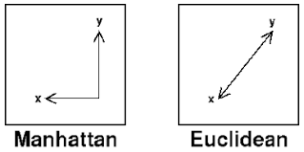
Interpreting examples as points in space provides a way to find natural groupings or clusters among data e.g. which stars are the closest to our sun? Networks can also be built from point sets (vertices) by connecting related points.

Measuring Distances/Similarity Measure

There are several ways of measuring distances between points a and b in d dimensions- with closer distances implying similarity.

Minkowski Distance Metric: $d_k(a,b) = \sqrt[k]{\sum_{i=1}^{d} |a_i - b_i|^k}$
The parameter k provides a way to tradeoff between the largest and the total dimensional difference. In other words, larger values of k place more emphasis on large differences between feature values than smaller values. Selecting the right k can significantly impact the the meaningfulness of your distance function. The most popular values are 1 and 2.

- Manhattan (k=1): city block distance, or the sum of the absolute difference between two points
- Euclidean (k=2): straight line distance



Weighted Minkowski: $d_k(a,b) = \sqrt[k]{\sum_{i=1}^{d} w_i |a_i - b_i|^k}$, in some scenarios, not all dimensions are equal. Can convey this idea using w_i. Generally not a good idea- should normalize data by Z-scores before computing distances.

Cosine Similarity: $cos(a,b) = \frac{a \cdot b}{|a||b|}$, calculates the similarity between 2 non-zero vectors, where $a \cdot b$ is the dot product (normalized between 0 and 1), higher values imply more similar vectors

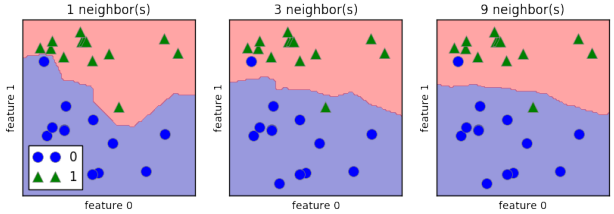
Kullback-Leibler Divergence: $KL(A||B) = \sum_{i=i}^{d} a_i log_2 \frac{a_i}{b_i}$
KL divergence measures the distances between probability distributions by measuring the uncertainty gained or uncertainty lost when replacing distribution A with distribution B. However, this is not a metric but forms the basis for the Jensen-Shannon Divergence Metric.
Jensen-Shannon: $JS(A,B) = \frac{1}{2}KL(A||M) + \frac{1}{2}KL(M||B)$, where M is the average of A and B. The JS function is the right metric for calculating distances between probability distributions

Nearest Neighbor Classification

Distance functions allow us to identify the points closest to a given target, or the *nearest neighbors (NN)* to a given point. The advantages of NN include simplicity, interpretability and non-linearity.

k-Nearest Neighbors

Given a positive integer k and a point x_0, the KNN classifier first identifies k points in the training data most similar to x_0, then estimates the conditional probability of x_0 being in class j as the fraction of the k points whose values belong to j. The optimal value for k can be found using cross validation.



KNN Algorithm

1. Compute distance D(a,b) from point b to all points
2. Select k closest points and their labels
3. Output class with most frequent labels in k points

Optimizing KNN

Comparing a query point a in d dimensions against n training examples computes with a runtime of $O(nd)$, which can cause lag as points reach millions or billions. Popular choices to speed up KNN include:

- **Vernoi Diagrams**: partitioning plane into regions based on distance to points in a specific subset of the plane
- **Grid Indexes**: carve up space into d-dimensional boxes or grids and calculate the NN in the same cell as the point
- **Locality Sensitive Hashing (LSH)**: abandons the idea of finding the exact nearest neighbors. Instead, batch up nearby points to quickly find the most appropriate bucket B for our query point. LSH is defined by a hash function $h(p)$ that takes a point/vector as input and produces a number/ code as output, such that it is likely that h(a) = h(b) if a and b are close to each other, and h(a)!= h(b) if they are far apart.

Clustering

Clustering is the problem of grouping points by similarity using distance metrics, which ideally reflect the similarities you are looking for. Often items come from logical "sources" and clustering is a good way to reveal those origins. Perhaps the first thing to do with any data set. Possible applications include: hypothesis development, modeling over smaller subsets of data, data reduction, outlier detection.

K-Means Clustering

Simple and elegant algorithm to partition a dataset into K distinct, non-overlapping clusters.

1. Choose a K. Randomly assign a number between 1 and K to each observation. These serve as initial cluster assignments
2. Iterate until cluster assignments stop changing
   (a) For each of the K clusters, compute the cluster centroid. The kth cluster centroid is the vector of the p feature means for the observations in the kth cluster.
   (b) Assign each observation to the cluster whose centroid is closest (where closest is defined using distance metric).

Since the results of the algorithm depends on the initial random assignments, it is a good idea to repeat the algorithm from different random initializations to obtain the best overall results. Can use MSE to determine which cluster assignment is better.

Hierarchical Clustering

Alternative clustering algorithm that does not require us to commit to a particular K. Another advantage is that it results in a nice visualization called a **dendrogram**. Observations that fuse at bottom are similar, where those at the top are quite different- we draw conclusions based on the location on the vertical rather than horizontal axis.

1. Begin with n observations and a measure of all the $\frac{(n)n-1}{2}$ pairwise dissimilarities. Treat each observation as its own cluster.
2. For i = n, n-1, ...2
   (a) Examine all pairwise inter-cluster dissimilarities among the i clusters and identify the pair of clusters that are least dissimilar (most similar). Fuse these two clusters. The dissimilarity between these two clusters indicates height in dendrogram where fusion should be placed.
   (b) Assign each observation to the cluster whose centroid is closest (where closest is defined using distance metric).

Linkage: Complete (max dissimilarity), Single (min), Average, Centroid (between centroids of cluster A and B)

Machine Learning Part I

Comparing ML Algorithms

Power and Expressibility: ML methods differ in terms of complexity. Linear regression fits linear functions while NN define piecewise-linear separation boundaries. More complex models can provide more accurate models, but at the risk of overfitting.

Interpretability: some models are more transparent and understandable than others (white box vs. black box models)

Ease of Use: some models feature few parameters/decisions (linear regression/NN), while others require more decision making to optimize (SVMs)

Training Speed: models differ in how fast they fit the necessary parameters

Prediction Speed: models differ in how fast they make predictions given a query

Method	Power of Expression	Ease of Interpretation	Ease of Use	Training Speed	Prediction Speed
Linear Regression	5	9	9	9	9
Nearest Neighbor	5	9	8	10	2
Naive Bayes	4	8	7	9	8
Decision Trees	8	8	7	7	9
Support Vector Machines	8	6	6	7	7
Boosting	9	6	6	6	6
Graphical Models	9	8	3	4	4
Deep Learning	10	3	4	3	7

Naive Bayes

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of independence between every pair of features.

Problem: Suppose we need to classify vector X = $x_1...x_n$ into m classes, $C_1...C_m$. We need to compute the probability of each possible class given X, so we can assign X the label of the class with highest probability. We can calculate a probability using the Bayes' Theorem:

$$P(C_i|X) = \frac{P(X|C_i)P(C_i)}{P(X)}$$

Where:

1. $P(C_i)$: the prior probability of belonging to class i
2. $P(X)$: normalizing constant, or probability of seeing the given input vector over all possible input vectors
3. $P(X|C_i)$: the conditional probability of seeing input vector X given we know the class is C_i

The prediction model will formally look like:

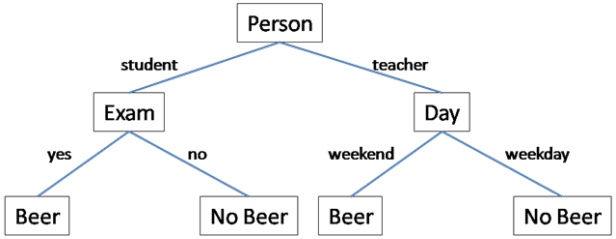
$$C(X) = argmax_{i \in classes(t)} \frac{P(X|C_i)P(C_i)}{P(X)}$$

where C(X) is the prediction returned for input X.

Machine Learning Part II

Decision Trees

Binary branching structure used to classify an arbitrary input vector X. Each node in the tree contains a simple feature comparison against some field ($x_i > 42$?). Result of each comparison is either true or false, which determines if we should proceed along to the left or right child of the given node. Also known as sometimes called classification and regression trees (CART).



Advantages: Non-linearity, support for categorical variables, easy to interpret, application to regression.

Disadvantages: Prone to overfitting, instable (not robust to noise), high variance, low bias

Note: rarely do models just use one decision tree. Instead, we aggregate many decision trees using methods like ensembling, bagging, and boosting.

Ensembles, Bagging, Random Forests, Boosting

Ensemble learning is the strategy of combining many different classifiers/models into one predictive model. It revolves around the idea of voting: a so-called "wisdom of crowds" approach. The most predicted class will be the final prediction.

Bagging: ensemble method that works by taking B bootstrapped subsamples of the training data and constructing B trees, each tree training on a distinct subsample as

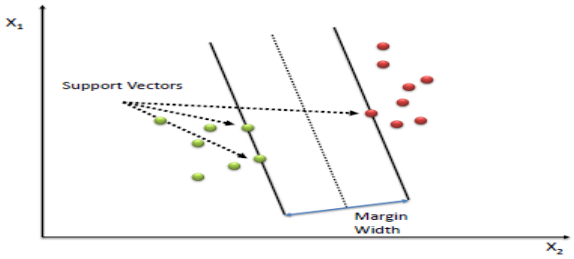
Random Forests: builds on bagging by decorrelating the trees. We do everything the same like in bagging, but when we build the trees, everytime we consider a split, a random sample of the p predictors is chosen as split candidates, not the full set (typically m $\approx \sqrt{p}$). When m = p, then we are just doing bagging.

Boosting: the main idea is to improve our model where it is not performing well by using information from previously constructed classifiers. Slow learner. Has 3 tuning parameters: number of classifiers B, learning parameter λ, interaction depth d (controls interaction order of model).

Machine Learning Part III

Support Vector Machines

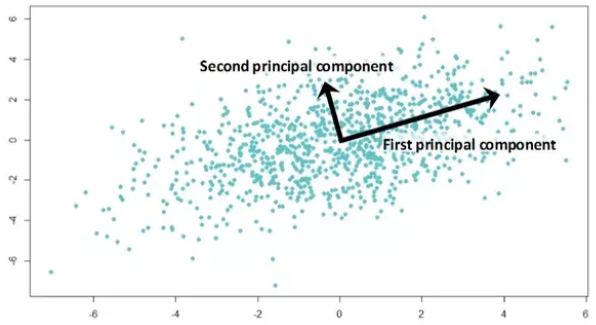
Work by constructing a hyperplane that separates points between two classes. The hyperplane is determined using the maximal margin hyperplane, which is the hyperplane that is the maximum distance from the training observations. This distance is called the margin. Points that fall on one side of the hyperplane are classified as -1 and the other +1.



Principal Component Analysis (PCA)

Principal components allow us to summarize a set of correlated variables with a smaller set of variables that collectively explain most of the variability in the original set. Essentially, we are "dropping" the least important feature variables.

Principal Component Analysis is the process by which principal components are calculated and the use of them to analyzing and understanding the data. PCA is an unsupervised approach and is used for dimensionality reduction, feature extraction, and data visualization. Variables after performing PCA are independent. Scaling variables is also important while performing PCA.



Machine Learning Part IV

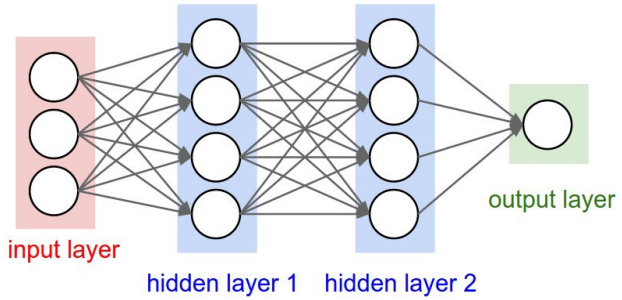
ML Terminology and Concepts

Features: input data/variables used by the ML model
Feature Engineering: transforming input features to be more useful for the models. e.g. mapping categories to buckets, normalizing between -1 and 1, removing null
Train/Eval/Test: training is data used to optimize the model, evaluation is used to asses the model on new data during training, test is used to provide the final result
Classification/Regression: regression is prediction a number (e.g. housing price), classification is prediction from a set of categories(e.g. predicting red/blue/green)
Linear Regression: predicts an output by multiplying and summing input features with weights and biases
Logistic Regression: similar to linear regression but predicts a probability
Overfitting: model performs great on the input data but poorly on the test data (combat by dropout, early stopping, or reduce # of nodes or layers)
Bias/Variance: how much output is determined by the features. more variance often can mean overfitting, more bias can mean a bad model
Regularization: variety of approaches to reduce overfitting, including adding the weights to the loss function, randomly dropping layers (dropout)
Ensemble Learning: training multiple models with different parameters to solve the same problem
A/B testing: statistical way of comparing 2+ techniques to determine which technique performs better and also if difference is statistically significant
Baseline Model: simple model/heuristic used as reference point for comparing how well a model is performing
Bias: prejudice or favoritism towards some things, people, or groups over others that can affect collection/sampling and interpretation of data, the design of a system, and how users interact with a system
Dynamic Model: model that is trained online in a continuously updating fashion
Static Model: model that is trained offline
Normalization: process of converting an actual range of values into a standard range of values, typically -1 to +1
Independently and Identically Distributed (i.i.d): data drawn from a distribution that doesn't change, and where each value drawn doesn't depend on previously drawn values; ideal but rarely found in real life
Hyperparameters: the "knobs" that you tweak during successive runs of training a model
Generalization: refers to a model's ability to make correct predictions on new, previously unseen data as opposed to the data used to train the model
Cross-Entropy: quantifies the difference between two probability distributions

Deep Learning Part I

What is Deep Learning?

Deep learning is a subset of machine learning. One popular DL technique is based on Neural Networks (NN), which loosely mimic the human brain and the code structures are arranged in layers. Each layer's input is the previous layer's output, which yields progressively higher-level features and defines a hierarchy. A Deep Neural Network is just a NN that has more than 1 hidden layer.



Recall that statistical learning is all about approximating $f(X)$. Neural networks are known as **universal approximators**, meaning no matter how complex a function is, there exists a NN that can (approximately) do the job. We can increase the approximation (or complexity) by adding more hidden layers and neurons.

Popular Architectures

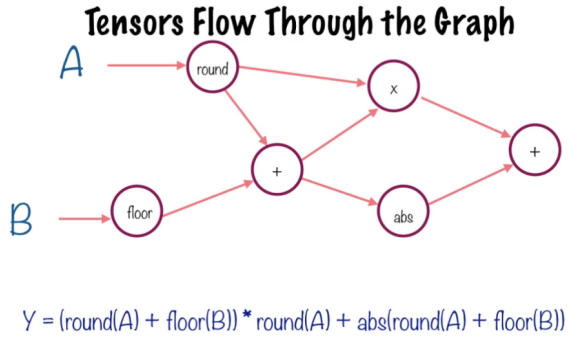
There are different kinds of NNs that are suitable for certain problems, which depend on the NN's architecture.

Linear Classifier: takes input features and combines them with weights and biases to predict output value
DNN: deep neural net, contains intermediate layers of nodes that represent "hidden features" and activation functions to represent non-linearity
CNN: convolutional NN, has a combination of convolutional, pooling, dense layers. popular for image classification.
Transfer Learning: use existing trained models as starting points and add additional layers for the specific use case. idea is that highly trained existing models know general features that serve as a good starting point for training a small network on specific examples
RNN: recurrent NN, designed for handling a sequence of inputs that have "memory" of the sequence. LSTMs are a fancy version of RNNs, popular for NLP
GAN: general adversarial NN, one model creates fake examples, and another model is served both fake example and real examples and is asked to distinguish
Wide and Deep: combines linear classifiers with deep neural net classifiers, "wide" linear parts represent memorizing specific examples and "deep" parts represent understanding high level features

Deep Learning Part II

Tensorflow

Tensorflow is an open source software library for numerical computation using data flow graphs. Everything in TF is a graph, where nodes represent operations on data and edges represent the data. Phase 1 of TF is building up a computation graph and phase 2 is executing it. It is also distributed, meaning it can run on either a cluster of machines or just a single machine.
TF is extremely popular/suitable for working with Neural Networks, since the way TF sets up the computational graph pretty much resembles a NN.



Tensors

In a graph, tensors are the edges and are multidimensional data arrays that flow through the graph. Central unit of data in TF and consists of a set of primitive values shaped into an array of any number of dimensions.
A tensor is characterized by its rank (# dimensions in tensor), shape (# of dimensions and size of each dimension), data type (data type of each element in tensor).

Placeholders and Variables

Variables: best way to represent shared, persistent state manipulated by your program. These are the parameters of the ML model are altered/trained during the training process. Training variables.
Placeholders: way to specify inputs into a graph that hold the place for a Tensor that will be fed at runtime. They are assigned once, do not change after. Input nodes

Deep Learning Part III

Deep Learning Terminology and Concepts

Neuron: node in a NN, typically taking in multiple input values and generating one output value, calculates the output value by applying an activation function (nonlinear transformation) to a weighted sum of input values
Weights: edges in a NN, the goal of training is to determine the optimal weight for each feature; if weight = 0, corresponding feature does not contribute
Neural Network: composed of neurons (simple building blocks that actually "learn"), contains activation functions that makes it possible to predict non-linear outputs
Activation Functions: mathematical functions that introduce non-linearity to a network e.g. RELU, tanh
Sigmoid Function: function that maps very negative numbers to a number very close to 0, huge numbers close to 1, and 0 to .5. Useful for predicting probabilities
Gradient Descent/Backpropagation: fundamental loss optimizer algorithms, of which the other optimizers are usually based. Backpropagation is similar to gradient descent but for neural nets
Optimizer: operation that changes the weights and biases to reduce loss e.g. Adagrad or Adam
Weights / Biases: weights are values that the input features are multiplied by to predict an output value. Biases are the value of the output given a weight of 0.
Converge: algorithm that converges will eventually reach an optimal answer, even if very slowly. An algorithm that doesn't converge may never reach an optimal answer.
Learning Rate: rate at which optimizers change weights and biases. High learning rate generally trains faster but risks not converging, whereas a lower rate trains slower
Numerical Instability: issues with very large/small values due to limits of floating point numbers in computers
Embeddings: mapping from discrete objects, such as words, to vectors of real numbers. useful because classifiers/neural networks work well on vectors of real numbers
Convolutional Layer: series of convolutional operations, each acting on a different slice of the input matrix
Dropout: method for regularization in training NNs, works by removing a random selection of some units in a network layer for a single gradient step
Early Stopping: method for regularization that involves ending model training early
Gradient Descent: technique to minimize loss by computing the gradients of loss with respect to the model's parameters, conditioned on training data
Pooling: Reducing a matrix (or matrices) created by an earlier convolutional layer to a smaller matrix. Pooling usually involves taking either the maximum or average value across the pooled area

Big Data- Hadoop Overview

Data can no longer fit in memory on one machine (monolithic), so a new way of computing was devised using a group of computers to process this "big data" (distributed). Such a group is called a cluster, which makes up server farms. All of these servers have to be coordinated in the following ways: partition data, coordinate computing tasks, handle fault tolerance/recovery, and allocate capacity to process.

Hadoop

Hadoop is an open source *distributed* processing framework that manages data processing and storage for big data applications running in clustered systems. It is comprised of 3 main components:

- **Hadoop Distributed File System (HDFS)**: a distributed file system that provides high-throughput access to application data by partitioning data across many machines
- **YARN**: framework for job scheduling and cluster resource management (task coordination)
- **MapReduce**: YARN-based system for parallel processing of large data sets on multiple machines

HDFS

Each disk on a different machine in a cluster is comprised of 1 master node and the rest are workers/data nodes.; The **master node** manages the overall file system by storing the directory structure and the metadata of the files. The **data nodes** physically store the data. Large files are broken up and distributed across multiple machines, which are also replicated across multiple machines to provide fault tolerance.

MapReduce

Parallel programming paradigm which allows for processing of huge amounts of data by running processes on multiple machines. Defining a MapReduce job requires two stages: map and reduce.

- **Map**: operation to be performed in parallel on small portions of the dataset. the output is a key-value pair $< K, V >$
- **Reduce**: operation to combine the results of Map

YARN- Yet Another Resource Negotiator

Coordinates tasks running on the cluster and assigns new nodes in case of failure. Comprised of 2 subcomponents: the resource manager and the node manager. The **resource manager** runs on a single master node and schedules tasks across nodes. The **node manager** runs on all other nodes and manages tasks on the individual node.

Big Data- Hadoop Ecosystem

An entire ecosystem of tools have emerged around Hadoop, which are based on interacting with HDFS. Below are some popular ones:

Hive: data warehouse software built o top of Hadoop that facilitates reading, writing, and managing large datasets residing in distributed storage using SQL-like queries (HiveQL). Hive abstracts away underlying MapReduce jobs and returns HDFS in the form of tables (not HDFS).
Pig: high level scripting language (Pig Latin) that enables writing complex data transformations. It pulls unstructured/incomplete data from sources, cleans it, and places it in a database/data warehouses. Pig performs ETL into data warehouse while Hive queries from data warehouse to perform analysis (GCP: DataFlow).
Spark: framework for writing fast, distributed programs for data processing and analysis. Spark solves similar problems as Hadoop MapReduce but with a fast in-memory approach. It is an unified engine that supports SQL queries, streaming data, machine learning and graph processing. Can operate separately from Hadoop but integrates well with Hadoop. Data is processed using Resilient Distributed Datasets (RDDs), which are immutable, lazily evaluated, and tracks lineage. ;
Hbase: non-relational, NoSQL, column-oriented database management system that runs on top of HDFS. Well suited for sparse data sets (GCP: BigTable)
Flink/Kafka: stream processing framework. Batch streaming is for bounded, finite datasets, with periodic updates, and delayed processing. Stream processing is for unbounded datasets, with continuous updates, and immediate processing. Stream data and stream processing must be decoupled via a message queue. Can group streaming data (windows) using tumbling (non-overlapping time), sliding (overlapping time), or session (session gap) windows.
Beam: programming model to define and execute data processing pipelines, including ETL, batch and stream (continuous) processing. After building the pipeline, it is executed by one of Beam's distributed processing back-ends (Apache Apex, Apache Flink, Apache Spark, and Google Cloud Dataflow). Modeled as a Directed Acyclic Graph (DAG).
Oozie: workflow scheduler system to manage Hadoop jobs
Sqoop: transferring framework to transfer large amounts of data into HDFS from relational databases (MySQL)

SQL Part I

Structured Query Language (SQL) is a declarative language used to access & manipulate data in databases. Usually the database is a Relational Database Management System (RDBMS), which stores data arranged in relational database tables. A table is arranged in columns and rows, where columns represent characteristics of stored data and rows represent actual data entries.

Basic Queries

- filter columns: **SELECT** col1, col3... **FROM** table1
- filter the rows: **WHERE** col4 = 1 **AND** col5 = 2
- aggregate the data: **GROUP BY**...
- limit aggregated data: **HAVING** count(*) > 1
- order of the results: **ORDER BY** col2

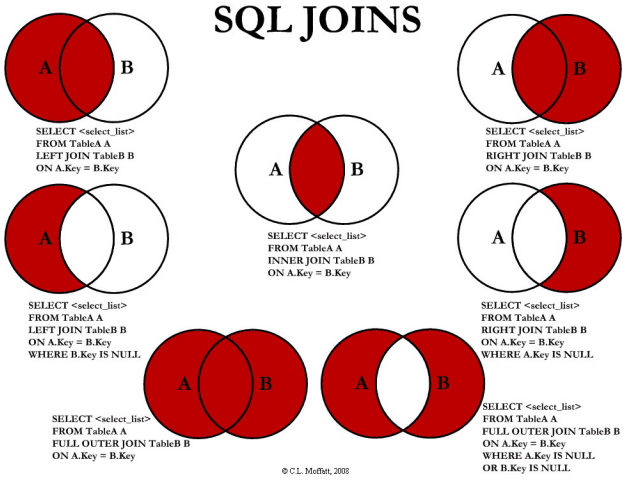
Useful Keywords for **SELECT**
DISTINCT- return unique results
BETWEEN a AND b- limit the range, the values can be numbers, text, or dates
LIKE- pattern search within the column text
IN (a, b, c) - check if the value is contained among given

Data Modification

- update specific data with the **WHERE** clause:
UPDATE table1 **SET** col1 = 1 **WHERE** col2 = 2
- insert values manually
INSERT INTO table1 (col1,col3) **VALUES** (val1,val3);
- by using the results of a query
INSERT INTO table1 (col1,col3) **SELECT** col,col2 FROM table2;

Joins

The JOIN clause is used to combine rows from two or more tables, based on a related column between them.



Python- Data Structures

Data structures are a way of storing and manipulating data and each data structure has its own strengths and weaknesses. Combined with algorithms, data structures allow us to efficiently solve problems. It is important to know the main types of data structures that you will need to efficiently solve problems.

Lists: or arrays, ordered sequences of objects, mutable

```
>>> l = [42, 3.14, "hello","world"]
```

Tuples: like lists, but immutable

```
>>> t = (42, 3.14, "hello","world")
```

Dictionaries: hash tables, key-value pairs, unsorted

```
>>> d = {"life": 42, "pi": 3.14}
```

Sets: mutable, unordered sequence of unique elements. frozensets are just immutable sets

```
>>> s = set([42, 3.14, "hello","world"])
```

Collections Module

deque: double-ended queue, generalization of stacks and queues; supports append, appendLeft, pop, rotate, etc

```
>>> s = deque([42, 3.14, "hello","world"])
```

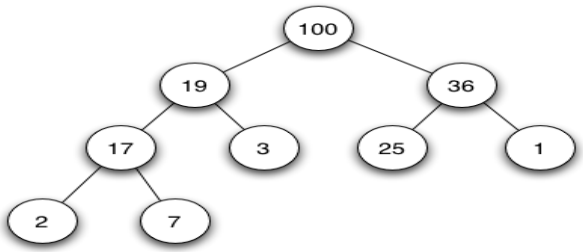
Counter: dict subclass, unordered collection where elements are stored as keys and counts stored as values

```
>>> c = Counter('apple')
>>> print(c)
Counter({'p': 2, 'a': 1, 'l': 1, 'e': 1})
```

heqpq Module

Heap Queue: priority queue, heaps are binary trees for which every parent node has a value greater than or equal to any of its children (max-heap), order is important; supports push, pop, pushpop, heapify, replace functionality

```
>>> heap = []
>>> for n in data:
...     heappush(heap, n)
>>> heap
[0, 1, 3, 6, 2, 8, 4, 7, 9, 5]
```



Recommended Resources

- Data Science Design Manual (`www.springer.com/us/book/9783319554433`)
- Introduction to Statistical Learning (`www-bcf.usc.edu/~gareth/ISL/`)
- Probability Cheatsheet (`/www.wzchen.com/probability-cheatsheet/`)
- Google's Machine Learning Crash Course (`developers.google.com/machine-learning/crash-course/`)

fournova

GIT CHEAT SHEET

presented by **TOWER** › Version control with Git - made easy

CREATE

Clone an existing repository

```
$ git clone ssh://user@domain.com/repo.git
```

Create a new local repository

```
$ git init
```

LOCAL CHANGES

Changed files in your working directory

```
$ git status
```

Changes to tracked files

```
$ git diff
```

Add all current changes to the next commit

```
$ git add .
```

Add some changes in <file> to the next commit

```
$ git add -p <file>
```

Commit all local changes in tracked files

```
$ git commit -a
```

Commit previously staged changes

```
$ git commit
```

Change the last commit

Don‘t amend published commits!

```
$ git commit --amend
```

COMMIT HISTORY

Show all commits, starting with newest

```
$ git log
```

Show changes over time for a specific file

```
$ git log -p <file>
```

Who changed what and when in <file>

```
$ git blame <file>
```

BRANCHES & TAGS

List all existing branches

```
$ git branch -av
```

Switch HEAD branch

```
$ git checkout <branch>
```

Create a new branch based on your current HEAD

```
$ git branch <new-branch>
```

Create a new tracking branch based on a remote branch

```
$ git checkout --track <remote/branch>
```

Delete a local branch

```
$ git branch -d <branch>
```

Mark the current commit with a tag

```
$ git tag <tag-name>
```

UPDATE & PUBLISH

List all currently configured remotes

```
$ git remote -v
```

Show information about a remote

```
$ git remote show <remote>
```

Add new remote repository, named <remote>

```
$ git remote add <shortname> <url>
```

Download all changes from <remote>, but don‘t integrate into HEAD

```
$ git fetch <remote>
```

Download changes and directly merge/integrate into HEAD

```
$ git pull <remote> <branch>
```

Publish local changes on a remote

```
$ git push <remote> <branch>
```

Delete a branch on the remote

```
$ git branch -dr <remote/branch>
```

Publish your tags

```
$ git push --tags
```

MERGE & REBASE

Merge <branch> into your current HEAD

```
$ git merge <branch>
```

Rebase your current HEAD onto <branch>

Don‘t rebase published commits!

```
$ git rebase <branch>
```

Abort a rebase

```
$ git rebase --abort
```

Continue a rebase after resolving conflicts

```
$ git rebase --continue
```

Use your configured merge tool to solve conflicts

```
$ git mergetool
```

Use your editor to manually solve conflicts and (after resolving) mark file as resolved

```
$ git add <resolved-file>
$ git rm <resolved-file>
```

UNDO

Discard all local changes in your working directory

```
$ git reset --hard HEAD
```

Discard local changes in a specific file

```
$ git checkout HEAD <file>
```

Revert a commit (by producing a new commit with contrary changes)

```
$ git revert <commit>
```

Reset your HEAD pointer to a previous commit

...and discard all changes since then

```
$ git reset --hard <commit>
```

...and preserve all changes as unstaged changes

```
$ git reset <commit>
```

...and preserve uncommitted local changes

```
$ git reset --keep <commit>
```

fournova

VERSION CONTROL

BEST PRACTICES

COMMIT RELATED CHANGES

A commit should be a wrapper for related changes. For example, fixing two different bugs should produce two separate commits. Small commits make it easier for other developers to understand the changes and roll them back if something went wrong.
With tools like the staging area and the ability to stage only parts of a file, Git makes it easy to create very granular commits.

COMMIT OFTEN

Committing often keeps your commits small and, again, helps you commit only related changes. Moreover, it allows you to share your code more frequently with others. That way it's easier for everyone to integrate changes regularly and avoid having merge conflicts. Having few large commits and sharing them rarely, in contrast, makes it hard to solve conflicts.

DON'T COMMIT HALF-DONE WORK

You should only commit code when it's completed. This doesn't mean you have to complete a whole, large feature before committing. Quite the contrary: split the feature's implementation into logical chunks and remember to commit early and often. But don't commit just to have something in the repository before leaving the office at the end of the day. If you're tempted to commit just because you need a clean working copy (to check out a branch, pull in changes, etc.) consider using Git's «Stash» feature instead.

TEST CODE BEFORE YOU COMMIT

Resist the temptation to commit something that you «think» is completed. Test it thoroughly to make sure it really is completed and has no side effects (as far as one can tell). While committing half-baked things in your local repository only requires you to forgive yourself, having your code tested is even more important when it comes to pushing/sharing your code with others.

WRITE GOOD COMMIT MESSAGES

Begin your message with a short summary of your changes (up to 50 characters as a guideline). Separate it from the following body by including a blank line. The body of your message should provide detailed answers to the following questions:

› What was the motivation for the change?

› How does it differ from the previous implementation?

Use the imperative, present tense («change», not «changed» or «changes») to be consistent with generated messages from commands like git merge.

VERSION CONTROL IS NOT A BACKUP SYSTEM

Having your files backed up on a remote server is a nice side effect of having a version control system. But you should not use your VCS like it was a backup system. When doing version control, you should pay attention to committing semantically (see «related changes») - you shouldn't just cram in files.

USE BRANCHES

Branching is one of Git's most powerful features - and this is not by accident: quick and easy branching was a central requirement from day one. Branches are the perfect tool to help you avoid mixing up different lines of development. You should use branches extensively in your development workflows: for new features, bug fixes, ideas...

AGREE ON A WORKFLOW

Git lets you pick from a lot of different workflows: long-running branches, topic branches, merge or rebase, git-flow... Which one you choose depends on a couple of factors: your project, your overall development and deployment workflows and (maybe most importantly) on your and your teammates' personal preferences. However you choose to work, just make sure to agree on a common workflow that everyone follows.

HELP & DOCUMENTATION

Get help on the command line

```
$ git help <command>
```

FREE ONLINE RESOURCES

http://www.git-tower.com/learn

http://rogerdudler.github.io/git-guide/

http://www.git-scm.org/

MACHINE LEARNING CHEATSHEET

Summary of Machine Learning Algorithms descriptions, advantages and use cases. Inspired by the very good book and articles of *MachineLearningMastery,* with added math, and *ML Pros & Cons* of *HackingNote.* Design inspired by *The Probability Cheatsheet* of W. Chen. Written by Rémi Canard.

General

Definition

We want to learn a target function ***f*** that maps input variables ***X*** to output variable ***Y***, with an error ***e***:

$$Y = f(X) + e$$

Linear, Nonlinear

Different algorithms make different assumptions about the shape and structure of *f*, thus the need of testing several methods. Any algorithm can be either:

- **Parametric** (or **Linear**): simplify the mapping to a known linear combination form and learning its coefficients.

- **Non parametric** (or **Nonlinear**): free to learn any functional form from the training data, while maintaining some ability to generalize.

Linear algorithms are usually simpler, faster and requires less data, while Nonlinear can be are more flexible, more powerful and more performant.

Supervised, Unsupervised

Supervised learning methods learn to predict Y from X given that the data is labeled.

Unsupervised learning methods learn to find the inherent structure of the unlabeled data.

Bias-Variance trade-off

In supervised learning, the prediction error ***e*** is composed of the **bias**, the **variance** and the **irreducible** part.

Bias refers to **simplifying assumptions** made to learn the target function easily.

Variance refers to sensitivity of the model to changes in the training data.

The **goal of parameterization** is to achieve a **low bias** (underlying pattern not too simplified) and **low variance** (not sensitive to specificities of the training data) **tradeoff**.

Underfitting, Overfitting

In statistics, ***fit*** refers to how well the target function is approximated.

Underfitting refers to poor inductive learning from training data and poor generalization.

Overfitting refers to learning the training data detail and noise which leads to poor generalization. It can be **limited** by using resampling and defining a validation dataset.

Optimization

Almost every machine learning method has an optimization algorithm at its core.

Gradient Descent

Gradient Descent is used to **find the coefficients** of f that **minimizes a cost function** (for example MSE, SSR).

Procedure:

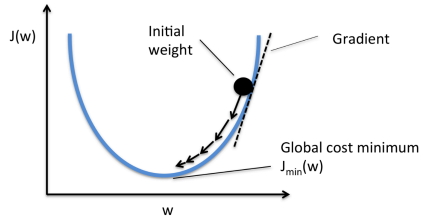
→ Initialization $\theta = 0$ (coefficients to 0 or random)

→ Calculate cost $J(\theta) = evaluate(f(coefficients))$

→ Gradient of cost $\frac{\partial}{\partial \theta j} J(\theta)$ we know the uphill direction

→ Update coeff $\theta j = \theta j - \alpha \frac{\partial}{\partial \theta j} J(\theta)$ we go downhill

The cost updating process is repeated until convergence (minimum found).



Batch Gradient Descend does summing/averaging of the cost over all the observations.

Stochastic Gradient Descent apply the procedure of parameter updating for each observation.

Tips:

- Change **learning rate** α ("size of jump" at each iteration)

- Plot *Cost vs Time* to assess learning rate performance

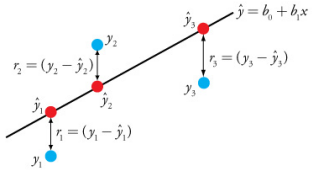
- Rescaling the input variables

- Reduce passes through training set with SGD

- Average over 10 or more updated to observe the learning trend while using SGD

Ordinary Least Squares

OLS is used to find the estimator $\hat{\beta}$ that **minimizes the sum of squared residuals:** $\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij})^2 = y - X\hat{\beta}$



Using linear algebra such that we have $\hat{\beta} = (X^T X)^{-1} X^T y$

Maximum Likelihood Estimation

MLE is used to find the estimators that **minimizes the likelihood function:**

$\mathcal{L}(\theta|x) = f_\theta(x)$ density function of the data distribution

Linear Algorithms

All linear Algorithms assume a linear relationship between the input variables *X* and the output variable *Y*.

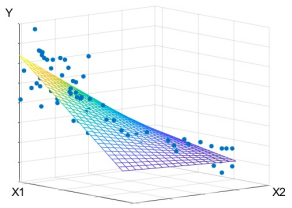
Linear Regression

Representation:

A LR model representation is a linear equation:

$$y = \beta_0 + \beta_1 x_1 + \cdots + \beta_i x_i$$

β_0 is usually called intercept or **bias** coefficient. The dimension of the hyperplane of the regression is its **complexity.**



Learning:

Learning a LR means estimating the coefficients from the training data. Common methods include **Gradient Descent** or **Ordinary Least Squares**.

Variations:

There are extensions of LR training called **regularization** methods, that aim to **reduce the complexity** of the model:

- **Lasso Regression**: where OLS is modified to minimize the sum of the coefficients (L1 regularization)

$$\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij})^2 + \lambda\sum_{j=1}^{p}|\beta_j| = RSS + \lambda\sum_{j=1}^{p}|\beta_j|$$

- **Ridge Regression**: where OLS is modified to minimize the squared sum of the coefficients (L2 regularization)

$$\sum_{i=1}^{n}(y_i - \beta_0 - \sum_{j=1}^{p}\beta_j x_{ij})^2 + \lambda\sum_{j=1}^{p}{\beta_j}^2 = RSS + \lambda\sum_{j=1}^{p}{\beta_j}^2$$

where $\lambda \geq 0$ is a tuning parameter to be determined.

Data preparation:

- Transform data for linear relationship (ex: log transform for exponential relationship)
- Remove noise such as outliers
- Rescale inputs using standardization or normalization

Advantages:

+ Good regression baseline considering simplicity
+ Lasso/Ridge can be used to avoid overfitting
+ Lasso/Ridge permit feature selection in case of collinearity

Usecase examples:

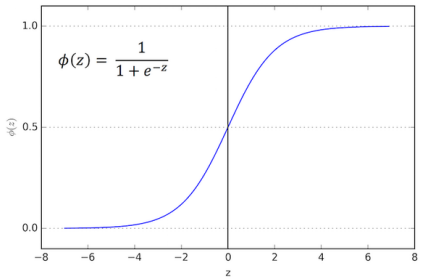
- Product sales prediction according to prices or promotions
- Call-center waiting-time prediction according to the number of complaints and the number of working agents

Logistic Regression

It is the go-to for **binary classification.**

Representation:

Logistic regression a linear method but predictions are transformed using the **logistic function** (or sigmoid):



ϕ is *S*-shaped and map real-valued number in (0,1).

The representation is an equation with binary output:

$$y = \frac{e^{\beta_0+\beta_1 x_1+\cdots+\beta_i x_i}}{1 + e^{\beta_0+\beta_1 x_1+\cdots+\beta_i x_i}}$$

Which actually models the probability of default class:

$$p(X) = \frac{e^{\beta_0+\beta_1 x_1+\cdots+\beta_i x_i}}{1 + e^{\beta_0+\beta_1 x_1+\cdots+\beta_i x_i}} = p(Y = 1|X)$$

Learning:

Learning the Logistic regression coefficients is done using **maximum-likelihood estimation**, to predict values close to 1 for default class and close to 0 for the other class.

Data preparation:

- Probability transformation to binary for classification
- Remove noise such as outliers

Advantages:

+ Good classification baseline considering simplicity
+ Possibility to change cutoff for precision/recall tradeoff
+ Robust to noise/overfitting with L1/L2 regularization
+ Probability output can be used for ranking

Usecase examples:

- Customer scoring with probability of purchase
- Classification of loan defaults according to profile

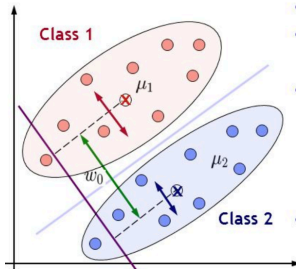
Linear Discriminant Analysis

For **multiclass classification**, LDA is the preferred linear technique.

Representation:

LDA representation consists of **statistical properties** calculated for **each class**: **means** and the **covariance matrix:**

$$\mu_k = \frac{1}{n_k}\sum_{i=1}^{n} x_i \quad \text{and} \quad \sigma^2 = \frac{1}{n-K}\sum_{i=1}^{n}(x_i - \mu_k)^2$$



LDA assumes **Gaussian** data and attributes of **same σ^2**.

Predictions are made using **Bayes Theorem:**

$$P(Y = k|X = x) = \frac{P(k)\times P(x|k)}{\sum_{l=1}^{K} P(l)\times P(x|l)}$$

to obtain a discriminate function (latent variable) for each class *k*, estimating $P(x|k)$ with a Gaussian distribution:

$$D_k(x) = x \times \frac{\mu_k}{\sigma^2} - \frac{{\mu_k}^2}{2\sigma^2} + \ln(P(k))$$

The class with largest **discriminant value** is the **output class**.

Variations:

- **Quadratic DA**: Each class uses its own variance estimate
- **Regularized DA**: Regularization into the variance estimate

Data preparation:

- Review and modify univariate distributions to be Gaussian
- Standardize data to $\mu = 0, \sigma = 1$ to have same variance
- Remove noise such as outliers

Advantages:

+ Can be used for dimensionality reduction by keeping the latent variables as new variables

Usecase example:

- Prediction of customer churn

Nonlinear Algorithms

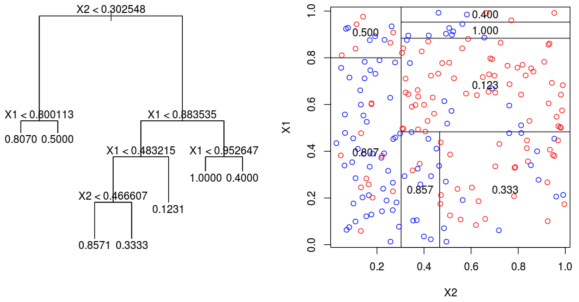
All Nonlinear Algorithms are non-parametric and more flexible. They are not sensible to outliers and do not require any shape of distribution.

Classification and Regression Trees

Also referred as CART or Decision Trees, this algorithm is the foundation of Random Forest and Boosted Trees.

Representation:

The model representation is a **binary tree**, where each **node** is an **input variable** *x* with a split point and each **leaf** contain an **output variable** *y* for prediction.



The model actually **split the input space** into (hyper) rectangles, and predictions are made according to the **area** observations *fall* into.

Learning:

Learning of a CART is done by a greedy approach called **recursive binary splitting** of the input space:

At each step, the best **predictor** X_j and the best **cutpoint** s are selected such that $\{X|X_j < s\}$ and $\{X|X_j \geq s\}$ **minimizes the cost**.

- For **regression** the cost is the **Sum of Squared Error:**

$$\sum_{i=1}^{n}(y_i - \hat{y})^2$$

- For **classification** the cost function is the **Gini index:**

$$G = \sum_{i=1}^{n} p_k(1 - p_k)$$

The Gini index is **an indication of how *pure* are the leaves**, if all observations are the same type G=0 (perfect purity), while a 50-50 split for binary would be G=0.5 (worst purity).

The most common **Stopping Criterion** for splitting is a minimum of **training observations per node**.

The simplest form of pruning is **Reduced Error Pruning:** Starting at the leaves, each node is replaced with its most popular class. If the prediction accuracy is not affected, then the change is kept

Advantages:

+ Easy to interpret and no overfitting with pruning

+ Works for both regression and classification problems

+ Can take any type of variables without modifications, and do not require any data preparation

Usecase examples:

- Fraudulent transaction classification

- Predict human resource allocation in companies

Naive Bayes Classifier

Naive Bayes is a **classification** algorithm interested in selecting the **best hypothesis *h* given data *d*** assuming there is no interaction between features.

Representation:

The representation is the based on Bayes Theorem:

$$P(h|d) = \frac{P(d|h) \times P(h)}{P(d)}$$

with naïve hypothesis $P(h|d) = P(x_1|h) \times \ldots \times P(x_i|h)$

The prediction is the **Maximum A posteriori Hypothesis**:

$$MAP(h) = \max\big(P(h|d)\big) = \max\left(P(d|h) \times P(h)\right)$$

The denominator is not kept as it is only for normalization.

Learning:

Training is **fast** because only **probabilities** need to be calculated:

$$P(h) = \frac{instances_h}{all\ instances} \text{ and } P(x|h) = \frac{count(x \wedge h)}{instances_h}$$

Variations:

Gaussian Naive Bayes can extend to numerical attributes by assuming a Gaussian distribution.

Instead of $P(x|h)$ are calculated with $P(h)$ during **learning**:

$$\mu(x) = \frac{1}{n}\sum_{i=1}^{n} x_i \text{ and } \sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu(x))^2}$$

and MAP for **prediction** is calculated using Gaussian PDF

$$f(x|\mu(x), \sigma) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Data preparation:

- Change numerical inputs to categorical (binning) or near-Gaussian inputs (remove outliers, log & boxcox transform)

- Other distributions can be used instead of Gaussian

- Log-transform of the probabilities can avoid overflow

- Probabilities can be updated as data becomes available

Advantages:

+ Fast because of the calculations

+ If the naive assumptions works can converge quicker than other models. Can be used on smaller training data.

+ Good for few categories variables

Usecase examples:

- Article classification using binary word presence

- Email spam detection using a similar technique

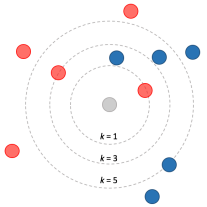
K-Nearest Neighbors

If you are similar to your neighbors, you are one of them.

Representation:

KNN uses the **entire training set**, **no training** is required.

Predictions are made by searching the ***k* similar instances**, according to a **distance**, and **summarizing the output**.



For **regression** the output can be the **mean**, while for **classification** the output can be the **most common class**.

Various distances can be used, for example:

- **Euclidean** Distance, good for **similar** type of variables

$$d(a,b) = \sqrt{\sum_{i=1}^{n}(a_i - b_i)^2}$$

- **Manhattan** Distance, good for **different** type of variables

$$d(a,b) = \sum_{i=1}^{n}|a_i - b_i|$$

The **best value of *k*** must be found by **testing**, and the algorithm is **sensible** to the ***Curse of dimensionality***.

Data preparation:

- Rescale inputs using standardization or normalization

- Address missing data for distance calculations

- Dimensionality reduction or feature selection for *COD*

Advantages:

+ Effective if the training data is large

+ No learning phase

+ Robust to noisy data, no need to filter outliers

Usecase examples:

- Recommending products based on similar customers

- Anomaly detection in customer behavior

Support Vector Machines

SVM is a go-to for high performance with little tuning

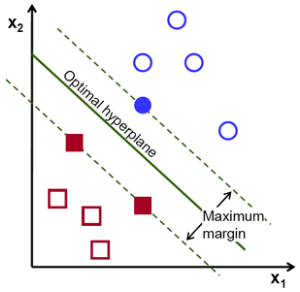
Representation:

In SVM, a **hyperplane** is selected to **separate the points** in the input variable space by their class, with the **largest margin**.

The closest datapoints (defining the margin) are called the **support vectors**.

But real **data cannot be perfectly separated**, that is why a ***C*** defines the amount of **violation** of the margin allowed.

The lower C, the more sensitive SVM is to training data.



The **prediction** function is the signed **distance** of the new input *x* to the separating hyperplane *w*:

$$f(x) = < w, x > + \rho = w^T x + \rho \quad \text{with } \rho \text{ the bias}$$

Which gives for **linear kernel**, with x_i the support vectors:

$$f(x) = \sum_{i=1}^{n} a_i \times (x \times x_i)) + \rho$$

Learning:

The hyperplane learning is done by transforming the problem using linear algebra, and minimizing:

$$\left[\frac{1}{n}\sum_{i=1}^{n}\max\left(0, 1 - y_i(\vec{w}.\vec{x_i} - b)\right)\right] + \lambda||\vec{w}||^2$$

Variations:

SVM is implemented using various kernels, which define the measure between new data and support vectors:

- **Linear** (dot-product): $K(x, x_i) = \sum(x \times x_i)$

- **Polynomial**: $K(x, x_i) = 1 + \sum(x \times x_i)^d$

- **Radial**: $K(x, x_i) = e^{-\gamma\sum((x - x_i)^2)}$

Data preparation:

- SVM assumes numeric inputs, may require dummy transformation of categorical features

Advantages:

+ Allow nonlinear separation with nonlinear Kernels

+ Works good in high dimensional space

+ Robust to multicollinearity and overfitting

Usecase examples:

- Face detection from images

- Target Audience Classification from tweets

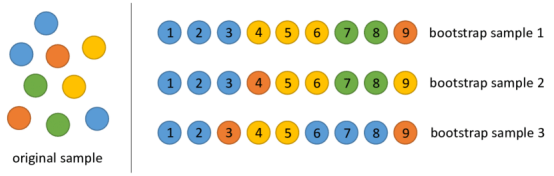
Ensemble Algorithms

Ensemble methods use multiple, simpler algorithms combined to obtain better performance.

Bagging and Random Forest

Random Forest is part of a bigger type of ensemble methods called **Bootstrap Aggregation** or **Bagging**. Bagging can **reduce the variance** of high-variance models.

It uses the **Bootstrap statistical procedure**: estimate a quantity from a sample by creating many random subsamples with replacement, and computing the mean of each subsample.



Representation:

For **bagged decision trees**, the steps would be:

- Create many subsamples of the training dataset

- Train a CART model on each sample

- Given a new dataset, calculate the average prediction

However, combining models works best if submodels are **weakly correlated** at best.

Random Forest is a tweaked version of bagged decision trees to **reduce tree correlation**.

Learning:

During learning, each **sub-tree** can only access a random **sample of features** when **selecting the split points**. The size of the feature sample at each split is a parameter *m*.

A good default is $\sqrt{p}$ for classification and $\frac{p}{3}$ for regression.

The **OOB** estimate is the performance of **each model on its Out-Of-Bag** (not selected) samples. It is a reliable estimate of test error.

Bagged method can provide **feature importance**, by calculating and averaging the **error function drop for individual variables** (depending of samples where a variable is selected or not).

Advantages:

In addition to the advantages of the CART algorithm

+ Robust to overfitting and missing variables

+ Can be parallelized for distributed computing

+ Performance as good as SVM but easier to interpret

Usecase examples:

- Predictive machine maintenance

- Optimizing line decision for credit cards

Boosting and AdaBoost

AdaBoost was the first successful boosting algorithm developed for binary classification.

Representation:

A boost classifier is of the form

$$F_T(x) = \sum_{t=1}^{T} f_t(x)$$

where each f_t is a **week learner correcting the errors** of the previous one.

Adaboost is commonly used with decision trees with one level (***decision stumps***).

Predictions are made using the weighted average of the weak classifiers.

Learning:

Each training set instance is initially weighted $w(x_i) = \frac{1}{n}$

One *decision stump* is prepared using the weighted samples, and a **misclassification rate** is calculated:

$$\epsilon = \frac{\sum_{i=1}^{n}(w_i \times p_{error\ i})}{\sum_{i=1}^{n} w}$$

Which is the weighted sum of the misclassification rates, where w is the training instance i weight and $p_{error\ i}$ its prediction error (1 or 0).

A **stage value** is computed from the misclassification rate:

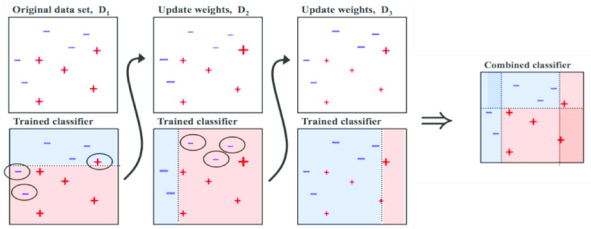
$$stage = \ln\left(\frac{1-\epsilon}{\epsilon}\right)$$

This stage value is used to **update the instances weights**:

$$w = w \times e^{stage \times \epsilon}$$

The **incorrectly** predicted instance are given **more** weight.

Weak models are added sequentially using the training weights, until no improvement can be made or the number of rounds has been attained.



Data preparation:

- Outliers should be removed for AdaBoost

Advantages:

+ High performance with no tuning (only number of rounds)

Interesting Resources

Machine Learning Mastery website

> https://machinelearningmastery.com/

Scikit-learn website, for python implementation

> http://scikit-learn.org/

W.Chen probability cheatsheet

> https://github.com/wzchen/probability_cheatsheet

HackingNote, for interesting, condensed insights

> https://www.hackingnote.com/

Seattle Data Guy blog, for business oriented articles

> https://www.theseattledataguy.com/

Explained visually, making hard ideas intuitive

> http://setosa.io/ev/

This Machine Learning Cheatsheet

> https://github.com/remicnrd/ml_cheatsheet

Beginner's Python Cheat Sheet

Variables and Strings

Variables are used to store values. A string is a series of characters, surrounded by single or double quotes.

Hello world

```
print("Hello world!")
```

Hello world with a variable

```
msg = "Hello world!"
print(msg)
```

Concatenation (combining strings)

```
first_name = 'albert'
last_name = 'einstein'
full_name = first_name + ' ' + last_name
print(full_name)
```

Lists

A list stores a series of items in a particular order. You access items using an index, or within a loop.

Make a list

```
bikes = ['trek', 'redline', 'giant']
```

Get the first item in a list

```
first_bike = bikes[0]
```

Get the last item in a list

```
last_bike = bikes[-1]
```

Looping through a list

```
for bike in bikes:
    print(bike)
```

Adding items to a list

```
bikes = []
bikes.append('trek')
bikes.append('redline')
bikes.append('giant')
```

Making numerical lists

```
squares = []
for x in range(1, 11):
    squares.append(x**2)
```

Lists (cont.)

List comprehensions

```
squares = [x**2 for x in range(1, 11)]
```

Slicing a list

```
finishers = ['sam', 'bob', 'ada', 'bea']
first_two = finishers[:2]
```

Copying a list

```
copy_of_bikes = bikes[:]
```

Tuples

Tuples are similar to lists, but the items in a tuple can't be modified.

Making a tuple

```
dimensions = (1920, 1080)
```

If statements

If statements are used to test for particular conditions and respond appropriately.

Conditional tests

```
equals            x == 42
not equal         x != 42
greater than      x > 42
  or equal to     x >= 42
less than         x < 42
  or equal to     x <= 42
```

Conditional test with lists

```
'trek' in bikes
'surly' not in bikes
```

Assigning boolean values

```
game_active = True
can_edit = False
```

A simple if test

```
if age >= 18:
    print("You can vote!")
```

If-elif-else statements

```
if age < 4:
    ticket_price = 0
elif age < 18:
    ticket_price = 10
else:
    ticket_price = 15
```

Dictionaries

Dictionaries store connections between pieces of information. Each item in a dictionary is a key-value pair.

A simple dictionary

```
alien = {'color': 'green', 'points': 5}
```

Accessing a value

```
print("The alien's color is " + alien['color'])
```

Adding a new key-value pair

```
alien['x_position'] = 0
```

Looping through all key-value pairs

```
fav_numbers = {'eric': 17, 'ever': 4}
for name, number in fav_numbers.items():
    print(name + ' loves ' + str(number))
```

Looping through all keys

```
fav_numbers = {'eric': 17, 'ever': 4}
for name in fav_numbers.keys():
    print(name + ' loves a number')
```

Looping through all the values

```
fav_numbers = {'eric': 17, 'ever': 4}
for number in fav_numbers.values():
    print(str(number) + ' is a favorite')
```

User input

Your programs can prompt the user for input. All input is stored as a string.

Prompting for a value

```
name = input("What's your name? ")
print("Hello, " + name + "!")
```

Prompting for numerical input

```
age = input("How old are you? ")
age = int(age)

pi = input("What's the value of pi? ")
pi = float(pi)
```

While loops

A while loop repeats a block of code as long as a certain condition is true.

A simple while loop

```
current_value = 1
while current_value <= 5:
    print(current_value)
    current_value += 1
```

Letting the user choose when to quit

```
msg = ''
while msg != 'quit':
    msg = input("What's your message? ")
    print(msg)
```

Functions

Functions are named blocks of code, designed to do one specific job. Information passed to a function is called an argument, and information received by a function is called a parameter.

A simple function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

Passing an argument

```
def greet_user(username):
    """Display a personalized greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
```

Default values for parameters

```
def make_pizza(topping='bacon'):
    """Make a single-topping pizza."""
    print("Have a " + topping + " pizza!")

make_pizza()
make_pizza('pepperoni')
```

Returning a value

```
def add_numbers(x, y):
    """Add two numbers and return the sum."""
    return x + y

sum = add_numbers(3, 5)
print(sum)
```

Classes

A class defines the behavior of an object and the kind of information an object can store. The information in a class is stored in attributes, and functions that belong to a class are called methods. A child class inherits the attributes and methods from its parent class.

Creating a dog class

```
class Dog():
    """Represent a dog."""

    def __init__(self, name):
        """Initialize dog object."""
        self.name = name

    def sit(self):
        """Simulate sitting."""
        print(self.name + " is sitting.")

my_dog = Dog('Peso')

print(my_dog.name + " is a great dog!")
my_dog.sit()
```

Inheritance

```
class SARDog(Dog):
    """Represent a search dog."""

    def __init__(self, name):
        """Initialize the sardog."""
        super().__init__(name)

    def search(self):
        """Simulate searching."""
        print(self.name + " is searching.")

my_dog = SARDog('Willie')

print(my_dog.name + " is a search dog.")
my_dog.sit()
my_dog.search()
```

Infinite Skills

If you had infinite programming skills, what would you build?

As you're learning to program, it's helpful to think about the real-world projects you'd like to create. It's a good habit to keep an "ideas" notebook that you can refer to whenever you want to start a new project. If you haven't done so already, take a few minutes and describe three projects you'd like to create.

Working with files

Your programs can read from files and write to files. Files are opened in read mode ('r') by default, but can also be opened in write mode ('w') and append mode ('a').

Reading a file and storing its lines

```
filename = 'siddhartha.txt'
with open(filename) as file_object:
    lines = file_object.readlines()

for line in lines:
    print(line)
```

Writing to a file

```
filename = 'journal.txt'
with open(filename, 'w') as file_object:
    file_object.write("I love programming.")
```

Appending to a file

```
filename = 'journal.txt'
with open(filename, 'a') as file_object:
    file_object.write("\nI love making games.")
```

Exceptions

Exceptions help you respond appropriately to errors that are likely to occur. You place code that might cause an error in the try block. Code that should run in response to an error goes in the except block. Code that should run only if the try block was successful goes in the else block.

Catching an exception

```
prompt = "How many tickets do you need? "
num_tickets = input(prompt)

try:
    num_tickets = int(num_tickets)
except ValueError:
    print("Please try again.")
else:
    print("Your tickets are printing.")
```

Zen of Python

Simple is better than complex

If you have a choice between a simple and a complex solution, and both work, use the simple solution. Your code will be easier to maintain, and it will be easier for you and others to build on that code later on.

Beginner's Python Cheat Sheet - Lists

What are lists?

A list stores a series of items in a particular order. Lists allow you to store sets of information in one place, whether you have just a few items or millions of items. Lists are one of Python's most powerful features readily accessible to new programmers, and they tie together many important concepts in programming.

Defining a list

Use square brackets to define a list, and use commas to separate individual items in the list. Use plural names for lists, to make your code easier to read.

Making a list

```
users = ['val', 'bob', 'mia', 'ron', 'ned']
```

Accessing elements

Individual elements in a list are accessed according to their position, called the index. The index of the first element is 0, the index of the second element is 1, and so forth. Negative indices refer to items at the end of the list. To get a particular element, write the name of the list and then the index of the element in square brackets.

Getting the first element

```
first_user = users[0]
```

Getting the second element

```
second_user = users[1]
```

Getting the last element

```
newest_user = users[-1]
```

Modifying individual items

Once you've defined a list, you can change individual elements in the list. You do this by referring to the index of the item you want to modify.

Changing an element

```
users[0] = 'valerie'
users[-2] = 'ronald'
```

Adding elements

You can add elements to the end of a list, or you can insert them wherever you like in a list.

Adding an element to the end of the list

```
users.append('amy')
```

Starting with an empty list

```
users = []
users.append('val')
users.append('bob')
users.append('mia')
```

Inserting elements at a particular position

```
users.insert(0, 'joe')
users.insert(3, 'bea')
```

Removing elements

You can remove elements by their position in a list, or by the value of the item. If you remove an item by its value, Python removes only the first item that has that value.

Deleting an element by its position

```
del users[-1]
```

Removing an item by its value

```
users.remove('mia')
```

Popping elements

If you want to work with an element that you're removing from the list, you can "pop" the element. If you think of the list as a stack of items, pop() takes an item off the top of the stack. By default pop() returns the last element in the list, but you can also pop elements from any position in the list.

Pop the last item from a list

```
most_recent_user = users.pop()
print(most_recent_user)
```

Pop the first item in a list

```
first_user = users.pop(0)
print(first_user)
```

List length

The len() function returns the number of items in a list.

Find the length of a list

```
num_users = len(users)
print("We have " + str(num_users) + " users.")
```

Sorting a list

The sort() method changes the order of a list permanently. The sorted() function returns a copy of the list, leaving the original list unchanged. You can sort the items in a list in alphabetical order, or reverse alphabetical order. You can also reverse the original order of the list. Keep in mind that lowercase and uppercase letters may affect the sort order.

Sorting a list permanently

```
users.sort()
```

Sorting a list permanently in reverse alphabetical order

```
users.sort(reverse=True)
```

Sorting a list temporarily

```
print(sorted(users))
print(sorted(users, reverse=True))
```

Reversing the order of a list

```
users.reverse()
```

Looping through a list

Lists can contain millions of items, so Python provides an efficient way to loop through all the items in a list. When you set up a loop, Python pulls each item from the list one at a time and stores it in a temporary variable, which you provide a name for. This name should be the singular version of the list name.

The indented block of code makes up the body of the loop, where you can work with each individual item. Any lines that are not indented run after the loop is completed.

Printing all items in a list

```
for user in users:
    print(user)
```

Printing a message for each item, and a separate message afterwards

```
for user in users:
    print("Welcome, " + user + "!")

print("Welcome, we're glad to see you all!")
```

The range() function

You can use the range() function to work with a set of numbers efficiently. The range() function starts at 0 by default, and stops one number below the number passed to it. You can use the list() function to efficiently generate a large list of numbers.

Printing the numbers 0 to 1000

```
for number in range(1001):
    print(number)
```

Printing the numbers 1 to 1000

```
for number in range(1, 1001):
    print(number)
```

Making a list of numbers from 1 to a million

```
numbers = list(range(1, 1000001))
```

Simple statistics

There are a number of simple statistics you can run on a list containing numerical data.

Finding the minimum value in a list

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
youngest = min(ages)
```

Finding the maximum value

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
oldest = max(ages)
```

Finding the sum of all values

```
ages = [93, 99, 66, 17, 85, 1, 35, 82, 2, 77]
total_years = sum(ages)
```

Slicing a list

You can work with any set of elements from a list. A portion of a list is called a slice. To slice a list start with the index of the first item you want, then add a colon and the index after the last item you want. Leave off the first index to start at the beginning of the list, and leave off the last index to slice through the end of the list.

Getting the first three items

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']
first_three = finishers[:3]
```

Getting the middle three items

```
middle_three = finishers[1:4]
```

Getting the last three items

```
last_three = finishers[-3:]
```

Copying a list

To copy a list make a slice that starts at the first item and ends at the last item. If you try to copy a list without using this approach, whatever you do to the copied list will affect the original list as well.

Making a copy of a list

```
finishers = ['kai', 'abe', 'ada', 'gus', 'zoe']
copy_of_finishers = finishers[:]
```

List comprehensions

You can use a loop to generate a list based on a range of numbers or on another list. This is a common operation, so Python offers a more efficient way to do it. List comprehensions may look complicated at first; if so, use the for loop approach until you're ready to start using comprehensions.

To write a comprehension, define an expression for the values you want to store in the list. Then write a for loop to generate input values needed to make the list.

Using a loop to generate a list of square numbers

```
squares = []
for x in range(1, 11):
    square = x**2
    squares.append(square)
```

Using a comprehension to generate a list of square numbers

```
squares = [x**2 for x in range(1, 11)]
```

Using a loop to convert a list of names to upper case

```
names = ['kai', 'abe', 'ada', 'gus', 'zoe']

upper_names = []
for name in names:
    upper_names.append(name.upper())
```

Using a comprehension to convert a list of names to upper case

```
names = ['kai', 'abe', 'ada', 'gus', 'zoe']

upper_names = [name.upper() for name in names]
```

Styling your code

Readability counts

- Use four spaces per indentation level.
- Keep your lines to 79 characters or fewer.
- Use single blank lines to group parts of your program visually.

Tuples

A tuple is like a list, except you can't change the values in a tuple once it's defined. Tuples are good for storing information that shouldn't be changed throughout the life of a program. Tuples are designated by parentheses instead of square brackets. (You can overwrite an entire tuple, but you can't change the individual elements in a tuple.)

Defining a tuple

```
dimensions = (800, 600)
```

Looping through a tuple

```
for dimension in dimensions:
    print(dimension)
```

Overwriting a tuple

```
dimensions = (800, 600)
print(dimensions)

dimensions = (1200, 900)
```

Visualizing your code

When you're first learning about data structures such as lists, it helps to visualize how Python is working with the information in your program. pythontutor.com is a great tool for seeing how Python keeps track of the information in a list. Try running the following code on pythontutor.com, and then run your own code.

Build a list and print the items in the list

```
dogs = []
dogs.append('willie')
dogs.append('hootz')
dogs.append('peso')
dogs.append('goblin')

for dog in dogs:
    print("Hello " + dog + "!")
print("I love these dogs!")

print("\nThese were my first two dogs:")
old_dogs = dogs[:2]
for old_dog in old_dogs:
    print(old_dog)

del dogs[0]
dogs.remove('peso')
print(dogs)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Dictionaries

What are dictionaries?

Python's dictionaries allow you to connect pieces of related information. Each piece of information in a dictionary is stored as a key-value pair. When you provide a key, Python returns the value associated with that key. You can loop through all the key-value pairs, all the keys, or all the values.

Defining a dictionary

Use curly braces to define a dictionary. Use colons to connect keys and values, and use commas to separate individual key-value pairs.

Making a dictionary

```
alien_0 = {'color': 'green', 'points': 5}
```

Accessing values

To access the value associated with an individual key give the name of the dictionary and then place the key in a set of square brackets. If the key you're asking for is not in the dictionary, an error will occur.

You can also use the get() method, which returns None instead of an error if the key doesn't exist. You can also specify a default value to use if the key is not in the dictionary.

Getting the value associated with a key

```
alien_0 = {'color': 'green', 'points': 5}

print(alien_0['color'])
print(alien_0['points'])
```

Getting the value with get()

```
alien_0 = {'color': 'green'}

alien_color = alien_0.get('color')
alien_points = alien_0.get('points', 0)

print(alien_color)
print(alien_points)
```

Adding new key-value pairs

You can store as many key-value pairs as you want in a dictionary, until your computer runs out of memory. To add a new key-value pair to an existing dictionary give the name of the dictionary and the new key in square brackets, and set it equal to the new value.

This also allows you to start with an empty dictionary and add key-value pairs as they become relevant.

Adding a key-value pair

```
alien_0 = {'color': 'green', 'points': 5}

alien_0['x'] = 0
alien_0['y'] = 25
alien_0['speed'] = 1.5
```

Adding to an empty dictionary

```
alien_0 = {}
alien_0['color'] = 'green'
alien_0['points'] = 5
```

Modifying values

You can modify the value associated with any key in a dictionary. To do so give the name of the dictionary and enclose the key in square brackets, then provide the new value for that key.

Modifying values in a dictionary

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

# Change the alien's color and point value.
alien_0['color'] = 'yellow'
alien_0['points'] = 10
print(alien_0)
```

Removing key-value pairs

You can remove any key-value pair you want from a dictionary. To do so use the del keyword and the dictionary name, followed by the key in square brackets. This will delete the key and its associated value.

Deleting a key-value pair

```
alien_0 = {'color': 'green', 'points': 5}
print(alien_0)

del alien_0['points']
print(alien_0)
```

Visualizing dictionaries

Try running some of these examples on pythontutor.com.

Looping through a dictionary

You can loop through a dictionary in three ways: you can loop through all the key-value pairs, all the keys, or all the values.

A dictionary only tracks the connections between keys and values; it doesn't track the order of items in the dictionary. If you want to process the information in order, you can sort the keys in your loop.

Looping through all key-value pairs

```
# Store people's favorite languages.
fav_languages = {
    'jen': 'python',
    'sarah': 'c',
    'edward': 'ruby',
    'phil': 'python',
    }

# Show each person's favorite language.
for name, language in fav_languages.items():
    print(name + ": " + language)
```

Looping through all the keys

```
# Show everyone who's taken the survey.
for name in fav_languages.keys():
    print(name)
```

Looping through all the values

```
# Show all the languages that have been chosen.
for language in fav_languages.values():
    print(language)
```

Looping through all the keys in order

```
# Show each person's favorite language,
#   in order by the person's name.
for name in sorted(fav_languages.keys()):
    print(name + ": " + language)
```

Dictionary length

You can find the number of key-value pairs in a dictionary.

Finding a dictionary's length

```
num_responses = len(fav_languages)
```

Nesting — A list of dictionaries

It's sometimes useful to store a set of dictionaries in a list; this is called nesting.

Storing dictionaries in a list

```
# Start with an empty list.
users = []

# Make a new user, and add them to the list.
new_user = {
    'last': 'fermi',
    'first': 'enrico',
    'username': 'efermi',
    }
users.append(new_user)

# Make another new user, and add them as well.
new_user = {
    'last': 'curie',
    'first': 'marie',
    'username': 'mcurie',
    }
users.append(new_user)

# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
    print("\n")
```

You can also define a list of dictionaries directly, without using append():

```
# Define a list of users, where each user
#   is represented by a dictionary.
users = [
    {
        'last': 'fermi',
        'first': 'enrico',
        'username': 'efermi',
    },
    {
        'last': 'curie',
        'first': 'marie',
        'username': 'mcurie',
    },
]

# Show all information about each user.
for user_dict in users:
    for k, v in user_dict.items():
        print(k + ": " + v)
    print("\n")
```

Nesting — Lists in a dictionary

Storing a list inside a dictionary alows you to associate more than one value with each key.

Storing lists in a dictionary

```
# Store multiple languages for each person.
fav_languages = {
    'jen': ['python', 'ruby'],
    'sarah': ['c'],
    'edward': ['ruby', 'go'],
    'phil': ['python', 'haskell'],
}

# Show all responses for each person.
for name, langs in fav_languages.items():
    print(name + ": ")
    for lang in langs:
        print("- " + lang)
```

Nesting — A dictionary of dictionaries

You can store a dictionary inside another dictionary. In this case each value associated with a key is itself a dictionary.

Storing dictionaries in a dictionary

```
users = {
    'aeinstein': {
        'first': 'albert',
        'last': 'einstein',
        'location': 'princeton',
        },
    'mcurie': {
        'first': 'marie',
        'last': 'curie',
        'location': 'paris',
        },
    }

for username, user_dict in users.items():
    print("\nUsername: " + username)
    full_name = user_dict['first'] + " "
    full_name += user_dict['last']
    location = user_dict['location']

    print("\tFull name: " + full_name.title())
    print("\tLocation: " + location.title())
```

Levels of nesting

Nesting is extremely useful in certain situations. However, be aware of making your code overly complex. If you're nesting items much deeper than what you see here there are probably simpler ways of managing your data, such as using classes.

Using an OrderedDict

Standard Python dictionaries don't keep track of the order in which keys and values are added; they only preserve the association between each key and its value. If you want to preserve the order in which keys and values are added, use an OrderedDict.

Preserving the order of keys and values

```
from collections import OrderedDict

# Store each person's languages, keeping
#  track of who respoded first.
fav_languages = OrderedDict()

fav_languages['jen'] = ['python', 'ruby']
fav_languages['sarah'] = ['c']
fav_languages['edward'] = ['ruby', 'go']
fav_languages['phil'] = ['python', 'haskell']

# Display the results, in the same order they
#  were entered.
for name, langs in fav_languages.items():
    print(name + ":")
    for lang in langs:
        print("- " + lang)
```

Generating a million dictionaries

You can use a loop to generate a large number of dictionaries efficiently, if all the dictionaries start out with similar data.

A million aliens

```
aliens = []

# Make a million green aliens, worth 5 points
#  each. Have them all start in one row.
for alien_num in range(1000000):
    new_alien = {}
    new_alien['color'] = 'green'
    new_alien['points'] = 5
    new_alien['x'] = 20 * alien_num
    new_alien['y'] = 0
    aliens.append(new_alien)

# Prove the list contains a million aliens.
num_aliens = len(aliens)

print("Number of aliens created:")
print(num_aliens)
```

Beginner's Python Cheat Sheet — If Statements and While Loops

What are if statements? What are while loops?

If statements allow you to examine the current state of a program and respond appropriately to that state. You can write a simple if statement that checks one condition, or you can create a complex series of if statements that idenitfy the exact conditions you're looking for.

While loops run as long as certain conditions remain true. You can use while loops to let your programs run as long as your users want them to.

Conditional Tests

A conditional test is an expression that can be evaluated as True or False. Python uses the values True and False to decide whether the code in an if statement should be executed.

Checking for equality

A single equal sign assigns a value to a variable. A double equal sign (==) checks whether two values are equal.

```
>>> car = 'bmw'
>>> car == 'bmw'
True
>>> car = 'audi'
>>> car == 'bmw'
False
```

Ignoring case when making a comparison

```
>>> car = 'Audi'
>>> car.lower() == 'audi'
True
```

Checking for inequality

```
>>> topping = 'mushrooms'
>>> topping != 'anchovies'
True
```

Numerical comparisons

Testing numerical values is similar to testing string values.

Testing equality and inequality

```
>>> age = 18
>>> age == 18
True
>>> age != 18
False
```

Comparison operators

```
>>> age = 19
>>> age < 21
True
>>> age <= 21
True
>>> age > 21
False
>>> age >= 21
False
```

Checking multiple conditions

You can check multiple conditions at the same time. The and operator returns True if all the conditions listed are True. The or operator returns True if any condition is True.

Using and to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 and age_1 >= 21
False
>>> age_1 = 23
>>> age_0 >= 21 and age_1 >= 21
True
```

Using or to check multiple conditions

```
>>> age_0 = 22
>>> age_1 = 18
>>> age_0 >= 21 or age_1 >= 21
True
>>> age_0 = 18
>>> age_0 >= 21 or age_1 >= 21
False
```

Boolean values

A boolean value is either True or False. Variables with boolean values are often used to keep track of certain conditions within a program.

Simple boolean values

```
game_active = True
can_edit = False
```

If statements

Several kinds of if statements exist. Your choice of which to use depends on the number of conditions you need to test. You can have as many elif blocks as you need, and the else block is always optional.

Simple if statement

```
age = 19

if age >= 18:
    print("You're old enough to vote!")
```

If-else statements

```
age = 17

if age >= 18:
    print("You're old enough to vote!")
else:
    print("You can't vote yet.")
```

The if-elif-else chain

```
age = 12

if age < 4:
   price = 0
elif age < 18:
    price = 5
else:
    price = 10

print("Your cost is $" + str(price) + ".")
```

Conditional tests with lists

You can easily test whether a certain value is in a list. You can also test whether a list is empty before trying to loop through the list.

Testing if a value is in a list

```
>>> players = ['al', 'bea', 'cyn', 'dale']
>>> 'al' in players
True
>>> 'eric' in players
False
```

Conditional tests with lists (cont.)

Testing if a value is not in a list

```
banned_users = ['ann', 'chad', 'dee']
user = 'erin'

if user not in banned_users:
    print("You can play!")
```

Checking if a list is empty

```
players = []

if players:
    for player in players:
        print("Player: " + player.title())
else:
    print("We have no players yet!")
```

Accepting input

You can allow your users to enter input using the input() statement. In Python 3, all input is stored as a string.

Simple input

```
name = input("What's your name? ")
print("Hello, " + name + ".")
```

Accepting numerical input

```
age = input("How old are you? ")
age = int(age)

if age >= 18:
    print("\nYou can vote!")
else:
    print("\nYou can't vote yet.")
```

Accepting input in Python 2.7

Use raw_input() in Python 2.7. This function interprets all input as a string, just as input() does in Python 3.

```
name = raw_input("What's your name? ")
print("Hello, " + name + ".")
```

While loops

A while loop repeats a block of code as long as a condition is True.

Counting to 5

```
current_number = 1

while current_number <= 5:
    print(current_number)
    current_number += 1
```

While loops (cont.)

Letting the user choose when to quit

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "

message = ""
while message != 'quit':
    message = input(prompt)

    if message != 'quit':
        print(message)
```

Using a flag

```
prompt = "\nTell me something, and I'll "
prompt += "repeat it back to you."
prompt += "\nEnter 'quit' to end the program. "

active = True
while active:
    message = input(prompt)

    if message == 'quit':
        active = False
    else:
        print(message)
```

Using break to exit a loop

```
prompt = "\nWhat cities have you visited?"
prompt += "\nEnter 'quit' when you're done. "

while True:
    city = input(prompt)

    if city == 'quit':
        break
    else:
        print("I've been to " + city + "!")
```

Accepting input with Sublime Text

Sublime Text doesn't run programs that prompt the user for input. You can use Sublime Text to write programs that prompt for input, but you'll need to run these programs from a terminal.

Breaking out of loops

You can use the break statement and the continue statement with any of Python's loops. For example you can use break to quit a for loop that's working through a list or a dictionary. You can use continue to skip over certain items when looping through a list or dictionary as well.

While loops (cont.)

Using continue in a loop

```
banned_users = ['eve', 'fred', 'gary', 'helen']

prompt = "\nAdd a player to your team."
prompt += "\nEnter 'quit' when you're done. "

players = []
while True:
    player = input(prompt)
    if player == 'quit':
        break
    elif player in banned_users:
        print(player + " is banned!")
        continue
    else:
        players.append(player)

print("\nYour team:")
for player in players:
    print(player)
```

Avoiding infinite loops

Every while loop needs a way to stop running so it won't continue to run forever. If there's no way for the condition to become False, the loop will never stop running.

An infinite loop

```
while True:
    name = input("\nWho are you? ")
    print("Nice to meet you, " + name + "!")
```

Removing all instances of a value from a list

The remove() method removes a specific value from a list, but it only removes the first instance of the value you provide. You can use a while loop to remove all instances of a particular value.

Removing all cats from a list of pets

```
pets = ['dog', 'cat', 'dog', 'fish', 'cat',
        'rabbit', 'cat']
print(pets)

while 'cat' in pets:
    pets.remove('cat')

print(pets)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Functions

What are functions?

Functions are named blocks of code designed to do one specific job. Functions allow you to write code once that can then be run whenever you need to accomplish the same task. Functions can take in the information they need, and return the information they generate. Using functions effectively makes your programs easier to write, read, test, and fix.

Defining a function

The first line of a function is its definition, marked by the keyword def. The name of the function is followed by a set of parentheses and a colon. A docstring, in triple quotes, describes what the function does. The body of a function is indented one level.

To call a function, give the name of the function followed by a set of parentheses.

Making a function

```
def greet_user():
    """Display a simple greeting."""
    print("Hello!")

greet_user()
```

Passing information to a function

Information that's passed to a function is called an argument; information that's received by a function is called a parameter. Arguments are included in parentheses after the function's name, and parameters are listed in parentheses in the function's definition.

Passing a single argument

```
def greet_user(username):
    """Display a simple greeting."""
    print("Hello, " + username + "!")

greet_user('jesse')
greet_user('diana')
greet_user('brandon')
```

Positional and keyword arguments

The two main kinds of arguments are positional and keyword arguments. When you use positional arguments Python matches the first argument in the function call with the first parameter in the function definition, and so forth.

With keyword arguments, you specify which parameter each argument should be assigned to in the function call. When you use keyword arguments, the order of the arguments doesn't matter.

Using positional arguments

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('hamster', 'harry')
describe_pet('dog', 'willie')
```

Using keyword arguments

```
def describe_pet(animal, name):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet(animal='hamster', name='harry')
describe_pet(name='willie', animal='dog')
```

Default values

You can provide a default value for a parameter. When function calls omit this argument the default value will be used. Parameters with default values must be listed after parameters without default values in the function's definition so positional arguments can still work correctly.

Using a default value

```
def describe_pet(name, animal='dog'):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    print("Its name is " + name + ".")

describe_pet('harry', 'hamster')
describe_pet('willie')
```

Using None to make an argument optional

```
def describe_pet(animal, name=None):
    """Display information about a pet."""
    print("\nI have a " + animal + ".")
    if name:
        print("Its name is " + name + ".")

describe_pet('hamster', 'harry')
describe_pet('snake')
```

Return values

A function can return a value or a set of values. When a function returns a value, the calling line must provide a variable in which to store the return value. A function stops running when it reaches a return statement.

Returning a single value

```
def get_full_name(first, last):
    """Return a neatly formatted full name."""
    full_name = first + ' ' + last
    return full_name.title()

musician = get_full_name('jimi', 'hendrix')
print(musician)
```

Returning a dictionary

```
def build_person(first, last):
    """Return a dictionary of information
    about a person.
    """
    person = {'first': first, 'last': last}
    return person

musician = build_person('jimi', 'hendrix')
print(musician)
```

Returning a dictionary with optional values

```
def build_person(first, last, age=None):
    """Return a dictionary of information
    about a person.
    """
    person = {'first': first, 'last': last}
    if age:
        person['age'] = age
    return person

musician = build_person('jimi', 'hendrix', 27)
print(musician)

musician = build_person('janis', 'joplin')
print(musician)
```

Visualizing functions

Try running some of these examples on pythontutor.com.

Passing a list to a function

You can pass a list as an argument to a function, and the function can work with the values in the list. Any changes the function makes to the list will affect the original list. You can prevent a function from modifying a list by passing a copy of the list as an argument.

Passing a list as an argument

```
def greet_users(names):
    """Print a simple greeting to everyone."""
    for name in names:
        msg = "Hello, " + name + "!"
        print(msg)

usernames = ['hannah', 'ty', 'margot']
greet_users(usernames)
```

Allowing a function to modify a list

The following example sends a list of models to a function for printing. The original list is emptied, and the second list is filled.

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
#  and print each of them.
unprinted = ['phone case', 'pendant', 'ring']
printed = []
print_models(unprinted, printed)

print("\nUnprinted:", unprinted)
print("Printed:", printed)
```

Preventing a function from modifying a list

The following example is the same as the previous one, except the original list is unchanged after calling print_models().

```
def print_models(unprinted, printed):
    """3d print a set of models."""
    while unprinted:
        current_model = unprinted.pop()
        print("Printing " + current_model)
        printed.append(current_model)

# Store some unprinted designs,
#  and print each of them.
original = ['phone case', 'pendant', 'ring']
printed = []

print_models(original[:], printed)
print("\nOriginal:", original)
print("Printed:", printed)
```

Passing an arbitrary number of arguments

*Sometimes you won't know how many arguments a function will need to accept. Python allows you to collect an arbitrary number of arguments into one parameter using the * operator. A parameter that accepts an arbitrary number of arguments must come last in the function definition.*

*The ** operator allows a parameter to collect an arbitrary number of keyword arguments.*

Collecting an arbitrary number of arguments

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)

# Make three pizzas with different toppings.
make_pizza('small', 'pepperoni')
make_pizza('large', 'bacon bits', 'pineapple')
make_pizza('medium', 'mushrooms', 'peppers',
        'onions', 'extra cheese')
```

Collecting an arbitrary number of keyword arguments

```
def build_profile(first, last, **user_info):
    """Build a user's profile dictionary."""
    # Build a dict with the required keys.
    profile = {'first': first, 'last': last}

    # Add any other keys and values.
    for key, value in user_info.items():
        profile[key] = value

    return profile

# Create two users with different kinds
#   of information.
user_0 = build_profile('albert', 'einstein',
        location='princeton')
user_1 = build_profile('marie', 'curie',
        location='paris', field='chemistry')

print(user_0)
print(user_1)
```

What's the best way to structure a function?

As you can see there are many ways to write and call a function. When you're starting out, aim for something that simply works. As you gain experience you'll develop an understanding of the more subtle advantages of different structures such as positional and keyword arguments, and the various approaches to importing functions. For now if your functions do what you need them to, you're doing well.

Modules

You can store your functions in a separate file called a module, and then import the functions you need into the file containing your main program. This allows for cleaner program files. (Make sure your module is stored in the same directory as your main program.)

Storing a function in a module

File: pizza.py

```
def make_pizza(size, *toppings):
    """Make a pizza."""
    print("\nMaking a " + size + " pizza.")
    print("Toppings:")
    for topping in toppings:
        print("- " + topping)
```

Importing an entire module

File: making_pizzas.py
Every function in the module is available in the program file.

```
import pizza

pizza.make_pizza('medium', 'pepperoni')
pizza.make_pizza('small', 'bacon', 'pineapple')
```

Importing a specific function

Only the imported functions are available in the program file.

```
from pizza import make_pizza

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

Giving a module an alias

```
import pizza as p

p.make_pizza('medium', 'pepperoni')
p.make_pizza('small', 'bacon', 'pineapple')
```

Giving a function an alias

```
from pizza import make_pizza as mp

mp('medium', 'pepperoni')
mp('small', 'bacon', 'pineapple')
```

Importing all functions from a module

Don't do this, but recognize it when you see it in others' code. It can result in naming conflicts, which can cause errors.

```
from pizza import *

make_pizza('medium', 'pepperoni')
make_pizza('small', 'bacon', 'pineapple')
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Classes

What are classes?

Classes are the foundation of object-oriented programming. Classes represent real-world things you want to model in your programs: for example dogs, cars, and robots. You use a class to make objects, which are specific instances of dogs, cars, and robots. A class defines the general behavior that a whole category of objects can have, and the information that can be associated with those objects.

Classes can inherit from each other – you can write a class that extends the functionality of an existing class. This allows you to code efficiently for a wide variety of situations.

Creating and using a class

Consider how we might model a car. What information would we associate with a car, and what behavior would it have? The information is stored in variables called attributes, and the behavior is represented by functions. Functions that are part of a class are called methods.

The Car class

```
class Car():
    """A simple attempt to model a car."""

    def __init__(self, make, model, year):
        """Initialize car attributes."""
        self.make = make
        self.model = model
        self.year = year

        # Fuel capacity and level in gallons.
        self.fuel_capacity = 15
        self.fuel_level = 0

    def fill_tank(self):
        """Fill gas tank to capacity."""
        self.fuel_level = self.fuel_capacity
        print("Fuel tank is full.")

    def drive(self):
        """Simulate driving."""
        print("The car is moving.")
```

Creating and using a class (cont.)

Creating an object from a class

```
my_car = Car('audi', 'a4', 2016)
```

Accessing attribute values

```
print(my_car.make)
print(my_car.model)
print(my_car.year)
```

Calling methods

```
my_car.fill_tank()
my_car.drive()
```

Creating multiple objects

```
my_car = Car('audi', 'a4', 2016)
my_old_car = Car('subaru', 'outback', 2013)
my_truck = Car('toyota', 'tacoma', 2010)
```

Modifying attributes

You can modify an attribute's value directly, or you can write methods that manage updating values more carefully.

Modifying an attribute directly

```
my_new_car = Car('audi', 'a4', 2016)
my_new_car.fuel_level = 5
```

Writing a method to update an attribute's value

```
def update_fuel_level(self, new_level):
    """Update the fuel level."""
    if new_level <= self.fuel_capacity:
        self.fuel_level = new_level
    else:
        print("The tank can't hold that much!")
```

Writing a method to increment an attribute's value

```
def add_fuel(self, amount):
    """Add fuel to the tank."""
    if (self.fuel_level + amount
            <= self.fuel_capacity):
        self.fuel_level += amount
        print("Added fuel.")
    else:
        print("The tank won't hold that much.")
```

Naming conventions

In Python class names are written in CamelCase and object names are written in lowercase with underscores. Modules that contain classes should still be named in lowercase with underscores.

Class inheritance

If the class you're writing is a specialized version of another class, you can use inheritance. When one class inherits from another, it automatically takes on all the attributes and methods of the parent class. The child class is free to introduce new attributes and methods, and override attributes and methods of the parent class.

To inherit from another class include the name of the parent class in parentheses when defining the new class.

The __init__() method for a child class

```
class ElectricCar(Car):
    """A simple model of an electric car."""

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attributes specific to electric cars.
        # Battery capacity in kWh.
        self.battery_size = 70
        # Charge level in %.
        self.charge_level = 0
```

Adding new methods to the child class

```
class ElectricCar(Car):
    --snip--
    def charge(self):
        """Fully charge the vehicle."""
        self.charge_level = 100
        print("The vehicle is fully charged.")
```

Using child methods and parent methods

```
my_ecar = ElectricCar('tesla', 'model s', 2016)

my_ecar.charge()
my_ecar.drive()
```

Finding your workflow

There are many ways to model real world objects and situations in code, and sometimes that variety can feel overwhelming. Pick an approach and try it – if your first attempt doesn't work, try a different approach.

Class inheritance (cont.)

Overriding parent methods

```
class ElectricCar(Car):
    --snip--
    def fill_tank(self):
        """Display an error message."""
        print("This car has no fuel tank!")
```

Instances as attributes

A class can have objects as attributes. This allows classes to work together to model complex situations.

A Battery class

```
class Battery():
    """A battery for an electric car."""

    def __init__(self, size=70):
        """Initialize battery attributes."""
        # Capacity in kWh, charge level in %.
        self.size = size
        self.charge_level = 0

    def get_range(self):
        """Return the battery's range."""
        if self.size == 70:
            return 240
        elif self.size == 85:
            return 270
```

Using an instance as an attribute

```
class ElectricCar(Car):
    --snip--

    def __init__(self, make, model, year):
        """Initialize an electric car."""
        super().__init__(make, model, year)

        # Attribute specific to electric cars.
        self.battery = Battery()

    def charge(self):
        """Fully charge the vehicle."""
        self.battery.charge_level = 100
        print("The vehicle is fully charged.")
```

Using the instance

```
my_ecar = ElectricCar('tesla', 'model x', 2016)

my_ecar.charge()
print(my_ecar.battery.get_range())
my_ecar.drive()
```

Importing classes

Class files can get long as you add detailed information and functionality. To help keep your program files uncluttered, you can store your classes in modules and import the classes you need into your main program.

Storing classes in a file

car.py

```
"""Represent gas and electric cars."""

class Car():
    """A simple attempt to model a car."""
    --snip—

class Battery():
    """A battery for an electric car."""
    --snip--

class ElectricCar(Car):
    """A simple model of an electric car."""
    --snip--
```

Importing individual classes from a module

my_cars.py

```
from car import Car, ElectricCar

my_beetle = Car('volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = ElectricCar('tesla', 'model s',
2016)
my_tesla.charge()
my_tesla.drive()
```

Importing an entire module

```
import car

my_beetle = car.Car(
        'volkswagen', 'beetle', 2016)
my_beetle.fill_tank()
my_beetle.drive()

my_tesla = car.ElectricCar(
        'tesla', 'model s', 2016)
my_tesla.charge()
my_tesla.drive()
```

Importing all classes from a module

(Don't do this, but recognize it when you see it.)

```
from car import *

my_beetle = Car('volkswagen', 'beetle', 2016)
```

Classes in Python 2.7

Classes should inherit from object

```
class ClassName(object):
```

The Car class in Python 2.7

```
class Car(object):
```

Child class __init__() method is different

```
class ChildClassName(ParentClass):
    def __init__(self):
        super(ClassName, self).__init__()
```

The ElectricCar class in Python 2.7

```
class ElectricCar(Car):
    def __init__(self, make, model, year):
        super(ElectricCar, self).__init__(
                make, model, year)
```

Storing objects in a list

A list can hold as many items as you want, so you can make a large number of objects from a class and store them in a list.

Here's an example showing how to make a fleet of rental cars, and make sure all the cars are ready to drive.

A fleet of rental cars

```
from car import Car, ElectricCar

# Make lists to hold a fleet of cars.
gas_fleet = []
electric_fleet = []

# Make 500 gas cars and 250 electric cars.
for _ in range(500):
    car = Car('ford', 'focus', 2016)
    gas_fleet.append(car)
for _ in range(250):
    ecar = ElectricCar('nissan', 'leaf', 2016)
    electric_fleet.append(ecar)

# Fill the gas cars, and charge electric cars.
for car in gas_fleet:
    car.fill_tank()
for ecar in electric_fleet:
    ecar.charge()

print("Gas cars:", len(gas_fleet))
print("Electric cars:", len(electric_fleet))
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Files and Exceptions

What are files? What are exceptions?

Your programs can read information in from files, and they can write data to files. Reading from files allows you to work with a wide variety of information; writing to files allows users to pick up where they left off the next time they run your program. You can write text to files, and you can store Python structures such as lists in data files.

Exceptions are special objects that help your programs respond to errors in appropriate ways. For example if your program tries to open a file that doesn't exist, you can use exceptions to display an informative error message instead of having the program crash.

Reading from a file

To read from a file your program needs to open the file and then read the contents of the file. You can read the entire contents of the file at once, or read the file line by line. The with statement makes sure the file is closed properly when the program has finished accessing the file.

Reading an entire file at once

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    contents = f_obj.read()

print(contents)
```

Reading line by line

Each line that's read from the file has a newline character at the end of the line, and the print function adds its own newline character. The rstrip() method gets rid of the the extra blank lines this would result in when printing to the terminal.

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    for line in f_obj:
        print(line.rstrip())
```

Reading from a file (cont.)

Storing the lines in a list

```
filename = 'siddhartha.txt'

with open(filename) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

Writing to a file

Passing the 'w' argument to open() tells Python you want to write to the file. Be careful; this will erase the contents of the file if it already exists. Passing the 'a' argument tells Python you want to append to the end of an existing file.

Writing to an empty file

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!")
```

Writing multiple lines to an empty file

```
filename = 'programming.txt'

with open(filename, 'w') as f:
    f.write("I love programming!\n")
    f.write("I love creating new games.\n")
```

Appending to a file

```
filename = 'programming.txt'

with open(filename, 'a') as f:
    f.write("I also love working with data.\n")
    f.write("I love making apps as well.\n")
```

File paths

When Python runs the open() function, it looks for the file in the same directory where the program that's being excuted is stored. You can open a file from a subfolder using a relative path. You can also use an absolute path to open any file on your system.

Opening a file from a subfolder

```
f_path = "text_files/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()

for line in lines:
    print(line.rstrip())
```

File paths (cont.)

Opening a file using an absolute path

```
f_path = "/home/ehmatthes/books/alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

Opening a file on Windows

Windows will sometimes interpret forward slashes incorrectly. If you run into this, use backslashes in your file paths.

```
f_path = "C:\Users\ehmatthes\books\alice.txt"

with open(f_path) as f_obj:
    lines = f_obj.readlines()
```

The try-except block

When you think an error may occur, you can write a try-except block to handle the exception that might be raised. The try block tells Python to try running some code, and the except block tells Python what to do if the code results in a particular kind of error.

Handling the ZeroDivisionError exception

```
try:
    print(5/0)
except ZeroDivisionError:
    print("You can't divide by zero!")
```

Handling the FileNotFoundError exception

```
f_name = 'siddhartha.txt'

try:
    with open(f_name) as f_obj:
        lines = f_obj.readlines()
except FileNotFoundError:
    msg = "Can't find file {0}.".format(f_name)
    print(msg)
```

Knowing which exception to handle

It can be hard to know what kind of exception to handle when writing code. Try writing your code without a try block, and make it generate an error. The traceback will tell you what kind of exception your program needs to handle.

The else block

The try block should only contain code that may cause an error. Any code that depends on the try block running successfully should be placed in the else block.

Using an else block

```
print("Enter two numbers. I'll divide them.")

x = input("First number: ")
y = input("Second number: ")

try:
    result = int(x) / int(y)
except ZeroDivisionError:
    print("You can't divide by zero!")
else:
    print(result)
```

Preventing crashes from user input

Without the except block in the following example, the program would crash if the user tries to divide by zero. As written, it will handle the error gracefully and keep running.

```
"""A simple calculator for division only."""

print("Enter two numbers. I'll divide them.")
print("Enter 'q' to quit.")

while True:
    x = input("\nFirst number: ")
    if x == 'q':
        break
    y = input("Second number: ")
    if y == 'q':
        break

    try:
        result = int(x) / int(y)
    except ZeroDivisionError:
        print("You can't divide by zero!")
    else:
        print(result)
```

Deciding which errors to report

Well-written, properly tested code is not very prone to internal errors such as syntax or logical errors. But every time your program depends on something external such as user input or the existence of a file, there's a possibility of an exception being raised.

It's up to you how to communicate errors to your users. Sometimes users need to know if a file is missing; sometimes it's better to handle the error silently. A little experience will help you know how much to report.

Failing silently

Sometimes you want your program to just continue running when it encounters an error, without reporting the error to the user. Using the pass statement in an else block allows you to do this.

Using the pass statement in an else block

```
f_names = ['alice.txt', 'siddhartha.txt',
        'moby_dick.txt', 'little_women.txt']

for f_name in f_names:
    # Report the length of each file found.
    try:
        with open(f_name) as f_obj:
            lines = f_obj.readlines()
    except FileNotFoundError:
        # Just move on to the next file.
        pass
    else:
        num_lines = len(lines)
        msg = "{0} has {1} lines.".format(
            f_name, num_lines)
        print(msg)
```

Avoid bare except blocks

Exception-handling code should catch specific exceptions that you expect to happen during your program's execution. A bare except block will catch all exceptions, including keyboard interrupts and system exits you might need when forcing a program to close.

If you want to use a try block and you're not sure which exception to catch, use Exception. It will catch most exceptions, but still allow you to interrupt programs intentionally.

Don't use bare except blocks

```
try:
    # Do something
except:
    pass
```

Use Exception instead

```
try:
    # Do something
except Exception:
    pass
```

Printing the exception

```
try:
    # Do something
except Exception as e:
    print(e, type(e))
```

Storing data with json

The json module allows you to dump simple Python data structures into a file, and load the data from that file the next time the program runs. The JSON data format is not specific to Python, so you can share this kind of data with people who work in other languages as well.

Knowing how to manage exceptions is important when working with stored data. You'll usually want to make sure the data you're trying to load exists before working with it.

Using json.dump() to store data

```
"""Store some numbers."""

import json

numbers = [2, 3, 5, 7, 11, 13]

filename = 'numbers.json'
with open(filename, 'w') as f_obj:
    json.dump(numbers, f_obj)
```

Using json.load() to read data

```
"""Load some previously stored numbers."""

import json

filename = 'numbers.json'
with open(filename) as f_obj:
    numbers = json.load(f_obj)

print(numbers)
```

Making sure the stored data exists

```
import json

f_name = 'numbers.json'

try:
    with open(f_name) as f_obj:
        numbers = json.load(f_obj)
except FileNotFoundError:
    msg = "Can't find {0}.".format(f_name)
    print(msg)
else:
    print(numbers)
```

Practice with exceptions

Take a program you've already written that prompts for user input, and add some error-handling code to the program.

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Testing Your Code

Why test your code?

When you write a function or a class, you can also write tests for that code. Testing proves that your code works as it's supposed to in the situations it's designed to handle, and also when people use your programs in unexpected ways. Writing tests gives you confidence that your code will work correctly as more people begin to use your programs. You can also add new features to your programs and know that you haven't broken existing behavior.

A unit test verifies that one specific aspect of your code works as it's supposed to. A test case is a collection of unit tests which verify your code's behavior in a wide variety of situations.

Testing a function: A passing test

Python's unittest module provides tools for testing your code. To try it out, we'll create a function that returns a full name. We'll use the function in a regular program, and then build a test case for the function.

A function to test

Save this as full_names.py

```
def get_full_name(first, last):
    """Return a full name."""
    full_name = "{0} {1}".format(first, last)
    return full_name.title()
```

Using the function

Save this as names.py

```
from full_names import get_full_name

janis = get_full_name('janis', 'joplin')
print(janis)

bob = get_full_name('bob', 'dylan')
print(bob)
```

Testing a function (cont.)

Building a testcase with one unit test

To build a test case, make a class that inherits from `unittest.TestCase` *and write methods that begin with* `test_`*. Save this as test_full_names.py*

```
import unittest
from full_names import get_full_name

class NamesTestCase(unittest.TestCase):
    """Tests for names.py."""

    def test_first_last(self):
        """Test names like Janis Joplin."""
        full_name = get_full_name('janis',
                'joplin')
        self.assertEqual(full_name,
                'Janis Joplin')

unittest.main()
```

Running the test

Python reports on each unit test in the test case. The dot reports a single passing test. Python informs us that it ran 1 test in less than 0.001 seconds, and the OK lets us know that all unit tests in the test case passed.

```
.
--------------------------------------
Ran 1 test in 0.000s

OK
```

Testing a function: A failing test

Failing tests are important; they tell you that a change in the code has affected existing behavior. When a test fails, you need to modify the code so the existing behavior still works.

Modifying the function

We'll modify `get_full_name()` *so it handles middle names, but we'll do it in a way that breaks existing behavior.*

```
def get_full_name(first, middle, last):
    """Return a full name."""
    full_name = "{0} {1} {2}".format(first,
                                     middle, last)
    return full_name.title()
```

Using the function

```
from full_names import get_full_name

john = get_full_name('john', 'lee', 'hooker')
print(john)

david = get_full_name('david', 'lee', 'roth')
print(david)
```

A failing test (cont.)

Running the test

When you change your code, it's important to run your existing tests. This will tell you whether the changes you made affected existing behavior.

```
E
======================================================
ERROR: test_first_last (__main__.NamesTestCase)
Test names like Janis Joplin.
------------------------------------------------------
Traceback (most recent call last):
  File "test_full_names.py", line 10,
    in test_first_last
    'joplin')
TypeError: get_full_name() missing 1 required
  positional argument: 'last'

------------------------------------------------------
Ran 1 test in 0.001s

FAILED (errors=1)
```

Fixing the code

When a test fails, the code needs to be modified until the test passes again. (Don't make the mistake of rewriting your tests to fit your new code.) Here we can make the middle name optional.

```
def get_full_name(first, last, middle=''):
    """Return a full name."""
    if middle:
        full_name = "{0} {1} {2}".format(first,
                                         middle, last)
    else:
        full_name = "{0} {1}".format(first,
                                     last)
    return full_name.title()
```

Running the test

Now the test should pass again, which means our original functionality is still intact.

```
.
--------------------------------------
Ran 1 test in 0.000s

OK
```

Adding new tests

You can add as many unit tests to a test case as you need. To write a new test, add a new method to your test case class.

Testing middle names

We've shown that `get_full_name()` works for first and last names. Let's test that it works for middle names as well.

```
import unittest
from full_names import get_full_name

class NamesTestCase(unittest.TestCase):
    """Tests for names.py."""

    def test_first_last(self):
        """Test names like Janis Joplin."""
        full_name = get_full_name('janis',
                'joplin')
        self.assertEqual(full_name,
                'Janis Joplin')

    def test_middle(self):
        """Test names like David Lee Roth."""
        full_name = get_full_name('david',
                'roth', 'lee')
        self.assertEqual(full_name,
                'David Lee Roth')

unittest.main()
```

Running the tests

The two dots represent two passing tests.

```
..
--------------------------------------
Ran 2 tests in 0.000s

OK
```

A variety of assert methods

Python provides a number of assert methods you can use to test your code.

Verify that a==b, or a != b

```
assertEqual(a, b)
assertNotEqual(a, b)
```

Verify that x is True, or x is False

```
assertTrue(x)
assertFalse(x)
```

Verify an item is in a list, or not in a list

```
assertIn(item, list)
assertNotIn(item, list)
```

Testing a class

Testing a class is similar to testing a function, since you'll mostly be testing your methods.

A class to test

Save as accountant.py

```
class Accountant():
    """Manage a bank account."""

    def __init__(self, balance=0):
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        self.balance -= amount
```

Building a testcase

For the first test, we'll make sure we can start out with different initial balances. Save this as test_accountant.py.

```
import unittest
from accountant import Accountant

class TestAccountant(unittest.TestCase):
    """Tests for the class Accountant."""

    def test_initial_balance(self):
        # Default balance should be 0.
        acc = Accountant()
        self.assertEqual(acc.balance, 0)

        # Test non-default balance.
        acc = Accountant(100)
        self.assertEqual(acc.balance, 100)

unittest.main()
```

Running the test

```
.
--------------------------------------
Ran 1 test in 0.000s

OK
```

When is it okay to modify tests?

In general you shouldn't modify a test once it's written. When a test fails it usually means new code you've written has broken existing functionality, and you need to modify the new code until all existing tests pass.

If your original requirements have changed, it may be appropriate to modify some tests. This usually happens in the early stages of a project when desired behavior is still being sorted out.

The setUp() method

When testing a class, you usually have to make an instance of the class. The `setUp()` method is run before every test. Any instances you make in `setUp()` are available in every test you write.

Using setUp() to support multiple tests

The instance `self.acc` can be used in each new test.

```
import unittest
from accountant import Accountant

class TestAccountant(unittest.TestCase):
    """Tests for the class Accountant."""

    def setUp(self):
        self.acc = Accountant()

    def test_initial_balance(self):
        # Default balance should be 0.
        self.assertEqual(self.acc.balance, 0)

        # Test non-default balance.
        acc = Accountant(100)
        self.assertEqual(acc.balance, 100)

    def test_deposit(self):
        # Test single deposit.
        self.acc.deposit(100)
        self.assertEqual(self.acc.balance, 100)

        # Test multiple deposits.
        self.acc.deposit(100)
        self.acc.deposit(100)
        self.assertEqual(self.acc.balance, 300)

    def test_withdrawal(self):
        # Test single withdrawal.
        self.acc.deposit(1000)
        self.acc.withdraw(100)
        self.assertEqual(self.acc.balance, 900)

unittest.main()
```

Running the tests

```
...
--------------------------------------
Ran 3 tests in 0.001s

OK
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet - Pygame

What is Pygame?

Pygame is a framework for making games using Python. Making games is fun, and it's a great way to expand your programming skills and knowledge. Pygame takes care of many of the lower-level tasks in building games, which lets you focus on the aspects of your game that make it interesting.

Installing Pygame

Pygame runs on all systems, but setup is slightly different on each OS. The instructions here assume you're using Python 3, and provide a minimal installation of Pygame. If these instructions don't work for your system, see the more detailed notes at http://ehmatthes.github.io/pcc/.

Pygame on Linux

```
$ sudo apt-get install python3-dev mercurial
    libsdl-image1.2-dev libsdl2-dev
    libsdl-ttf2.0-dev
$ pip install --user
    hg+http://bitbucket.org/pygame/pygame
```

Pygame on OS X

This assumes you've used Homebrew to install Python 3.

```
$ brew install hg sdl sdl_image sdl_ttf
$ pip install --user
    hg+http://bitbucket.org/pygame/pygame
```

Pygame on Windows

Find an installer at https://bitbucket.org/pygame/pygame/downloads/ or http://www.lfd.uci.edu/~gohlke/pythonlibs/#pygame that matches your version of Python. Run the installer file if it's a .exe or .msi file. If it's a .whl file, use pip to install Pygame:

```
> python -m pip install --user
    pygame-1.9.2a0-cp35-none-win32.whl
```

Testing your installation

To test your installation, open a terminal session and try to import Pygame. If you don't get any error messages, your installation was successful.

```
$ python
>>> import pygame
>>>
```

Starting a game

The following code sets up an empty game window, and starts an event loop and a loop that continually refreshes the screen.

An empty game window

```
import sys
import pygame as pg

def run_game():
    # Initialize and set up screen.
    pg.init()
    screen = pg.display.set_mode((1200, 800))
    pg.display.set_caption("Alien Invasion")

    # Start main loop.
    while True:
        # Start event loop.
        for event in pg.event.get():
            if event.type == pg.QUIT:
                sys.exit()

        # Refresh screen.
        pg.display.flip()

run_game()
```

Setting a custom window size

The display.set_mode() function accepts a tuple that defines the screen size.

```
screen_dim = (1200, 800)
screen = pg.display.set_mode(screen_dim)
```

Setting a custom background color

Colors are defined as a tuple of red, green, and blue values. Each value ranges from 0-255.

```
bg_color = (230, 230, 230)
screen.fill(bg_color)
```

Pygame rect objects

Many objects in a game can be treated as simple rectangles, rather than their actual shape. This simplifies code without noticeably affecting game play. Pygame has a rect object that makes it easy to work with game objects.

Getting the screen rect object

We already have a screen object; we can easily access the rect object associated with the screen.

```
screen_rect = screen.get_rect()
```

Finding the center of the screen

Rect objects have a center attribute which stores the center point.

```
screen_center = screen_rect.center
```

Pygame rect objects (cont.)

Useful rect attributes

Once you have a rect object, there are a number of attributes that are useful when positioning objects and detecting relative positions of objects. (You can find more attributes in the Pygame documentation.)

```
# Individual x and y values:
screen_rect.left, screen_rect.right
screen_rect.top, screen_rect.bottom
screen_rect.centerx, screen_rect.centery
screen_rect.width, screen_rect.height

# Tuples
screen_rect.center
screen_rect.size
```

Creating a rect object

You can create a rect object from scratch. For example a small rect object that's filled in can represent a bullet in a game. The Rect() class takes the coordinates of the upper left corner, and the width and height of the rect. The draw.rect() function takes a screen object, a color, and a rect. This function fills the given rect with the given color.

```
bullet_rect = pg.Rect(100, 100, 3, 15)
color = (100, 100, 100)
pg.draw.rect(screen, color, bullet_rect)
```

Working with images

Many objects in a game are images that are moved around the screen. It's easiest to use bitmap (.bmp) image files, but you can also configure your system to work with jpg, png, and gif files as well.

Loading an image

```
ship = pg.image.load('images/ship.bmp')
```

Getting the rect object from an image

```
ship_rect = ship.get_rect()
```

Positioning an image

With rects, it's easy to position an image wherever you want on the screen, or in relation to another object. The following code positions a ship object at the bottom center of the screen.

```
ship_rect.midbottom = screen_rect.midbottom
```

Working with images (cont.)

Drawing an image to the screen

Once an image is loaded and positioned, you can draw it to the screen with the blit() method. The blit() method acts on the screen object, and takes the image object and image rect as arguments.

```
# Draw ship to screen.
screen.blit(ship, ship_rect)
```

The blitme() method

Game objects such as ships are often written as classes. Then a blitme() method is usually defined, which draws the object to the screen.

```
def blitme(self):
    """Draw ship at current location."""
    self.screen.blit(self.image, self.rect)
```

Responding to keyboard input

Pygame watches for events such as key presses and mouse actions. You can detect any event you care about in the event loop, and respond with any action that's appropriate for your game.

Responding to key presses

Pygame's main event loop registers a KEYDOWN event any time a key is pressed. When this happens, you can check for specific keys.

```
for event in pg.event.get():
    if event.type == pg.KEYDOWN:
        if event.key == pg.K_RIGHT:
            ship_rect.x += 1
        elif event.key == pg.K_LEFT:
            ship_rect.x -= 1
        elif event.key == pg.K_SPACE:
            ship.fire_bullet()
        elif event.key == pg.K_q:
            sys.exit()
```

Responding to released keys

When the user releases a key, a KEYUP event is triggered.

```
if event.type == pg.KEYUP:
    if event.key == pg.K_RIGHT:
        ship.moving_right = False
```

Pygame documentation

The Pygame documentation is really helpful when building your own games. The home page for the Pygame project is at http://pygame.org/, and the home page for the documentation is at http://pygame.org/docs/.

The most useful part of the documentation are the pages about specific parts of Pygame, such as the Rect() class and the sprite module. You can find a list of these elements at the top of the help pages.

Responding to mouse events

Pygame's event loop registers an event any time the mouse moves, or a mouse button is pressed or released.

Responding to the mouse button

```
for event in pg.event.get():
    if event.type == pg.MOUSEBUTTONDOWN:
        ship.fire_bullet()
```

Finding the mouse position

The mouse position is returned as a tuple.

```
mouse_pos = pg.mouse.get_pos()
```

Clicking a button

You might want to know if the cursor is over an object such as a button. The rect.collidepoint() method returns true when a point is inside a rect object.

```
if button_rect.collidepoint(mouse_pos):
    start_game()
```

Hiding the mouse

```
pg.mouse.set_visible(False)
```

Pygame groups

Pygame has a Group class which makes working with a group of similar objects easier. A group is like a list, with some extra functionality that's helpful when building games.

Making and filling a group

An object that will be placed in a group must inherit from Sprite.

```
from pygame.sprite import Sprite, Group

def Bullet(Sprite):
    ...
    def draw_bullet(self):
        ...
    def update(self):
        ...

bullets = Group()

new_bullet = Bullet()
bullets.add(new_bullet)
```

Looping through the items in a group

The sprites() method returns all the members of a group.

```
for bullet in bullets.sprites():
    bullet.draw_bullet()
```

Calling update() on a group

Calling update() on a group automatically calls update() on each member of the group.

```
bullets.update()
```

Pygame groups (cont.)

Removing an item from a group

It's important to delete elements that will never appear again in the game, so you don't waste memory and resources.

```
bullets.remove(bullet)
```

Detecting collisions

You can detect when a single object collides with any member of a group. You can also detect when any member of one group collides with a member of another group.

Collisions between a single object and a group

The spritecollideany() function takes an object and a group, and returns True if the object overlaps with any member of the group.

```
if pg.sprite.spritecollideany(ship, aliens):
    ships_left -= 1
```

Collisions between two groups

The sprite.groupcollide() function takes two groups, and two booleans. The function returns a dictionary containing information about the members that have collided. The booleans tell Pygame whether to delete the members of either group that have collided.

```
collisions = pg.sprite.groupcollide(
        bullets, aliens, True, True)

score += len(collisions) * alien_point_value
```

Rendering text

You can use text for a variety of purposes in a game. For example you can share information with players, and you can display a score.

Displaying a message

The following code defines a message, then a color for the text and the background color for the message. A font is defined using the default system font, with a font size of 48. The font.render() function is used to create an image of the message, and we get the rect object associated with the image. We then center the image on the screen and display it.

```
msg = "Play again?"
msg_color = (100, 100, 100)
bg_color = (230, 230, 230)

f = pg.font.SysFont(None, 48)
msg_image = f.render(msg, True, msg_color,
        bg_color)
msg_image_rect = msg_image.get_rect()
msg_image_rect.center = screen_rect.center
screen.blit(msg_image, msg_image_rect)
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — matplotlib

What is matplotlib?

Data visualization involves exploring data through visual representations. The matplotlib package helps you make visually appealing representations of the data you're working with. matplotlib is extremely flexible; these examples will help you get started with a few simple visualizations.

Installing matplotlib

matplotlib runs on all systems, but setup is slightly different depending on your OS. If the minimal instructions here don't work for you, see the more detailed instructions at http://ehmatthes.github.io/pcc/. You should also consider installing the Anaconda distrubution of Python from https://continuum.io/downloads/, which includes matplotlib.

matplotlib on Linux

```
$ sudo apt-get install python3-matplotlib
```

matplotlib on OS X

Start a terminal session and enter `import matplotlib` *to see if it's already installed on your system. If not, try this command:*

```
$ pip install --user matplotlib
```

matplotlib on Windows

You first need to install Visual Studio, which you can do from https://dev.windows.com/. The Community edition is free. Then go to https://pypi.python.org/pypi/matplotlib/ or http://www.lfd.uic.edu/~gohlke/pythonlibs/#matplotlib and download an appropriate installer file.

Line graphs and scatter plots

Making a line graph

```
import matplotlib.pyplot as plt

x_values = [0, 1, 2, 3, 4, 5]
squares = [0, 1, 4, 9, 16, 25]
plt.plot(x_values, squares)
plt.show()
```

Line graphs and scatter plots (cont.)

Making a scatter plot

The scatter() function takes a list of x values and a list of y values, and a variety of optional arguments. The `s=10` *argument controls the size of each point.*

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]

plt.scatter(x_values, squares, s=10)
plt.show()
```

Customizing plots

Plots can be customized in a wide variety of ways. Just about any element of a plot can be customized.

Adding titles and labels, and scaling axes

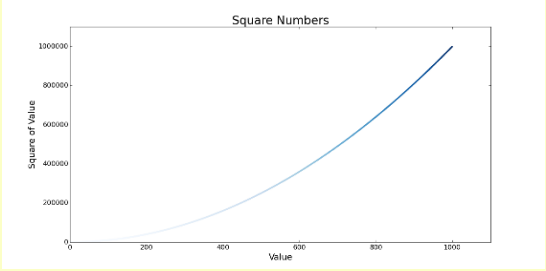
```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]
plt.scatter(x_values, squares, s=10)

plt.title("Square Numbers", fontsize=24)
plt.xlabel("Value", fontsize=18)
plt.ylabel("Square of Value", fontsize=18)
plt.tick_params(axis='both', which='major',
                labelsize=14)
plt.axis([0, 1100, 0, 1100000])

plt.show()
```

Using a colormap

A colormap varies the point colors from one shade to another, based on a certain value for each point. The value used to determine the color of each point is passed to the `c` *argument, and the* `cmap` *argument specifies which colormap to use.*

The `edgecolor='none'` *argument removes the black outline from each point.*

```
plt.scatter(x_values, squares, c=squares,
        cmap=plt.cm.Blues, edgecolor='none',
        s=10)
```



Customizing plots (cont.)

Emphasizing points

You can plot as much data as you want on one plot. Here we re-plot the first and last points larger to emphasize them.

```
import matplotlib.pyplot as plt

x_values = list(range(1000))
squares = [x**2 for x in x_values]
plt.scatter(x_values, squares, c=squares,
        cmap=plt.cm.Blues, edgecolor='none',
        s=10)

plt.scatter(x_values[0], squares[0], c='green',
        edgecolor='none', s=100)
plt.scatter(x_values[-1], squares[-1], c='red',
        edgecolor='none', s=100)

plt.title("Square Numbers", fontsize=24)
--snip--
```

Removing axes

You can customize or remove axes entirely. Here's how to access each axis, and hide it.

```
plt.axes().get_xaxis().set_visible(False)
plt.axes().get_yaxis().set_visible(False)
```

Setting a custom figure size

You can make your plot as big or small as you want. Before plotting your data, add the following code. The `dpi` *argument is optional; if you don't know your system's resolution you can omit the argument and adjust the* `figsize` *argument accordingly.*

```
plt.figure(dpi=128, figsize=(10, 6))
```

Saving a plot

The matplotlib viewer has an interactive save button, but you can also save your visualizations programmatically. To do so, replace `plt.show()` *with* `plt.savefig()`*. The* `bbox_inches='tight'` *argument trims extra whitespace from the plot.*

```
plt.savefig('squares.png', bbox_inches='tight')
```

Online resources

The matplotlib gallery and documentation are at http://matplotlib.org/. Be sure to visit the examples, gallery, and pyplot links.

Multiple plots

You can make as many plots as you want on one figure. When you make multiple plots, you can emphasize relationships in the data. For example you can fill the space between two sets of data.

Plotting two sets of data

Here we use `plt.scatter()` twice to plot square numbers and cubes on the same figure.

```
import matplotlib.pyplot as plt

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

plt.scatter(x_values, squares, c='blue',
        edgecolor='none', s=20)
plt.scatter(x_values, cubes, c='red',
        edgecolor='none', s=20)

plt.axis([0, 11, 0, 1100])
plt.show()
```

Filling the space between data sets

The `fill_between()` method fills the space between two data sets. It takes a series of x-values and two series of y-values. It also takes a `facecolor` to use for the fill, and an optional `alpha` argument that controls the color's transparency.

```
plt.fill_between(x_values, cubes, squares,
        facecolor='blue', alpha=0.25)
```

Working with dates and times

Many interesting data sets have a date or time as the x-value. Python's datetime module helps you work with this kind of data.

Generating the current date

The `datetime.now()` function returns a datetime object representing the current date and time.

```
from datetime import datetime as dt

today = dt.now()
date_string = dt.strftime(today, '%m/%d/%Y')
print(date_string)
```

Generating a specific date

You can also generate a datetime object for any date and time you want. The positional order of arguments is year, month, and day. The hour, minute, second, and microsecond arguments are optional.

```
from datetime import datetime as dt

new_years = dt(2017, 1, 1)
fall_equinox = dt(year=2016, month=9, day=22)
```

Working with dates and times (cont.)

Datetime formatting arguments

The `strftime()` function generates a formatted string from a datetime object, and the `strptime()` function genereates a datetime object from a string. The following codes let you work with dates exactly as you need to.

```
%A    Weekday name, such as Monday
%B    Month name, such as January
%m    Month, as a number (01 to 12)
%d    Day of the month, as a number (01 to 31)
%Y    Four-digit year, such as 2016
%y    Two-digit year, such as 16
%H    Hour, in 24-hour format (00 to 23)
%I    Hour, in 12-hour format (01 to 12)
%p    AM or PM
%M    Minutes (00 to 59)
%S    Seconds (00 to 61)
```

Converting a string to a datetime object

```
new_years = dt.strptime('1/1/2017', '%m/%d/%Y')
```

Converting a datetime object to a string

```
ny_string = dt.strftime(new_years, '%B %d, %Y')
print(ny_string)
```

Plotting high temperatures

The following code creates a list of dates and a corresponding list of high temperatures. It then plots the high temperatures, with the date labels displayed in a specific format.

```
from datetime import datetime as dt

import matplotlib.pyplot as plt
from matplotlib import dates as mdates

dates = [
    dt(2016, 6, 21), dt(2016, 6, 22),
    dt(2016, 6, 23), dt(2016, 6, 24),
    ]

highs = [57, 68, 64, 59]

fig = plt.figure(dpi=128, figsize=(10,6))
plt.plot(dates, highs, c='red')
plt.title("Daily High Temps", fontsize=24)
plt.ylabel("Temp (F)", fontsize=16)

x_axis = plt.axes().get_xaxis()
x_axis.set_major_formatter(
        mdates.DateFormatter('%B %d %Y')
        )
fig.autofmt_xdate()

plt.show()
```

Multiple plots in one figure

You can include as many individual graphs in one figure as you want. This is useful, for example, when comparing related datasets.

Sharing an x-axis

The following code plots a set of squares and a set of cubes on two separate graphs that share a common x-axis.

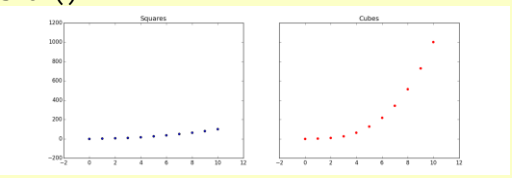
The `plt.subplots()` function returns a figure object and a tuple of axes. Each set of axes corresponds to a separate plot in the figure. The first two arguments control the number of rows and columns generated in the figure.

```
import matplotlib.pyplot as plt

x_vals = list(range(11))
squares = [x**2 for x in x_vals]
cubes = [x**3 for x in x_vals]

fig, axarr = plt.subplots(2, 1, sharex=True)

axarr[0].scatter(x_vals, squares)
axarr[0].set_title('Squares')

axarr[1].scatter(x_vals, cubes, c='red')
axarr[1].set_title('Cubes')

plt.show()
```

Sharing a y-axis

To share a y-axis, we use the `sharey=True` argument.

```
import matplotlib.pyplot as plt

x_vals = list(range(11))
squares = [x**2 for x in x_vals]
cubes = [x**3 for x in x_vals]

fig, axarr = plt.subplots(1, 2, sharey=True)

axarr[0].scatter(x_vals, squares)
axarr[0].set_title('Squares')

axarr[1].scatter(x_vals, cubes, c='red')
axarr[1].set_title('Cubes')

plt.show()
```



More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Pygal

What is Pygal?

Data visualization involves exploring data through visual representations. Pygal helps you make visually appealing representations of the data you're working with. Pygal is particularly well suited for visualizations that will be presented online, because it supports interactive elements.

Installing Pygal

Pygal can be installed using pip.

Pygal on Linux and OS X

```
$ pip install --user pygal
```

Pygal on Windows

```
> python -m pip install --user pygal
```

Line graphs, scatter plots, and bar graphs

To make a plot with Pygal, you specify the kind of plot and then add the data.

Making a line graph

To view the output, open the file squares.svg in a browser.

```
import pygal

x_values = [0, 1, 2, 3, 4, 5]
squares = [0, 1, 4, 9, 16, 25]

chart = pygal.Line()
chart.force_uri_protocol = 'http'
chart.add('x^2', squares)
chart.render_to_file('squares.svg')
```

Adding labels and a title

```
--snip--
chart = pygal.Line()
chart.force_uri_protocol = 'http'
chart.title = "Squares"
chart.x_labels = x_values
chart.x_title = "Value"
chart.y_title = "Square of Value"
chart.add('x^2', squares)
chart.render_to_file('squares.svg')
```

Line graphs, scatter plots, and bar graphs (cont.)

Making a scatter plot

The data for a scatter plot needs to be a list containing tuples of the form (x, y). The `stroke=False` argument tells Pygal to make an XY chart with no line connecting the points.

```
import pygal

squares = [
    (0, 0), (1, 1), (2, 4), (3, 9),
    (4, 16), (5, 25),
    ]

chart = pygal.XY(stroke=False)
chart.force_uri_protocol = 'http'
chart.add('x^2', squares)
chart.render_to_file('squares.svg')
```

Using a list comprehension for a scatter plot

A list comprehension can be used to effficiently make a dataset for a scatter plot.

```
squares = [(x, x**2) for x in range(1000)]
```

Making a bar graph

A bar graph requires a list of values for the bar sizes. To label the bars, pass a list of the same length to `x_labels`.

```
import pygal

outcomes = [1, 2, 3, 4, 5, 6]
frequencies = [18, 16, 18, 17, 18, 13]

chart = pygal.Bar()
chart.force_uri_protocol = 'http'
chart.x_labels = outcomes
chart.add('D6', frequencies)
chart.render_to_file('rolling_dice.svg')
```

Making a bar graph from a dictionary

Since each bar needs a label and a value, a dictionary is a great way to store the data for a bar graph. The keys are used as the labels along the x-axis, and the values are used to determine the height of each bar.

```
import pygal

results = {
    1:18, 2:16, 3:18,
    4:17, 5:18, 6:13,
    }

chart = pygal.Bar()
chart.force_uri_protocol = 'http'
chart.x_labels = results.keys()
chart.add('D6', results.values())
chart.render_to_file('rolling_dice.svg')
```

Multiple plots

You can add as much data as you want when making a visualization.

Plotting squares and cubes

```
import pygal

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

chart = pygal.Line()
chart.force_uri_protocol = 'http'
chart.title = "Squares and Cubes"
chart.x_labels = x_values

chart.add('Squares', squares)
chart.add('Cubes', cubes)
chart.render_to_file('squares_cubes.svg')
```

Filling the area under a data series

Pygal allows you to fill the area under or over each series of data. The default is to fill from the x-axis up, but you can fill from any horizontal line using the `zero` argument.

```
chart = pygal.Line(fill=True, zero=0)
```

Online resources

The documentation for Pygal is available at http://www.pygal.org/.

Enabling interactive features

If you're viewing svg output in a browser, Pygal needs to render the output file in a specific way. The `force_uri_protocol` attribute for chart objects needs to be set to `'http'`.

Styling plots

Pygal lets you customize many elements of a plot. There are some excellent default themes, and many options for styling individual plot elements.

Using built-in styles

To use built-in styles, import the style and make an instance of the style class. Then pass the style object with the `style` argument when you make the chart object.

```
import pygal
from pygal.style import LightGreenStyle

x_values = list(range(11))
squares = [x**2 for x in x_values]
cubes = [x**3 for x in x_values]

chart_style = LightGreenStyle()
chart = pygal.Line(style=chart_style)
chart.force_uri_protocol = 'http'
chart.title = "Squares and Cubes"
chart.x_labels = x_values

chart.add('Squares', squares)
chart.add('Cubes', cubes)
chart.render_to_file('squares_cubes.svg')
```

Parametric built-in styles

Some built-in styles accept a custom color, then generate a theme based on that color.

```
from pygal.style import LightenStyle

--snip--
chart_style = LightenStyle('#336688')
chart = pygal.Line(style=chart_style)
--snip--
```

Customizing individual style properties

Style objects have a number of properties you can set individually.

```
chart_style = LightenStyle('#336688')
chart_style.plot_background = '#CCCCCC'
chart_style.major_label_font_size = 20
chart_style.label_font_size = 16
--snip--
```

Custom style class

You can start with a bare style class, and then set only the properties you care about.

```
chart_style = Style()
chart_style.colors = [
    '#CCCCCC', '#AAAAAA', '#888888']
chart_style.plot_background = '#EEEEEE'

chart = pygal.Line(style=chart_style)
--snip--
```

Styling plots (cont.)

Configuration settings

Some settings are controlled by a `Config` object.

```
my_config = pygal.Config()
my_config.show_y_guides = False
my_config.width = 1000
my_config.dots_size = 5

chart = pygal.Line(config=my_config)
--snip--
```

Styling series

You can give each series on a chart different style settings.

```
chart.add('Squares', squares, dots_size=2)
chart.add('Cubes', cubes, dots_size=3)
```

Styling individual data points

You can style individual data points as well. To do so, write a dictionary for each data point you want to customize. A `'value'` key is required, and other properies are optional.

```
import pygal

repos = [
    {
        'value': 20506,
        'color': '#3333CC',
        'xlink': 'http://djangoproject.com/',
    },
    20054,
    12607,
    11827,
    ]

chart = pygal.Bar()
chart.force_uri_protocol = 'http'
chart.x_labels = [
    'django', 'requests', 'scikit-learn',
    'tornado',
    ]
chart.y_title = 'Stars'
chart.add('Python Repos', repos)
chart.render_to_file('python_repos.svg')
```

Plotting global datasets

Pygal can generate world maps, and you can add any data you want to these maps. Data is indicated by coloring, by labels, and by tooltips that show data when users hover over each country on the map.

Installing the world map module

The world map module is not included by default in Pygal 2.0. It can be installed with pip:

```
$ pip install --user pygal_maps_world
```

Making a world map

The following code makes a simple world map showing the countries of North America.

```
from pygal.maps.world import World

wm = World()
wm.force_uri_protocol = 'http'
wm.title = 'North America'
wm.add('North America', ['ca', 'mx', 'us'])

wm.render_to_file('north_america.svg')
```

Showing all the country codes

In order to make maps, you need to know Pygal's country codes. The following example will print an alphabetical list of each country and its code.

```
from pygal.maps.world import COUNTRIES

for code in sorted(COUNTRIES.keys()):
    print(code, COUNTRIES[code])
```

Plotting numerical data on a world map

To plot numerical data on a map, pass a dictionary to `add()` instead of a list.

```
from pygal.maps.world import World

populations = {
    'ca': 34126000,
    'us': 309349000,
    'mx': 113423000,
    }

wm = World()
wm.force_uri_protocol = 'http'
wm.title = 'Population of North America'
wm.add('North America', populations)

wm.render_to_file('na_populations.svg')
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Django

What is Django?

Django is a web framework which helps you build interactive websites using Python. With Django you define the kind of data your site needs to work with, and you define the ways your users can work with that data.

Installing Django

It's usualy best to install Django to a virtual environment, where your project can be isolated from your other Python projects. Most commands assume you're working in an active virtual environment.

Create a virtual environment

```
$ python -m venv ll_env
```

Activate the environment (Linux and OS X)

```
$ source ll_env/bin/activate
```

Activate the environment (Windows)

```
> ll_env\Scripts\activate
```

Install Django to the active environment

```
(ll_env)$ pip install Django
```

Creating a project

To start a project we'll create a new project, create a database, and start a development server.

Create a new project

```
$ django-admin.py startproject learning_log .
```

Create a database

```
$ python manage.py migrate
```

View the project

After issuing this command, you can view the project at http://localhost:8000/.

```
$ python manage.py runserver
```

Create a new app

A Django project is made up of one or more apps.

```
$ python manage.py startapp learning_logs
```

Working with models

The data in a Django project is structured as a set of models.

Defining a model

To define the models for your app, modify the file models.py that was created in your app's folder. The __str__() method tells Django how to represent data objects based on this model.

```
from django.db import models

class Topic(models.Model):
    """A topic the user is learning about."""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(
            auto_now_add=True)

    def __str__(self):
        return self.text
```

Activating a model

To use a model the app must be added to the tuple `INSTALLED_APPS`*, which is stored in the project's settings.py file.*

```
INSTALLED_APPS = (
    --snip--
    'django.contrib.staticfiles',

    # My apps
    'learning_logs',
    )
```

Migrating the database

The database needs to be modified to store the kind of data that the model represents.

```
$ python manage.py makemigrations learning_logs
$ python manage.py migrate
```

Creating a superuser

A superuser is a user account that has access to all aspects of the project.

```
$ python manage.py createsuperuser
```

Registering a model

You can register your models with Django's admin site, which makes it easier to work with the data in your project. To do this, modify the app's admin.py file. View the admin site at http://localhost:8000/admin/.

```
from django.contrib import admin

from learning_logs.models import Topic

admin.site.register(Topic)
```

Building a simple home page

Users interact with a project through web pages, and a project's home page can start out as a simple page with no data. A page usually needs a URL, a view, and a template.

Mapping a project's URLs

The project's main urls.py file tells Django where to find the urls.py files associated with each app in the project.

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'', include('learning_logs.urls',
            namespace='learning_logs')),
    ]
```

Mapping an app's URLs

An app's urls.py file tells Django which view to use for each URL in the app. You'll need to make this file yourself, and save it in the app's folder.

```
from django.conf.urls import url

from . import views

urlpatterns = [
    url(r'^$', views.index, name='index'),
    ]
```

Writing a simple view

A view takes information from a request and sends data to the browser, often through a template. View functions are stored in an app's views.py file. This simple view function doesn't pull in any data, but it uses the template index.html to render the home page.

```
from django.shortcuts import render

def index(request):
    """The home page for Learning Log."""
    return render(request,
            'learning_logs/index.html')
```

Online resources

The documentation for Django is available at http://docs.djangoproject.com/. The Django documentation is thorough and user-friendly, so check it out!

Building a simple home page (cont.)

Writing a simple template

A template sets up the structure for a page. It's a mix of html and template code, which is like Python but not as powerful. Make a folder called templates inside the project folder. Inside the templates folder make another folder with the same name as the app. This is where the template files should be saved.

```
<p>Learning Log</p>

<p>Learning Log helps you keep track of your
learning, for any topic you're learning
about.</p>
```

Template inheritance

Many elements of a web page are repeated on every page in the site, or every page in a section of the site. By writing one parent template for the site, and one for each section, you can easily modify the look and feel of your entire site.

The parent template

The parent template defines the elements common to a set of pages, and defines blocks that will be filled by individual pages.

```
<p>
  <a href="{% url 'learning_logs:index' %}">
    Learning Log
  </a>
</p>

{% block content %}{% endblock content %}
```

The child template

The child template uses the `{% extends %}` template tag to pull in the structure of the parent template. It then defines the content for any blocks defined in the parent template.

```
{% extends 'learning_logs/base.html' %}

{% block content %}
  <p>
    Learning Log helps you keep track
    of your learning, for any topic you're
    learning about.
  </p>
{% endblock content %}
```

Template indentation

Python code is usually indented by four spaces. In templates you'll often see two spaces used for indentation, because elements tend to be nested more deeply in templates.

Another model

A new model can use an existing model. The ForeignKey attribute establishes a connection between instances of the two related models. Make sure to migrate the database after adding a new model to your app.

Defining a model with a foreign key

```
class Entry(models.Model):
    """Learning log entries for a topic."""
    topic = models.ForeignKey(Topic)
    text = models.TextField()
    date_added = models.DateTimeField(
            auto_now_add=True)

    def __str__(self):
        return self.text[:50] + "..."
```

Building a page with data

Most pages in a project need to present data that's specific to the current user.

URL parameters

A URL often needs to accept a parameter telling it which data to access from the database. The second URL pattern shown here looks for the ID of a specific topic and stores it in the parameter `topic_id`.

```
urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^topics/(?P<topic_id>\d+)/$',
            views.topic, name='topic'),
    ]
```

Using data in a view

The view uses a parameter from the URL to pull the correct data from the database. In this example the view is sending a `context` dictionary to the template, containing data that should be displayed on the page.

```
def topic(request, topic_id):
    """Show a topic and all its entries."""
    topic = Topics.objects.get(id=topic_id)
    entries = topic.entry_set.order_by(
            '-date_added')
    context = {
        'topic': topic,
        'entries': entries,
        }
    return render(request,
        'learning_logs/topic.html', context)
```

Restarting the development server

If you make a change to your project and the change doesn't seem to have any effect, try restarting the server:

```
$ python manage.py runserver
```

Building a page with data (cont.)

Using data in a template

The data in the view function's `context` dictionary is available within the template. This data is accessed using template variables, which are indicated by doubled curly braces.

The vertical line after a template variable indicates a filter. In this case a filter called `date` formats date objects, and the filter `linebreaks` renders paragraphs properly on a web page.

```
{% extends 'learning_logs/base.html' %}

{% block content %}

  <p>Topic: {{ topic }}</p>

  <p>Entries:</p>
  <ul>
  {% for entry in entries %}
    <li>
    <p>
      {{ entry.date_added|date:'M d, Y H:i' }}
    </p>
    <p>
      {{ entry.text|linebreaks }}
    </p>
    </li>
  {% empty %}
    <li>There are no entries yet.</li>
  {% endfor %}
  </ul>

{% endblock content %}
```

The Django shell

You can explore the data in your project from the command line. This is helpful for developing queries and testing code snippets.

Start a shell session

```
$ python manage.py shell
```

Access data from the project

```
>>> from learning_logs.models import Topic
>>> Topic.objects.all()
[<Topic: Chess>, <Topic: Rock Climbing>]
>>> topic = Topic.objects.get(id=1)
>>> topic.text
'Chess'
```

More cheat sheets available at
ehmatthes.github.io/pcc/

Beginner's Python Cheat Sheet — Django, Part 2

Users and forms

Most web applications need to let users create accounts. This lets users create and work with their own data. Some of this data may be private, and some may be public. Django's forms allow users to enter and modify their data.

User accounts

User accounts are handled by a dedicated app called `users`*. Users need to be able to register, log in, and log out. Django automates much of this work for you.*

Making a users app

After making the app, be sure to add `'users'` *to* `INSTALLED_APPS` *in the project's settings.py file.*

```
$ python manage.py startapp users
```

Including URLS for the users app

Add a line to the project's urls.py file so the `users` *app's URLs are included in the project.*

```
urlpatterns = [
    url(r'^admin/', include(admin.site.urls)),
    url(r'^users/', include('users.urls',
            namespace='users')),
    url(r'', include('learning_logs.urls',
            namespace='learning_logs')),
    ]
```

Using forms in Django

There are a number of ways to create forms and work with them. You can use Django's defaults, or completely customize your forms. For a simple way to let users enter data based on your models, use a `ModelForm`*. This creates a form that allows users to enter data that will populate the fields on a model.*

The register view on the back of this sheet shows a simple approach to form processing. If the view doesn't receive data from a form, it responds with a blank form. If it receives `POST` *data from a form, it validates the data and then saves it to the database.*

User accounts (cont.)

Defining the URLs

Users will need to be able to log in, log out, and register. Make a new urls.py file in the `users` *app folder. The login view is a default view provided by Django.*

```
from django.conf.urls import url
from django.contrib.auth.views import login

from . import views

urlpatterns = [
    url(r'^login/$', login,
        {'template_name': 'users/login.html'},
        name='login'),
    url(r'^logout/$', views.logout_view,
        name='logout'),
    url(r'^register/$', views.register,
        name='register'),
    ]
```

The login template

The login view is provided by default, but you need to provide your own login template. The template shown here displays a simple login form, and provides basic error messages. Make a templates folder in the users folder, and then make a users folder in the templates folder. Save this file as login.html.

The tag `{% csrf_token %}` *helps prevent a common type of attack with forms. The* `{{ form.as_p }}` *element displays the default login form in paragraph format. The* `<input>` *element named* `next` *redirects the user to the home page after a successful login.*

```
{% extends "learning_logs/base.html" %}

{% block content %}
  {% if form.errors %}
  <p>
    Your username and password didn't match.
    Please try again.
  </p>
  {% endif %}

  <form method="post"
      action="{% url 'users:login' %}">
    {% csrf token %}
    {{ form.as_p }}
    <button name="submit">log in</button>

    <input type="hidden" name="next"
      value="{% url 'learning_logs:index' %}"/>
  </form>

{% endblock content %}
```

User accounts (cont.)

Showing the current login status

You can modify the base.html template to show whether the user is currently logged in, and to provide a link to the login and logout pages. Django makes a `user` *object available to every template, and this template takes advantage of this object.*

The `user.is_authenticated` *tag allows you to serve specific content to users depending on whether they have logged in or not. The* `{{ user.username }}` *property allows you to greet users who have logged in. Users who haven't logged in see links to register or log in.*

```
<p>
  <a href="{% url 'learning_logs:index' %}">
    Learning Log
  </a>
  {% if user.is_authenticated %}
    Hello, {{ user.username }}.
    <a href="{% url 'users:logout' %}">
      log out
    </a>
  {% else %}
    <a href="{% url 'users:register' %}">
      register
    </a> -
    <a href="{% url 'users:login' %}">
      log in
    </a>
  {% endif %}
</p>

{% block content %}{% endblock content %}
```

The logout view

The `logout_view()` *function uses Django's* `logout()` *function and then redirects the user back to the home page. Since there is no logout page, there is no logout template. Make sure to write this code in the views.py file that's stored in the users app folder.*

```
from django.http import HttpResponseRedirect
from django.core.urlresolvers import reverse
from django.contrib.auth import logout

def logout_view(request):
    """Log the user out."""
    logout(request)
    return HttpResponseRedirect(
        reverse('learning_logs:index'))
```

User accounts (cont.)

The register view

The register view needs to display a blank registration form when the page is first requested, and then process completed registration forms. A successful registration logs the user in and redirects to the home page.

```
from django.contrib.auth import login
from django.contrib.auth import authenticate
from django.contrib.auth.forms import \
        UserCreationForm

def register(request):
    """Register a new user."""
    if request.method != 'POST':
        # Show blank registration form.
        form = UserCreationForm()
    else:
        # Process completed form.
        form = UserCreationForm(
                data=request.POST)

        if form.is_valid():
            new_user = form.save()
            # Log in, redirect to home page.
            pw = request.POST['password1']
            authenticated_user = authenticate(
                username=new_user.username,
                password=pw
            )
            login(request, authenticated_user)
            return HttpResponseRedirect(
                reverse('learning_logs:index'))

    context = {'form': form}
    return render(request,
        'users/register.html', context)
```

Styling your project

The django-bootstrap3 app allows you to use the Bootstrap library to make your project look visually appealing. The app provides tags that you can use in your templates to style individual elements on a page. Learn more at http://django-bootstrap3.readthedocs.io/.

Deploying your project

Heroku lets you push your project to a live server, making it available to anyone with an internet connection. Heroku offers a free service level, which lets you learn the deployment process without any commitment. You'll need to install a set of heroku tools, and use git to track the state of your project. See http://devcenter.heroku.com/, and click on the Python link.

User accounts (cont.)

The register template

The register template displays the registration form in paragraph formats.

```
{% extends 'learning_logs/base.html' %}

{% block content %}

  <form method='post'
      action="{% url 'users:register' %}">

    {% csrf_token %}
    {{ form.as_p }}

    <button name='submit'>register</button>
    <input type='hidden' name='next'
      value="{% url 'learning_logs:index' %}"/>

  </form>

{% endblock content %}
```

Connecting data to users

Users will have data that belongs to them. Any model that should be connected directly to a user needs a field connecting instances of the model to a specific user.

Making a topic belong to a user

Only the highest-level data in a hierarchy needs to be directly connected to a user. To do this import the `User` model, and add it as a foreign key on the data model.

After modifying the model you'll need to migrate the database. You'll need to choose a user ID to connect each existing instance to.

```
from django.db import models
from django.contrib.auth.models import User

class Topic(models.Model):
    """A topic the user is learning about."""
    text = models.CharField(max_length=200)
    date_added = models.DateTimeField(
            auto_now_add=True)
    owner = models.ForeignKey(User)

    def __str__(self):
        return self.text
```

Querying data for the current user

In a view, the request object has a `user` attribute. You can use this attribute to query for the user's data. The `filter()` function then pulls the data that belongs to the current user.

```
topics = Topic.objects.filter(
                owner=request.user)
```

Connecting data to users (cont.)

Restricting access to logged-in users

Some pages are only relevant to registered users. The views for these pages can be protected by the `@login_required` decorator. Any view with this decorator will automatically redirect non-logged in users to an appropriate page. Here's an example views.py file.

```
from django.contrib.auth.decorators import /
        login_required
--snip--

@login_required
def topic(request, topic_id):
    """Show a topic and all its entries."""
```

Setting the redirect URL

The `@login_required` decorator sends unauthorized users to the login page. Add the following line to your project's settings.py file so Django will know how to find your login page.

```
LOGIN_URL = '/users/login/'
```

Preventing inadvertent access

Some pages serve data based on a parameter in the URL. You can check that the current user owns the requested data, and return a 404 error if they don't. Here's an example view.

```
from django.http import Http404

--snip--
def topic(request, topic_id):
    """Show a topic and all its entries."""
    topic = Topics.objects.get(id=topic_id)
    if topic.owner != request.user:
        raise Http404
    --snip--
```

Using a form to edit data

If you provide some initial data, Django generates a form with the user's existing data. Users can then modify and save their data.

Creating a form with initial data

The instance parameter allows you to specify initial data for a form.

```
form = EntryForm(instance=entry)
```

Modifying data before saving

The argument `commit=False` allows you to make changes before writing data to the database.

```
new_topic = form.save(commit=False)
new_topic.owner = request.user
new_topic.save()
```

More cheat sheets available at
ehmatthes.github.io/pcc/

SQL CHEAT SHEET http://www.sqltutorial.org

QUERYING DATA FROM A TABLE

```
SELECT c1, c2 FROM t;
```
Query data in columns c1, c2 from a table

```
SELECT * FROM t;
```
Query all rows and columns from a table

```
SELECT c1, c2 FROM t
WHERE condition;
```
Query data and filter rows with a condition

```
SELECT DISTINCT c1 FROM t
WHERE condition;
```
Query distinct rows from a table

```
SELECT c1, c2 FROM t
ORDER BY c1 ASC [DESC];
```
Sort the result set in ascending or descending order

```
SELECT c1, c2 FROM t
ORDER BY c1
LIMIT n OFFSET offset;
```
Skip *offset* of rows and return the next n rows

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1;
```
Group rows using an aggregate function

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1
HAVING condition;
```
Filter groups using HAVING clause

QUERYING FROM MULTIPLE TABLES

```
SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
```
Inner join t1 and t2

```
SELECT c1, c2
FROM t1
LEFT JOIN t2 ON condition;
```
Left join t1 and t1

```
SELECT c1, c2
FROM t1
RIGHT JOIN t2 ON condition;
```
Right join t1 and t2

```
SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
```
Perform full outer join

```
SELECT c1, c2
FROM t1
CROSS JOIN t2;
```
Produce a Cartesian product of rows in tables

```
SELECT c1, c2
FROM t1, t2;
```
Another way to perform cross join

```
SELECT c1, c2
FROM t1 A
INNER JOIN t2 B ON condition;
```
Join t1 to itself using INNER JOIN clause

USING SQL OPERATORS

```
SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
```
Combine rows from two queries

```
SELECT c1, c2 FROM t1
INTERSECT
SELECT c1, c2 FROM t2;
```
Return the intersection of two queries

```
SELECT c1, c2 FROM t1
MINUS
SELECT c1, c2 FROM t2;
```
Subtract a result set from another result set

```
SELECT c1, c2 FROM t1
WHERE c1 [NOT] LIKE pattern;
```
Query rows using pattern matching %, _

```
SELECT c1, c2 FROM t
WHERE c1 [NOT] IN value_list;
```
Query rows in a list

```
SELECT c1, c2 FROM t
WHERE  c1 BETWEEN low AND high;
```
Query rows between two values

```
SELECT c1, c2 FROM t
WHERE  c1 IS [NOT] NULL;
```
Check if values in a table is NULL or not

SQL CHEAT SHEET http://www.sqltutorial.org

MANAGING TABLES

```
CREATE TABLE t (
    id INT PRIMARY KEY,
    name VARCHAR NOT NULL,
    price INT DEFAULT 0
);
```
Create a new table with three columns

```
DROP TABLE t ;
```
Delete the table from the database

```
ALTER TABLE t ADD column;
```
Add a new column to the table

```
ALTER TABLE t DROP COLUMN c ;
```
Drop column c from the table

```
ALTER TABLE t ADD constraint;
```
Add a constraint

```
ALTER TABLE t DROP constraint;
```
Drop a constraint

```
ALTER TABLE t1 RENAME TO t2;
```
Rename a table from t1 to t2

```
ALTER TABLE t1 RENAME c1 TO c2 ;
```
Rename column c1 to c2

```
TRUNCATE TABLE t;
```
Remove all data in a table

USING SQL CONSTRAINTS

```
CREATE TABLE t(
   c1 INT, c2 INT, c3 VARCHAR,
   PRIMARY KEY (c1,c2)
);
```
Set c1 and c2 as a primary key

```
CREATE TABLE t1(
   c1 INT PRIMARY KEY,
   c2 INT,
   FOREIGN KEY (c2) REFERENCES t2(c2)
);
```
Set c2 column as a foreign key

```
CREATE TABLE t(
   c1 INT, c1 INT,
   UNIQUE(c2,c3)
);
```
Make the values in c1 and c2 unique

```
CREATE TABLE t(
  c1 INT, c2 INT,
  CHECK(c1> 0 AND c1 >= c2)
);
```
Ensure c1 > 0 and values in c1 >= c2

```
CREATE TABLE t(
    c1 INT PRIMARY KEY,
    c2 VARCHAR NOT NULL
);
```
Set values in c2 column not NULL

MODIFYING DATA

```
INSERT INTO t(column_list)
VALUES(value_list);
```
Insert one row into a table

```
INSERT INTO t(column_list)
VALUES (value_list),
       (value_list), ....;
```
Insert multiple rows into a table

```
INSERT INTO t1(column_list)
SELECT column_list
FROM t2;
```
Insert rows from t2 into t1

```
UPDATE t
SET c1 = new_value;
```
Update new value in the column c1 for all rows

```
UPDATE t
SET c1 = new_value,
    c2 = new_value
WHERE condition;
```
Update values in the column c1, c2 that match the condition

```
DELETE FROM t;
```
Delete all data in a table

```
DELETE FROM t
WHERE condition;
```
Delete subset of rows in a table

SQL CHEAT SHEET http://www.sqltutorial.org

MANAGING VIEWS

```
CREATE VIEW v(c1,c2)
AS
SELECT c1, c2
FROM t;
```
Create a new view that consists of c1 and c2

```
CREATE VIEW v(c1,c2)
AS
SELECT c1, c2
FROM t;
WITH [CASCADED | LOCAL] CHECK OPTION;
```
Create a new view with check option

```
CREATE RECURSIVE VIEW v
AS
select-statement -- anchor part
UNION [ALL]
select-statement; -- recursive part
```
Create a recursive view

```
CREATE TEMPORARY VIEW v
AS
SELECT c1, c2
FROM t;
```
Create a temporary view

```
DROP VIEW view_name;
```
Delete a view

MANAGING INDEXES

```
CREATE INDEX idx_name
ON t(c1,c2);
```
Create an index on c1 and c2 of the table t

```
CREATE UNIQUE INDEX idx_name
ON t(c3,c4);
```
Create a unique index on c3, c4 of the table t

```
DROP INDEX idx_name;
```
Drop an index

SQL AGGREGATE FUNCTIONS

AVG returns the average of a list

COUNT returns the number of elements of a list

SUM returns the total of a list

MAX returns the maximum value in a list

MIN returns the minimum value in a list

MANAGING TRIGGERS

```
CREATE OR MODIFY TRIGGER trigger_name
WHEN EVENT
ON table_name TRIGGER_TYPE
EXECUTE stored_procedure;
```
Create or modify a trigger

WHEN
- **BEFORE** – invoke before the event occurs
- **AFTER** – invoke after the event occurs

EVENT
- **INSERT** – invoke for INSERT
- **UPDATE** – invoke for UPDATE
- **DELETE** – invoke for DELETE

TRIGGER_TYPE
- **FOR EACH ROW**
- **FOR EACH STATEMENT**

```
CREATE TRIGGER before_insert_person
BEFORE INSERT
ON person FOR EACH ROW
EXECUTE stored_procedure;
```
Create a trigger invoked before a new row is inserted into the person table

```
DROP TRIGGER trigger_name;
```
Delete a specific trigger

VIP Cheatsheet: Supervised Learning

Afshine Amidi and Shervine Amidi

September 9, 2018

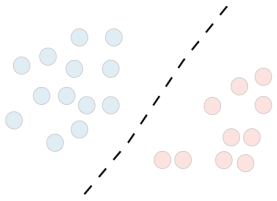
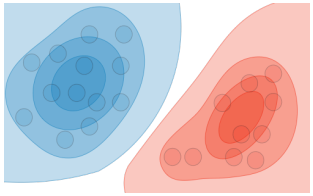
Introduction to Supervised Learning

Given a set of data points $\{x^{(1)},...,x^{(m)}\}$ associated to a set of outcomes $\{y^{(1)},...,y^{(m)}\}$, we want to build a classifier that learns how to predict y from x.

❒ **Type of prediction** – The different types of predictive models are summed up in the table below:

	Regression	**Classifier**
Outcome	Continuous	Class
Examples	Linear regression	Logistic regression, SVM, Naive Bayes

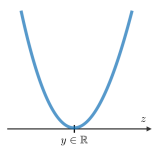
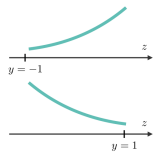
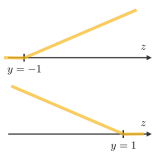
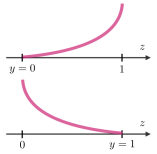
❒ **Type of model** – The different models are summed up in the table below:

	Discriminative model	**Generative model**
Goal	Directly estimate $P(y\|x)$	Estimate $P(x\|y)$ to deduce $P(y\|x)$
What's learned	Decision boundary	Probability distributions of the data
Illustration		
Examples	Regressions, SVMs	GDA, Naive Bayes

Notations and general concepts

❒ **Hypothesis** – The hypothesis is noted h_θ and is the model that we choose. For a given input data $x^{(i)}$, the model prediction output is $h_\theta(x^{(i)})$.

❒ **Loss function** – A loss function is a function $L:(z,y)\in\mathbb{R}\times Y\longmapsto L(z,y)\in\mathbb{R}$ that takes as inputs the predicted value z corresponding to the real data value y and outputs how different they are. The common loss functions are summed up in the table below:

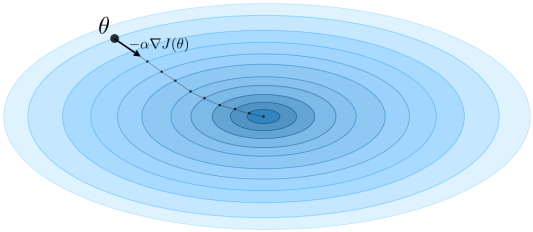
Least squared	**Logistic**	**Hinge**	**Cross-entropy**
$\frac{1}{2}(y-z)^2$	$\log(1+\exp(-yz))$	$\max(0,1-yz)$	$-\left[y\log(z)+(1-y)\log(1-z)\right]$
			
Linear regression	Logistic regression	SVM	Neural Network

❒ **Cost function** – The cost function J is commonly used to assess the performance of a model, and is defined with the loss function L as follows:

$$J(\theta)=\sum_{i=1}^{m}L(h_\theta(x^{(i)}),y^{(i)})$$

❒ **Gradient descent** – By noting $\alpha\in\mathbb{R}$ the learning rate, the update rule for gradient descent is expressed with the learning rate and the cost function J as follows:

$$\theta\longleftarrow\theta-\alpha\nabla J(\theta)$$



Remark: Stochastic gradient descent (SGD) is updating the parameter based on each training example, and batch gradient descent is on a batch of training examples.

❒ **Likelihood** – The likelihood of a model $L(\theta)$ given parameters θ is used to find the optimal parameters θ through maximizing the likelihood. In practice, we use the log-likelihood $\ell(\theta)=\log(L(\theta))$ which is easier to optimize. We have:

$$\theta^{\text{opt}}=\arg\max_\theta L(\theta)$$

❒ **Newton's algorithm** – The Newton's algorithm is a numerical method that finds θ such that $\ell'(\theta)=0$. Its update rule is as follows:

$$\theta\leftarrow\theta-\frac{\ell'(\theta)}{\ell''(\theta)}$$

Remark: the multidimensional generalization, also known as the Newton-Raphson method, has the following update rule:

$$\theta\leftarrow\theta-\left(\nabla^2_\theta\ell(\theta)\right)^{-1}\nabla_\theta\ell(\theta)$$

Linear regression

We assume here that $y|x;\theta \sim \mathcal{N}(\mu,\sigma^2)$

❒ **Normal equations** – By noting X the matrix design, the value of θ that minimizes the cost function is a closed-form solution such that:

$$\boxed{\theta = (X^TX)^{-1}X^Ty}$$

❒ **LMS algorithm** – By noting α the learning rate, the update rule of the Least Mean Squares (LMS) algorithm for a training set of m data points, which is also known as the Widrow-Hoff learning rule, is as follows:

$$\boxed{\forall j, \quad \theta_j \leftarrow \theta_j + \alpha \sum_{i=1}^{m} \left[y^{(i)} - h_\theta(x^{(i)})\right] x_j^{(i)}}$$

Remark: the update rule is a particular case of the gradient ascent.

❒ **LWR** – Locally Weighted Regression, also known as LWR, is a variant of linear regression that weights each training example in its cost function by $w^{(i)}(x)$, which is defined with parameter $\tau \in \mathbb{R}$ as:

$$\boxed{w^{(i)}(x) = \exp\left(-\frac{(x^{(i)}-x)^2}{2\tau^2}\right)}$$

Classification and logistic regression

❒ **Sigmoid function** – The sigmoid function g, also known as the logistic function, is defined as follows:

$$\forall z \in \mathbb{R}, \quad \boxed{g(z) = \frac{1}{1+e^{-z}} \in]0,1[}$$

❒ **Logistic regression** – We assume here that $y|x;\theta \sim \text{Bernoulli}(\phi)$. We have the following form:

$$\boxed{\phi = p(y=1|x;\theta) = \frac{1}{1+\exp(-\theta^Tx)} = g(\theta^Tx)}$$

Remark: there is no closed form solution for the case of logistic regressions.

❒ **Softmax regression** – A softmax regression, also called a multiclass logistic regression, is used to generalize logistic regression when there are more than 2 outcome classes. By convention, we set $\theta_K = 0$, which makes the Bernoulli parameter ϕ_i of each class i equal to:

$$\boxed{\phi_i = \frac{\exp(\theta_i^Tx)}{\sum_{j=1}^{K}\exp(\theta_j^Tx)}}$$

Generalized Linear Models

❒ **Exponential family** – A class of distributions is said to be in the exponential family if it can be written in terms of a natural parameter, also called the canonical parameter or link function, η, a sufficient statistic $T(y)$ and a log-partition function $a(\eta)$ as follows:

$$\boxed{p(y;\eta) = b(y)\exp(\eta T(y) - a(\eta))}$$

Remark: we will often have $T(y) = y$. Also, $\exp(-a(\eta))$ can be seen as a normalization parameter that will make sure that the probabilities sum to one.

Here are the most common exponential distributions summed up in the following table:

Distribution	η	$T(y)$	$a(\eta)$	$b(y)$
Bernoulli	$\log\left(\frac{\phi}{1-\phi}\right)$	y	$\log(1+\exp(\eta))$	1
Gaussian	μ	y	$\frac{\eta^2}{2}$	$\frac{1}{\sqrt{2\pi}}\exp\left(-\frac{y^2}{2}\right)$
Poisson	$\log(\lambda)$	y	e^η	$\frac{1}{y!}$
Geometric	$\log(1-\phi)$	y	$\log\left(\frac{e^\eta}{1-e^\eta}\right)$	1

❒ **Assumptions of GLMs** – Generalized Linear Models (GLM) aim at predicting a random variable y as a function fo $x \in \mathbb{R}^{n+1}$ and rely on the following 3 assumptions:

(1) $\boxed{y|x;\theta \sim \text{ExpFamily}(\eta)}$ (2) $\boxed{h_\theta(x) = E[y|x;\theta]}$ (3) $\boxed{\eta = \theta^Tx}$

Remark: ordinary least squares and logistic regression are special cases of generalized linear models.

Support Vector Machines

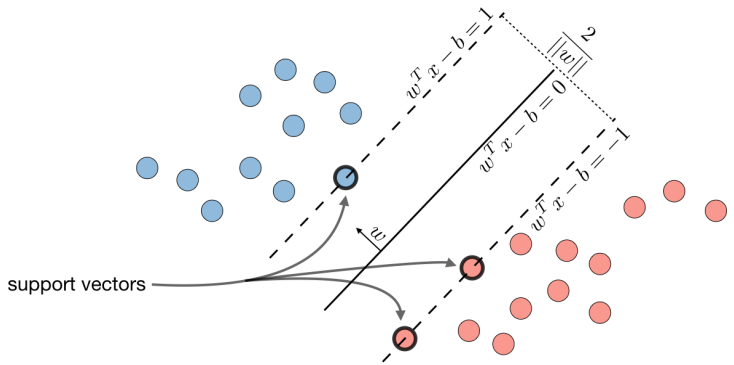
The goal of support vector machines is to find the line that maximizes the minimum distance to the line.

❒ **Optimal margin classifier** – The optimal margin classifier h is such that:

$$\boxed{h(x) = \text{sign}(w^Tx - b)}$$

where $(w,b) \in \mathbb{R}^n \times \mathbb{R}$ is the solution of the following optimization problem:

$$\boxed{\min \frac{1}{2}||w||^2} \quad \text{such that} \quad \boxed{y^{(i)}(w^Tx^{(i)} - b) \geqslant 1}$$



Remark: the line is defined as $\boxed{w^T x - b = 0}$.

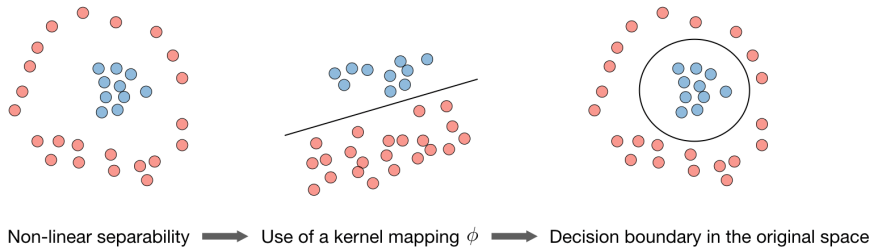
❒ **Hinge loss** – The hinge loss is used in the setting of SVMs and is defined as follows:

$$\boxed{L(z,y) = [1-yz]_+ = \max(0,1-yz)}$$

❒ **Kernel** – Given a feature mapping ϕ, we define the kernel K to be defined as:

$$\boxed{K(x,z) = \phi(x)^T \phi(z)}$$

In practice, the kernel K defined by $K(x,z) = \exp\left(-\frac{||x-z||^2}{2\sigma^2}\right)$ is called the Gaussian kernel and is commonly used.



Remark: we say that we use the "kernel trick" to compute the cost function using the kernel because we actually don't need to know the explicit mapping ϕ, which is often very complicated. Instead, only the values $K(x,z)$ are needed.

❒ **Lagrangian** – We define the Lagrangian $\mathcal{L}(w,b)$ as follows:

$$\boxed{\mathcal{L}(w,b) = f(w) + \sum_{i=1}^{l} \beta_i h_i(w)}$$

Remark: the coefficients β_i are called the Lagrange multipliers.

Generative Learning

A generative model first tries to learn how the data is generated by estimating $P(x|y)$, which we can then use to estimate $P(y|x)$ by using Bayes' rule.

Gaussian Discriminant Analysis

❒ **Setting** – The Gaussian Discriminant Analysis assumes that y and $x|y=0$ and $x|y=1$ are such that:

$$\boxed{y \sim \text{Bernoulli}(\phi)}$$

$$\boxed{x|y=0 \sim \mathcal{N}(\mu_0,\Sigma)} \quad \text{and} \quad \boxed{x|y=1 \sim \mathcal{N}(\mu_1,\Sigma)}$$

❒ **Estimation** – The following table sums up the estimates that we find when maximizing the likelihood:

$\widehat{\phi}$	$\widehat{\mu_j}$ $(j=0,1)$	$\widehat{\Sigma}$
$\frac{1}{m}\sum_{i=1}^{m} 1_{\{y^{(i)}=1\}}$	$\frac{\sum_{i=1}^{m} 1_{\{y^{(i)}=j\}} x^{(i)}}{\sum_{i=1}^{m} 1_{\{y^{(i)}=j\}}}$	$\frac{1}{m}\sum_{i=1}^{m}(x^{(i)}-\mu_{y^{(i)}})(x^{(i)}-\mu_{y^{(i)}})^T$

Naive Bayes

❒ **Assumption** – The Naive Bayes model supposes that the features of each data point are all independent:

$$\boxed{P(x|y) = P(x_1,x_2,...|y) = P(x_1|y)P(x_2|y)... = \prod_{i=1}^{n} P(x_i|y)}$$

❒ **Solutions** – Maximizing the log-likelihood gives the following solutions, with $k \in \{0,1\}$, $l \in [\![1,L]\!]$

$$\boxed{P(y=k) = \frac{1}{m} \times \#\{j|y^{(j)}=k\}} \quad \text{and} \quad \boxed{P(x_i=l|y=k) = \frac{\#\{j|y^{(j)}=k \text{ and } x_i^{(j)}=l\}}{\#\{j|y^{(j)}=k\}}}$$

Remark: Naive Bayes is widely used for text classification and spam detection.

Tree-based and ensemble methods

These methods can be used for both regression and classification problems.

❒ **CART** – Classification and Regression Trees (CART), commonly known as decision trees, can be represented as binary trees. They have the advantage to be very interpretable.

❒ **Random forest** – It is a tree-based technique that uses a high number of decision trees built out of randomly selected sets of features. Contrary to the simple decision tree, it is highly uninterpretable but its generally good performance makes it a popular algorithm.

Remark: random forests are a type of ensemble methods.

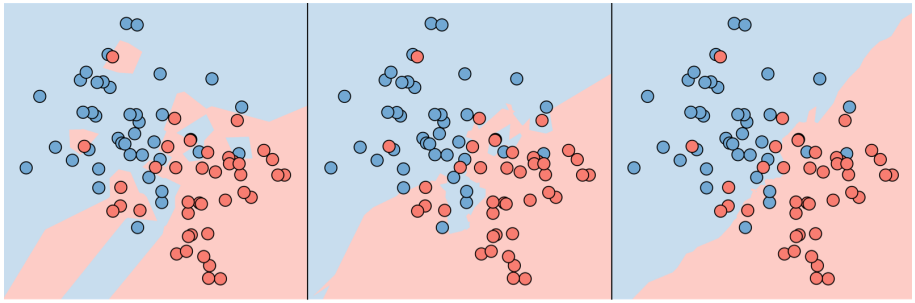
❒ **Boosting** – The idea of boosting methods is to combine several weak learners to form a stronger one. The main ones are summed up in the table below:

Adaptive boosting	Gradient boosting
- High weights are put on errors to improve at the next boosting step - Known as Adaboost	- Weak learners trained on remaining errors

Other non-parametric approaches

❒ **k-nearest neighbors** – The k-nearest neighbors algorithm, commonly known as k-NN, is a non-parametric approach where the response of a data point is determined by the nature of its k neighbors from the training set. It can be used in both classification and regression settings.

Remark: The higher the parameter k, the higher the bias, and the lower the parameter k, the higher the variance.

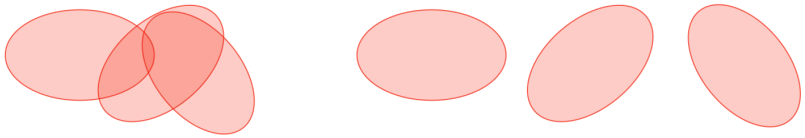


$k = 1$ $k = 3$ $k = 11$

Learning Theory

❒ **Union bound** – Let $A_1, ..., A_k$ be k events. We have:

$$\boxed{P(A_1 \cup ... \cup A_k) \leqslant P(A_1) + ... + P(A_k)}$$



$A_1 \cup A_2 \cup A_3$ A_1 A_2 A_3

❒ **Hoeffding inequality** – Let $Z_1, .., Z_m$ be m iid variables drawn from a Bernoulli distribution of parameter ϕ. Let $\widehat{\phi}$ be their sample mean and $\gamma > 0$ fixed. We have:

$$\boxed{P(|\phi - \widehat{\phi}| > \gamma) \leqslant 2\exp(-2\gamma^2 m)}$$

Remark: this inequality is also known as the Chernoff bound.

❒ **Training error** – For a given classifier h, we define the training error $\widehat{\epsilon}(h)$, also known as the empirical risk or empirical error, to be as follows:

$$\boxed{\widehat{\epsilon}(h) = \frac{1}{m}\sum_{i=1}^{m} 1_{\{h(x^{(i)}) \neq y^{(i)}\}}}$$

❒ **Probably Approximately Correct (PAC)** – PAC is a framework under which numerous results on learning theory were proved, and has the following set of assumptions:

- the training and testing sets follow the same distribution
- the training examples are drawn independently

❒ **Shattering** – Given a set $S = \{x^{(1)},...,x^{(d)}\}$, and a set of classifiers $\mathcal{H}$, we say that $\mathcal{H}$ shatters S if for any set of labels $\{y^{(1)}, ..., y^{(d)}\}$, we have:

$$\boxed{\exists h \in \mathcal{H}, \quad \forall i \in [\![1,d]\!], \quad h(x^{(i)}) = y^{(i)}}$$

❒ **Upper bound theorem** – Let $\mathcal{H}$ be a finite hypothesis class such that $|\mathcal{H}| = k$ and let δ and the sample size m be fixed. Then, with probability of at least $1 - \delta$, we have:

$$\boxed{\epsilon(\widehat{h}) \leqslant \left(\min_{h \in \mathcal{H}} \epsilon(h)\right) + 2\sqrt{\frac{1}{2m}\log\left(\frac{2k}{\delta}\right)}}$$

❒ **VC dimension** – The Vapnik-Chervonenkis (VC) dimension of a given infinite hypothesis class $\mathcal{H}$, noted $\text{VC}(\mathcal{H})$ is the size of the largest set that is shattered by $\mathcal{H}$.

Remark: the VC dimension of $\mathcal{H} = \{$set of linear classifiers in 2 dimensions$\}$ is 3.

❒ **Theorem (Vapnik)** – Let $\mathcal{H}$ be given, with $\text{VC}(\mathcal{H}) = d$ and m the number of training examples. With probability at least $1 - \delta$, we have:

$$\boxed{\epsilon(\widehat{h}) \leqslant \left(\min_{h \in \mathcal{H}} \epsilon(h)\right) + O\left(\sqrt{\frac{d}{m}\log\left(\frac{m}{d}\right) + \frac{1}{m}\log\left(\frac{1}{\delta}\right)}\right)}$$