The Little Book of Data Science Tricks

100+ Techniques to Elevate Your Data Science Skills

…and maybe win a Kaggle competition

Maverick Lin

Table of Contents

Introduction ***7***

Section I: Big Ideas in Data Science ***8***

1. Statistical Modeling 8
2. Types of Models 9
3. Occam's Razor 9
4. Curse of Dimensionality 9
5. Interpretability 10
6. No Free Lunch Theorem 10
7. Bias-Variance Tradeoff 10
8. Parallel Processing or Distributed Computing 11
9. Vectorization 11
10. Overfitting 11
11. Regularization 11
12. Observations as Points in Space 12
13. Big Data Hubris 12
14. Local Minimas 12
15. Gradient Descent 13
16. MLE vs. MAP 13
17. The Cloud and Cloud Computing 14

Section II: General ***15***

Overall **15**

18. Adopt the Right Mindset 15
19. Have an Approach 15
20. Diversity 15
21. Use the Right Tools & Frameworks 16
22. Establish a Baseline 16
23. Set Up a Recording System for Your Experiments 16
24. Obtain Domain Knowledge 17

Model Ensembling **17**

25. Voting Ensemble 17
26. Weighted Voting Ensembles 17
27. Averaging Results 18
28. Rank Averaging 18
29. Stacked Generalization (Stacking) 18
30. Blending 20
31. Automatic Stacking 20

Post-Processing Predictions **20**

32. Logloss Clipping 21
33. Mean Prediction 21
34. Raising Probabilities by *n* 21
35. Convert Predictions to Odds 21

Feature Understanding **21**

36. Correlation Matrix 21
37. Kolmogorov-Smirnov Statistic 22
38. Visualizations 22
40. Data Leakage 22

Section III: Feature Engineering ***24***

General **24**
41. Dealing with NANs 24
42. Aggregations / Group Statistics 24
43. Feature Hashing (Hashing Trick) 25
44. Memory Reduction 25
45. Splitting Features 25
46. Pseudo-Labeling 26
47. Row Statistics 26
48. Derivative or Integral 26

Numerical Data **26**
49. Interaction Features 26
50. Normalizing / Standardizing 26
51. Log Transform 26
52. Box-Cox Transformation 26
53. Outlier Detection 27
54. Discretization/Binning 27
55. Concept Hierarchy Generation 28
56. Matrix Factorization 28

Smoothing **29**
57. Bin Smoothing (Local) 29
58. Simple Moving Average (Local) 30
59. Running Line (Local or Global) 30
60. LOWESS / LOESS (Locally Weighted Scatter Plot Smooth) 30
61. Savitzky-Golay Filtering 30
62. Miscellaneous 31

Time Series Data **31**
63. Date-Time Features 31
64. Lag Features 31
65. Window Statistics 31

Categorical Data **32**
66. Frequency Encoding 32
67. Integer / Label Encoding 32
68. One-Hot Encoding 32
69. Categorizing Locations 32

Text Data **32**
70. Bag-of-Words 32
71. Bag of n-Grams 32
72. Character n-grams 33
73. Parts-of-Speech Tagging 33
74. Term-Frequency-Inverse Document Frequency (td-idf) 33
75. Frequency-Based Filtering 33
76. Stemming 33
77. Miscellaneous 33

Image Data **34**
78. Data Augmentation 34
79. Embeddings 34

Geographical Data (Latitude and Longitude Features) **34**
80. Manhattan Distance 34
81. Haversine Distance 34
82. Create Clusters 35

83. Distance to Locations or Points of Interest ... 35
84. Travel Speed ... 35

Feature Selection ... 35
85. Forward Feature Selection (using single or groups of features) ... 35
86. Recursive Feature Elimination (using single or groups of features) ... 35
89. Permutation Importance ... 35
89. Correlation Analysis ... 36
89. Time Consistency ... 36
90. Feature Trimming ... 36
91. Unsupervised Feature Selection ... 36
92. Boruta ... 36
93. PCA ... 37

Handling Imbalanced Datasets ... 37
94. Appropriate Performance Metric ... 37
95. Appropriate Algorithm ... 38
96. Class/Sample Weights or Cost-Sensitive Learning ... 38
97. Oversample the Minority Class ... 38
98. Undersample the Majority Class ... 38
99. Generate Synthetic Samples (SMOTE) ... 38

Section IV: Validation Strategy ... 39
100. Classic Train-Test Split ... 39
101. Train-Validation-Test Split ... 39
102. *k*-fold Cross-Validation ... 39
103. Stratified *k*-fold ... 40
104. Time Series Validation ... 40
105. Adversarial Validation ... 40

Hyperparameter Tuning ... 41
106. Exhaustive Grid Search ... 41
107. Random Search ... 41
108. Bayesian Optimization ... 41

Final Note ... 41

Foreword

Kaggle has become *the* platform and place for data science and machine learning. Acquired by Google in 2017, Kaggle not only hosts data science and machine learning competitions, but also is an active hub for sharing ideas, datasets, and kernels (Kaggle's analysis, coding, and collaboration product).

One of the biggest challenges I faced competing in data science competitions was that I simply did not know what to do; I felt like a mouse scurrying around in a maze, constantly bumping into dead ends. How do I explore the data? How do I go about engineering features? What models do I use?

It's simple enough to pull a model from scikit-learn, run model.fit() on the data, and submit the predictions; but what are the steps that will not only help me improve my leaderboard scores, but more importantly, improve my general problem-solving abilities?

So, I decided to do a little "training" of my own to understand what differentiates the top data scientists and Kaggle Grandmasters from the rest of us - to construct my training set, I decided to go through all of the winning solutions and as many data science blog posts as possible. The idea being that as I consumed this information, I would be able to identify the techniques and approaches behind the top data scientists. One common reaction I had while going through the solutions was, "Wow that's ingenious- how on earth did they think of that?"

Looking back on my "training", I realized that I had gained a few important insights and a few neat tricks to add to my data science toolbox and which I now present to you. I hope this book serves two purposes:

1) Help you become aware that these approaches and techniques exist and
2) Hopefully inspire you to come up with creative solutions to *your* problems

The most profound realization that I had was that, at the end of the day, data scientists are just problem-solvers. They are a multi-disciplinary breed that utilizes tools multiple domains (statistics, optimization, machine learning, and computer science) to solve problems- whether they be predicting the future, optimizing a system, or recommending better products.

Happy training.

Introduction

The book is organized into the following sections:

- Big Ideas in Data Science
- General
- Feature Engineering
- Validation Strategy

The Big Ideas section consists of 17 ideas in data science that any data scientist should; topics like overfitting, gradient descent, vectorization, curse of dimensionality, bias-variance tradeoff, etc…

The General section is comprised of general tips and observations.

The Feature Engineering section consists of various techniques to engineer and manipulate features.

The Validation Strategy section consists of various validation strategies to test the effectiveness of your models.

Section I: Big Ideas in Data Science

To a man with a hammer everything looks like a nail. -Charlie Munger

Every discipline, whether it be mathematics, physics, or biology, all have several "big ideas" or core principles. This idea is championed by Charlie Munger (Warren Buffet's business partner) who argues that in order to make better decisions in life, it is crucial to understand the big ideas from all disciplines and use them across disciplinary boundaries to solve problems in other disciplines.

Since data science is very much a multi-disciplinary field attempting to solve problems across disciplines, I thought it would be a good idea to compile a list of "big ideas" that data science currently has.

1. Statistical Modeling

Modeling is the process of incorporating information into a tool that 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 $X = (X_1, X_2, \ldots X_d)$ represents the input variables, Y represents the output variable, and ϵ represents random error. Every supervised learning problem can be viewed as finding and approximating a function between inputs X and outputs Y.

Statistical learning is the 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 $\hat{f}(X)$ as a black box, since we only care about the accuracy of the predictions (not necessarily why or how it works).

Inference: we want to understand the relationship between X and Y. We can no longer treat $\hat{f}(X)$ as a black box since we want to understand how Y changes with respect to $X = (X_1, X_2, \ldots X_d)$ (interpretability).

More About ϵ

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

below). Typically, as flexibility increases, interpretability decreases. Much of statistical learning is finding a way to balance the two.

2. Types of Models

There are many different classes of models. It is important to understand the trade-offs between them and know when it is appropriate to use a certain type of model.

Parametric vs. Nonparametric

- **Parametric**: models that first make an assumption about the shape of $f(X)$ (e.g. we assume the data to be linear) and then we fit the model. This simplifies the problem from estimating $f(X)$ to just estimating a set of parameters. However, if our initial assumption was wrong, this will lead to bad results (e.g. assume data is linear but in reality, it's not).
- **Non-Parametric**: models that don't make any assumptions about the shape of $f(X)$, which allows them to fit a wider range of shapes but may lead to overfitting (e.g. k-NN).

Supervised vs. Unsupervised

Supervised: models that fit input variables $X = (X_1, X_2, \ldots X_n)$ to a known output variables $Y = (y_1, y_2, \ldots y_n)$. Most problems and interview questions will be based on supervised learning.
Unsupervised: models that take in input variables $X = (X_1, X_2, \ldots X_n)$ but they do not have an associated output Y to "supervise" the training. The goal is to understand relationships between the variables or observations.

Blackbox vs. Interpretable

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

Generative vs. Discriminative

- **Generative**: learns the joint probability distribution $p(x, y)$. For example, if we wanted to distinguish between fraud or not-fraud, we first build a model for what a fraudulent transaction looks like and another one for what a non-fraudulent transaction looks like. Then, we compare a new transaction to our two models to see which is more similar.
- **Discriminative**: learns the conditional probability distribution $p(y \mid x)$. For example, building upon our fraud analogy, we just try to find a line that separates the two classes (fraud or not-fraud) and don't care about how the data was generated.

3. Occam's Razor

Philosophical principle that the simplest explanation is the best explanation. In modeling, if we are given two models that predict equally well, we should choose the simpler one; choosing the more complex one can often result in overfitting (or just memorizing the training data). Simpler is usually defined as having less parameters or assumptions.

4. Curse of Dimensionality

As the number of features d grows, points become very far apart in Euclidean distance and the entire feature space is needed to find the k nearest neighbors. Eventually, all points become

equidistant, which means all points are equally similar, which then means algorithms that use distance measures are pretty much useless. This is not a problem for some high dimensional datasets since the data lies on a low dimensional subspace (such as images of faces or handwritten digits). In other words, the data only sits in a small corner of the feature space (think how trees only grow near the surface of the Earth and not the entire atmosphere).

5. Interpretability
Briefly mentioned earlier, but we'll cover it more in-depth here. Interpretability (in data science) is the extent to which a human can understand the reasoning behind a decision or prediction from a model. If a model predicts a house is worth $300,000 or that a certain e-mail is junk, why did the model do that? A lot of powerful models may have better performance (e.g. being able to predict housing price or spam better), but certain companies or entire industries may not be able to use such models if the results cannot be clearly explained.

An example would be a twist on the trolley problem involving a self-driving car. If a self-driving car had to choose between killing 5 pedestrians or driving itself off the road and killing the driver, what would it choose? And if it did have to choose, what was the reasoning behind the decision? Another important consideration in this problem would be, was the decision manipulated in some way? An interpretable model can be easily debugged to detect any manipulation while a black-box would be much harder to.

Here are a few interpretable models: linear regression, logistic regression, generalized linear models (GLMs), generalized additive models (GAMs), decision trees, decision rules, RuleFit, naive bayes, and *k*-Nearest Neighbors.

There's also a whole class of models called Model-Agnostic Methods, which we won't cover here. But if you want to learn more, head over to Christoph Molnar's book *Interpretable Machine Learning. A Guide for Making Black Box Models Explainable.*[i]

6. No Free Lunch Theorem
The No Free Lunch (NFL) Theorem states that every successful machine learning algorithm *must make assumptions*. The implication of this is that no single algorithm will work for every problem and that no single algorithm will be the best for all problems. In other words, there is no "Master Algorithm" that will be the best algorithm for every single problem. The solution is to test multiple models in order to find the one that works best for a particular problem.

7. Bias-Variance Tradeoff
Bias is the error resulting from incorrect assumptions in the learning algorithm and the model is too simple (e.g. using linear models when the data is non-linear; high bias → missing relevant relations between inputs and outputs *aka* underfitting).

Variance is the error from sensitivity to fluctuations in the training data, or how much the target estimate would differ if different training data was used; this causes the model to capture random noise rather than the signal due to the model being too complex *aka* overfitting (high variance → modeling noise or overfitting).

The trade-off is the conflict in attempting to minimize *both* bias and variance, since models with lower bias will usually have higher variance and vice versa.

8. Parallel Processing or Distributed Computing

Parallel processing is the process of breaking down a complex problem into simpler tasks that can be simultaneously run independently on different machines. The results are then combined together at the end. Done right, parallelization can dramatically reduce processing time. The most basic example is addition: suppose you want to calculate the following equation: 1 + 1 + 1 + 1. We also assume that each + takes 1 second. If we compute sequentially, it will take 3 seconds. If done in parallel, it will take 2 seconds. It might not seem like much of a difference, but when your operations require millions or billions of calculations, you can really save a lot of time.

9. Vectorization

Vectorization is the process of performing operations on a list or vector instead of on scalar values. Suppose you have a list of numbers [1, 2, 3, 4] and you want to add one to each value. You could use a for loop and iterate over the list and add one to every value (which gets slow if you have to loop through millions of observations). The other way to view it as a matrix operation and can be done in one step (as opposed to n steps). Certain packages have been optimized for vectorization (NumPy) and is the recommended way to perform calculations since the code is more readable and is generally faster than looping. One example of vectorization is updating weights in a neural network during backpropagation (instead of updating each weight individually, we can update all the weights using a matrix operation).

Note: in Python, for loops are slower than for loops in C. NumPy offers vectorized operations on NumPy arrays (which pushes the for loop down to the C level, making the operation much faster than the for loop in Python).

10. Overfitting

Overfitting is a modeling error when the model we trained basically memorizes the data it has seen, which captures noise in the data and not the true underlying function. One analogy to think about it is you have a test tomorrow and your teacher gave you last year's test. Instead of studying the material on the test, you only studied last year's test and memorized all the answers. On the day of the test, you find out that the test is completely different.

Memorizing last year's test is essentially overfitting. You don't actually learn anything about the underlying topics on the test, so when presented with new questions about the topic, you cannot answer them well (also known as failing to generalize). A model that only performs well on the training data is pretty much useless when presented with new information. Some ways to prevent overfitting is cross-validation and regularization.

11. Regularization

A main problem in machine learning is overfitting or when the model does not perform well on new data because it has essentially memorized the training data. This can happen when the model is too complex. Regularization prevents the model from becoming too complex by adding

a tuning parameter that shrinks the coefficient estimates. Two popular types of regularizations are:

- **Lasso (L1):** adds the absolute value of the magnitude of coefficients as the penalty term to the loss function: $min\ (L(y, y') + \lambda \sum_{j=1}^{p} |\beta_j|)$
- **Ridge (L2):** adds the square of the magnitude of coefficients as the penalty term to the loss function: $min\ (L(y, y') + \lambda \sum_{j=1}^{p} \beta_j^2)$

λ represents the tuning parameter; as λ increases, flexibility decreases, which leads to decreased variance but increased bias (again, bias-variance tradeoff). λ is key in determining the sweet spot between under and overfitting. In addition, while Ridge will always produce a model with all the variables, Lasso can force coefficients to be equal to zero (feature selection).

12. Observations as Points in Space

Many machine learning models take in vectors of numbers as inputs e.g. [42, 21, 42]. Each vector can be represented visually as points in space. Imagine we have a dataset that has two features: "Height" and "Weight". Our dataset might look something like this:

Height	Weight
5.5	150
5.2	120
4.1	100

We can then visualize these points on a 2-dimensional graph as *(x, y)* pairs. Building off this logic, we can do the same for 3-dimensions: *(x, y, z)* coordinates. And so on and so forth. However, anything over 3-dimensions we cannot visualize but we can still think of them as points in *n*-dimensional space.

13. Big Data Hubris

"Big data hubris" is the assumption that big data is a substitute for, rather than a supplement to, traditional data collection and analysis. In other words, it's a belief (and overconfidence) that huge amounts of data is the answer to everything and that we can just train machines to solve problems automatically. Data by itself is not a panacea and we cannot ignore traditional analysis.

An infamous example is Google Flu Trends (GFT), which was a service that attempted to predict real-time flu outbreaks by analyzing flu-related Google Search keywords collected in the US. However, GFT performed terribly; between 2011 and 2013, GFT was wrong 100 out of 108 weeks and the service was discontinued in 2013. It was also discovered that GFT's accuracy was not much better than a simple projection using available CDC data; another example of Occam's Razor.

14. Local Minimas

Suppose we have a function *f(X)* and we want to find the point that minimizes that function. Local minimas are the best solutions or points in a corresponding neighborhood; in other words, these are pretty good solutions but they are not the best. The global solution is the best solution overall.

The concept of local minimas is related to the concept of convex and non-convex functions; convex functions are guaranteed to have a global minimum or solution, while non-convex functions do not have that guarantee, which results in models obtaining local minimas. The best we can do is run an optimization algorithm (e.g. gradient descent) from different starting points and use the best local minima we find for the solution.

15. Gradient Descent

Gradient descent is an optimization algorithm to find solutions or minimas that minimize the value of a given function. Suppose you are wandering around some mountains and you want to find the lowest point that exists. You first look around you and figure out which way is downhill and you take a small step. You then do that again and again until you reach a point where every direction is uphill. You have now found a minima, but it might not be the lowest point in the mountains.

Now, you are guaranteed to reach the bottom if there was only one valley (convex function), but you might get stuck in a valley that is sub-optimal if there were multiple valleys scattered throughout the mountains (non-convex function).

Gradient descent is analogous to you looking for the downhill direction; it starts at an arbitrary point and moves in a downward direction (or the negative direction of the gradient). After several iterations, it will eventually converge to the minimum. The learning rate α is simply how large a step we take; too large of a step-size will result in us bouncing around and never reach the bottom.

Stochastic gradient descent (SGD) is a variant of gradient descent and simply performs the gradient step on one or a few data points (as opposed to all the data points), which allows us to jump around and avoid getting stuck in a local minima. It is computationally more efficient and may lead to faster convergence, but results in noisier results.

Mini-batch stochastic gradient descent (mini-batch SGD) randomly chooses batches of data points (between 10 and 10,000) and then performs a gradient step. This helps reduce noise and also helps speed up training.

16. MLE vs. MAP

Maximum Likelihood Estimation (MLE) is a frequentist method to estimate the parameters of a probability distribution that best fits the observations. So the two main steps are:

1. Make an assumption about the distribution of the observations
2. Find the parameters of the distribution so that the observations you have are as likely as possible. This is done by maximizing the likelihood function:
    a. Write down the likelihood function $L(\theta) = \prod_{i=1}^{n} f_X(x_i;\ \theta)$
    b. Take the natural log of $L(\theta)$: $\log(L(\theta))$ (turns products into sums → easier to deal with)
    c. Take the derivative of the $\log(L(\theta))$ and set it equal to 0; solve for θ
    d. Check that the estimate found in the last step is truly the maximum by inspecting the second derivative of $\log(L(\theta))$ with respect to θ (maximum if second derivative is negative)

Maximum a Posteriori (MAP) is a Bayesian method to estimate the parameters of a probability distribution by maximizing the posterior distribution. This assumes a prior distribution and updates the prior distribution using Bayes' Rule to arrive at θ.

The connection between MLE and MAP is that MAP is just MLE if we use a uniform prior (or we just assume every value of θ is possible). MLE works well when the assumed distribution you are estimating parameters for is correct and you have a large number of observations n, but can be very wrong when your assumed distribution is wrong and n is small (can overfit). MAP works well when your prior is accurate, but can be very wrong if n is small and your prior is wrong.

17. The Cloud and Cloud Computing

Cloud computing is the delivery of on-demand computing services over the internet with pay-as-you-go pricing. Computing services include servers, storage, computing power, databases, software, analytics and machine learning, and quantum technologies. Utilizing the cloud is cost efficient, flexible, and scalable, allowing you to only pay for what you use and focus on developing and growing your product or organization without having to worry about IT management maintenance (hardware setup, software patching, etc.). For example, if you suddenly need terabytes of storage or hundreds of compute engines, you are just a couple of minutes away from fulfilling those requirements using the cloud.

Companies of all types are using the cloud for various purposes, from big data analytics to web applications to data storage and backup. For example, Netflix spent 7 years migrating everything to the cloud (AWS), which now runs all of Netflix's computing and storage needs, from storing customer information to recommendation engines.

The top cloud providers are Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform, IBM Cloud, Salesforce, and Alibaba Cloud.

Section II: General

This section is comprised of general tips that don't fit in Feature Engineering or Validation Strategy.

Overall

18. Adopt the Right Mindset

Data science isn't right for everyone. To be an effective data scientist, you should:

1) **Embrace Failure:** be willing to fail at a lot of things and learn from them; for example, you might engineer hundreds of features (or train hundreds of models) but none of them turn out to be useful; understand that 99% of your ideas might not work but that's ok.
2) **Be Curious**: have an insatiable curiosity for the problem and the data at hand
3) **Understand Your Tools:** understand the limitations of the tools you are working with and how to circumvent those limitations
4) **Welcome Diversity**: the winning Kaggle teams (and successful real-life data science teams) work in teams and understand that diversity is essential to better teams and helps increase problem-solving abilities
5) **Be Skeptical**: if the results are too good, then you should be cautious; be wary of overfitting and potential data leakage
6) **Be an Effective Communicator**: at the end of the day, you should be able to effectively communicate your results and findings with not only your teammates but also with various shareholders, business managers, etc.

19. Have an Approach

You should have an approach to solving data science problems[ii]:

1) **Understand the Data**: explore the features, visualize the data, detect and understand any anomalies
2) **Understand the Evaluation Metric**: every problem has its own evaluation metric, so understand how it changes with respect to the target variable
3) **Decide on a Validation Strategy**: a solid validation strategy will help you evaluate your model accurately and avoid overfitting
4) **Hyperparameter Tuning**: can think of everything as a hyperparameter to be tuned
    a. How you clean and transform your data
    b. What features you engineer
    c. What algorithms you choose to train and their parameters
    d. How you ensemble and combine your models

20. Diversity

Diversity plays a huge role in building successful models. In real-life, a diverse group leads to better problem-solving abilities; the blend of different backgrounds, experiences, personalities, and knowledge contributes to more opportunities to solve the given problem from different angles.

On the flip side, a homogenous group of people with the same background and thought processes is severely limited since every member will approach the problem from a limited set of viewpoints.

From a data science perspective, it's often helpful to have a diverse group of people working together to build diverse models. Kaggle competition winners pretty much all work in teams; to ensure diversity, they work independently so they don't limit each other's viewpoints. The results are then aggregated at the end to produce results (ensembling).

Apart from working independently, diversity can be achieved by:

1) Different Models
2) Different Features
3) Different Resampling of Training Data

To sum up, diverse groups lead to diverse preprocessing, which leads to diverse models, which finally leads to stronger ensembles.

21. Use the Right Tools & Frameworks

- Python Packages
    - Vowpal Wabbit: fast machine learning library that
    - XGBoost/LightGBM/CatBoost: libraries for gradient boosting
    - LibFFM: library for field-aware factorization machines
    - Keras: neural network library
    - NLTK: natural language processing toolkit
    - Numpy/Pandas/Scipy: data manipulation
    - Modin: Pandas wrapper that helps speed up computations by distributing the data and computation; all you need to do is install it and change one line of code
    - Data Visualization: matplotlib, seaborn, pyplot
    - Scikit-learn: machine learning models
    - Time Series: ARIMA, ARCH/GARCH
    - PySpark: allows us to handle huge datasets efficiently by processing data in a distributed way using a cluster; supports set of higher-level tools such as Spark SQL for SQL and DataFrames, MLlib for machine learning, GraphX for graph processing, and Spark Streaming for stream processing
    - Statsmodels: provides classes and functions for the estimation of various statistical models, as well for conducting statistical tests and data exploration

22. Establish a Baseline

Creating a simple baseline model/score that all future models are compared against is extremely important. It's no use to have a complex model predict with 75% accuracy when a baseline model already has 80% accuracy.

For example, a baseline model for predicting hair color might be just to predict the most popular color. Another example would be if you are predicting house prices, just predict the mean of the prices.

23. Set Up a Recording System for Your Experiments

It's important to keep note of what you have tried, what failed, and what succeeded. Otherwise, you might end up re-creating features or trying the same thing again, which wastes valuable

time. Plus, if you end up working on a similar problem in the future, you can look back and see what worked or not.

24. Obtain Domain Knowledge

Most problems require some form of domain knowledge. In other words, if the given data is from a particle accelerator, it would be good to have some knowledge or background (probably in-depth knowledge) in physics. Otherwise, you have a hard time deciding on what kind of features to engineer and how to interpret the results.

Model Ensembling[iii]

Ensembling is a powerful technique to increase the accuracy of predictive models. The main idea behind ensembling is for a group of weak learners (tens or hundreds or thousands) to come together to become one strong learner. There are multiple ways of ensembling and we'll go through some of the methods below.

Note: pretty much all Kaggle competitions are won by some form of ensembling.

25. Voting Ensemble

This method ensembles model predictions. If you already have 5 models trained and they have predictions for the test set, you can just combine the predictions using a simple vote. For example, if we have five predictions for test point A, [1, 1, 0, 0, 1], the voted result will be 1.

To see why voting works, we can turn to probability. If we have 3 models and each model has 60% accuracy, there are 4 outcomes for a data point:

1) All 3 Models Are Correct: (0.6)(0.6)(0.6) = 0.216
2) 2 Models are Correct: (0.6)(0.6)(0.4) + (0.6)(0.4)(0.6) + (0.4)(0.6)(0.6) = 0.432
3) 2 Models are Wrong: (0.6)(0.4)(0.4) + (0.4)(0.6)(0.4) + (0.4)(0.4)(0.6) = 0.288
4) All 3 are wrong: (0.4)(0.4)(0.4) = 0.064

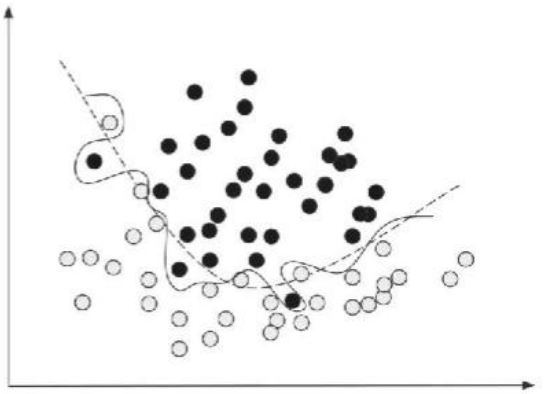
So, the probability that the voting ensemble of 3 models classifies a data point correctly is 0.216 + 0.432 = 0.648, which is higher than if we just used a single model. If you add more models, your ensemble should result in better results.

However, one caveat is that the results should not be correlated since correlation won't increase the models' ability to correct each other's errors. One trick is to calculate the correlation between results and choose the models whose results are less correlated.

26. Weighted Voting Ensembles

Using the plain vanilla ensemble essentially assigns each member in the ensemble an equal say. However, in some cases, we want to use a weighted ensemble by giving better models more weights. A simple way of doing this is giving the best model 2 votes and giving the rest 1 vote. The main idea is that the only way the worse models can overrule the best model is if they come together and vote it down.

27. Averaging Results
Averaging is simply taking the average predictions of many different models. This helps reduce overfitting and creates a smooth separation between classes (see image), since a single model's predictions can create jagged and rough separations. We can also use arithmetic mean or geometric mean.



28. Rank Averaging
Similar to simple averaging, but instead of averaging the predictions, we first turn the predictions into ranks (e.g [0.4, 0.2, 0.8, 0.9] → [2, 1, 3, 4]), average the ranked predictions of multiple models, and finally normalize the averaged ranks between 0 and 1.

Historical Ranks: a trick to predict the score for a new test sample is to store the old test set predictions (along with their ranks) and find the most similar old prediction in the old test set predictions and use its historical rank.

29. Stacked Generalization (Stacking)
This is an ensemble technique that uses a model (or second-stage stacker model) to learn how to combine the predictions from two or more models (or first-stage models).

A key for stacking is making sure that the first-stage models aren't just simple variations of each other that produce uncorrelated predictions. In other words, having a diverse set of first-stage models is extremely important to improving ensembles.

By having a diverse set of first-stage models, they can each "examine" the training set from various perspectives. This can be done by:

1) Using different models
2) Using models with varying parameters
3) Training models using different parts of the training set
4) Training using different features

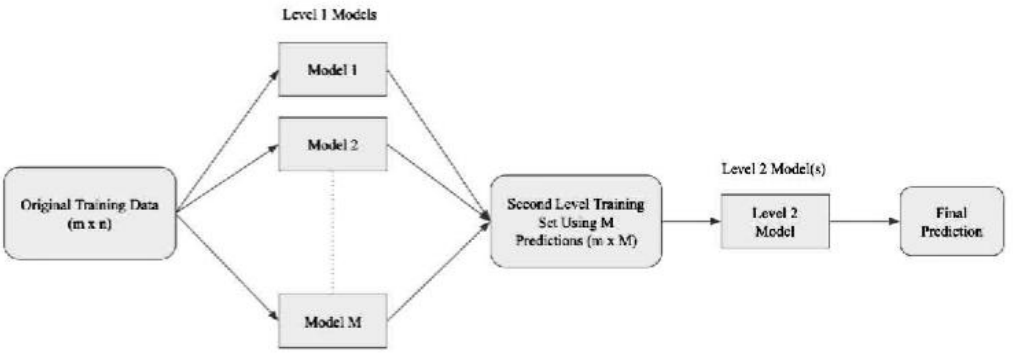
This way, the second-stage model can learn from all of these first-stage models and make better predictions. Think of the second-stage model as a general and the first-stage models as scouts.

As a general, you wouldn't just send one scout to survey the enemy since he/she might miss something. So, you send multiple scouts and learn from the group of scouts.

Here's an example of how you would do *k*-fold stacking:

1) Split the training set into *k* folds
2) Iteratively train a base learner on every *k* - 1 folds and create predictions for the k_{th} fold until we have predictions for each of the folds
    a) This step is key since we want out-of-fold (OOF) predictions- in other words, our model is generating predictions on data it has never seen before. Otherwise we would have trained a model on the entire training set, which biases the second-level model towards the best of the *L* base learners.
3) Use the same base learner and train on the entire training set and create predictions for the test set
4) Repeat step 2 and 3 until all the models have trained and generated OOF predictions.
    a) If we have *n* data points and *L* learners, we can stack the predictions and have an *n* x *L* dimension training set (or meta_training set)
    b) We will also have a stacked meta_test set by combining the predictions of the whole training set
5) Now we can train a second-stage stacker model on this new meta_training set and predict on the meta_test set

Note that you don't have to just use the meta_train dataset- you can combine the meta_train with the original train set to create an augmented train set.



To combine, you can use both regressors and classifiers, as well as linear (linear regression, logistic regression) and non-linear models (KNN, neural networks, random forests, and gradient boosting machines).

You can also stack with unsupervised learned features (*K*-means or *t*-SNE).

One important thing to realize with stacking/blending or meta-modeling in general, is that everything can be viewed as a hyper-parameter for the stacker model. How you engineer features, scale features, feature selection, the number of base models, model parameters- all of these can be seen as hyperparameters to tune. Just how random gridsearch works for tuning model parameters, you can use it to tune these "meta-parameters".

30. Blending
Blending is similar to stacked generalization but is a bit simpler and has a smaller risk of information leak. Instead of creating out-of-fold predictions for the train set, you create a small holdout set of the train set. The stacker model then only trains on this holdout set.

Here's an example of how you would do blending:

1) Split the training set into 2 parts: A and B (disjoint sets)
2) Train several base learners on A and create predictions for B and C (test)
3) Create a new dataset B1 by stacking the predictions for B
4) Create a new dataset C1 by stacking the predictions for C
5) Train a model on B1 and make predictions for C1

Blending is simpler than stacking, wards against information leak (since the base learners and stackers use different data), and your cross-validation is more solid with stacking. The downsides are that you use less data overall and the final model may overfit to the holdout set.

31. Automatic Stacking
Once you truly understand that stacking is just finding various ways to combine hundreds or even thousands of models, along with various ways to combine manipulated features, the entire process can be… well, automated. You can randomly select the base learners and stacker models, randomly choose ways to combine the features and meta-features, and let them train over and over again on the data.

Last Note on Ensembling
So why do we want to build these large, unwieldy models consisting of hundreds or thousands of models? Doesn't it seem like a waste of time and computational resources?

Well, yes. But the slight gain in accuracy might not justify the effort. This actually happened for the winning model for the Netflix Prize: a $1 million prize competition hosted by Netflix to improve the accuracy of its recommendation system. The winning model was essentially a blend of hundreds of predictive models, which took years of work, but Netflix decided not to move the algorithms into production because the additional accuracy did not "justify the engineering effort needed to bring them into a production environment".[v]

However, it is still one of the best methods to improve machine learning models and a 1% increase in performance might mean stopping millions of dollars from being the victim of financial fraud, or stopping a cyberattack that could have leaked countless of sensitive information, or saving millions of lives by improving healthcare screening tools.

So, in the end, whether or not it's worth building these "Frankenstein ensembles" depends entirely on the problem you are trying to solve.

Post-Processing Predictions
Sometimes it's necessary to post-process the predictions, meaning that you provide a manual override or apply some transformation to the predictions.

For example, in Kaggle's IEE-CIS Fraud Detection Challenge[vi], one realization was that if a card was declared fraudulent, then the rest of the transactions on that card are probably fraudulent too. So, if your model predicts that the transactions after the fraudulent transaction to be non-fraud, you might want to override that prediction. Intuitively, that makes sense, since a fraudulent card will continue to result in fraudulent transactions (until it is terminated)- it is unlikely that a fraudulent card makes a legitimate transaction.

32. Logloss Clipping

Logloss penalizes us if our predictions are very confident and wrong. In the case of classification problems where we have to predict probabilities, it might be a good idea to clip our probabilities between 0.05 and 0.95 so that we are never very confident with our predictions.

33. Mean Prediction

Similar to logloss clipping, but instead of clipping the probabilities, we can just take the mean of predictions for certain groups. For example, if we are predicting fraud for certain bank accounts, it might be good to replace the predictions for every unique bank account with the mean predictions.

34. Raising Probabilities by *n*

In certain cases, you might want to raise the prediction probabilities by *n* and then re-normalize. For example, if you have a random forest algorithm, it might do a good job of predicting the class but not the probabilities (maybe the probabilities bunch too close to the center). You can select *n* by performing grid search and select a *n* for each class.[vii]

35. Convert Predictions to Odds

You can also consider converting prediction probabilities *p* into odds[viii] using the following formula: $\boldsymbol{odds = \frac{p}{1-p}}$.

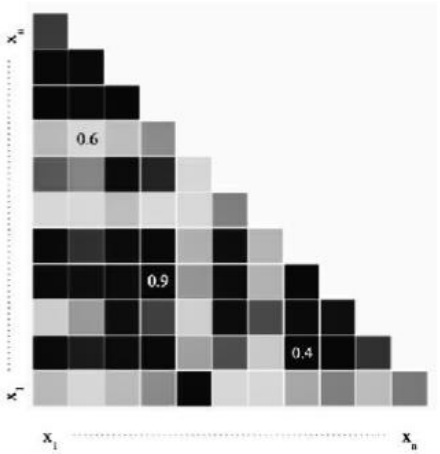
Feature Understanding

Understanding the data you have available is extremely important.

For example, in the Santander Customer Transaction Problem, one major detail was that the test set consisted of synthetic samples that were generated by sampling from the real samples. In another competition, the features were all independent, meaning that the features had already been processed (probably using PCA).

36. Correlation Matrix

Table showing the correlation coefficients between variables. Based on the correlation matrix, you can see which features are highly correlated and decide whether or not to include certain features in your model.



37. Kolmogorov-Smirnov Statistic
Test used to determine if a sample comes from a population with a specific distribution (one-sample K-S test) or to compare two samples (two-sample K-S test).

38. Visualizations
Visualizing the data is a great way to understand the data:

- **Histograms**: helps check the distribution of features
- **Scatter Plots**: helps with identifying dependencies between two variables
- **Line Graphs**: helps with plotting time-series and detecting outliers
- **Box-Plots**: displays measures of central tendency as well as providing information about symmetry and outliers
- **QQ-Plots**: helps detect non-normality

39. Train/Test Distribution Consistency
One of the most important parts in model building is the data. Specifically, the train and test data (and the future unseen data) should be from the same distribution- otherwise, the model that was trained on the training data will be ineffective on the test data. So, it is important to ensure that each feature is consistent across both train and test.

For example, an example of a discrepancy is that if feature X contains high values in the train data but low values in the test data.

40. Data Leakage
This phenomenon occurs when the training data contains the information that you are trying to predict. As a result, your model will just memorize the training data and overfit.

For example, accidentally having the output variable y in the training set will just lead the model to predict 1 whenever $y = 1$ and 0 whenever $y = 0$.

A few ways data leakage can occur are:

a. Leaking data from the test set into the training set

b. Leaking the correct prediction into the test set
c. Leaking information from the future into the past (for time series data)
d. Leaking external information into the training set that the algorithm normally doesn't have access to

Combatting or preventing data leakage can be done by exploratory data analysis (EDA) and being cautious when your model's performance is too good to be true. If you are building models to be put into production, then you could also put the model into production to see how well it performs (although this is definitely the most expensive step).[ix]

Section III: Feature Engineering

Since machine learning algorithms use input features to create outputs, the goal of feature engineering is to transform the given features into new features that help the algorithm understand the relationship between the inputs and the output. Done properly, it can greatly improve the predictive power of your models.

As Andrew Ng puts it:

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

Why do we need feature engineering? Because in a certain form, the given features might not be useful for our model. For example, let's say we are given a string of your browsing history as a feature: ["May 3, 2020 11:59:59 http://www.story-train.com Safari 72.247.244.88"] [(VISIT TIME, URL, WEB BROWSER, IP ADDRESS)]. If you feed that into an algorithm, it won't necessarily understand the significance of the features contained within the string- the data will just see it as a string.

But what we can do is extract features from the string: time features (hour, month, second, day of week, hours after original air time, etc.), location features (city/country of IP Address, coordinates of location, etc.), and even meta-features (how long the string is, how many of each letter or punctuation it contains, etc.). Keep in mind that the features you engineer might be unimportant or redundant, so you will have to discard those that are not through a constant process of trial-and-error.

General

Below are some general tips while coming up with features and data cleaning.[x]

41. Dealing with NANs

NaNs are an interesting problem to deal with. First step would be to understand why the NaN values are there- is the data corrupt? Pipeline problem? Faulty equipment? Once you understand why NaNs are created, you can start making assumptions on the expected behavior of the NaNs for each feature.

You can drop the rows containing NaNs (which might drop too many samples), impute the Nan values (using average, median, linear regression), or just convert all NaNs into a negative number lower than all non-NaN values. Before you do that, you can even create a new feature based on the number of NaNs in each column.

42. Aggregations / Group Statistics

You can also create group statistics, such as mean, max, or min, for a particular group of data. This will tell the algorithm if a value is usual or unusual for a particular group.

Suppose we are trying to identify ATM fraud. We might want create a variable 'avg_withdrawal_avg' that specifies the average amount that is withdrawn from the ATM so the

algorithm might learn that if a transaction amount is greater than the average, there might be a higher chance of fraud.

Transaction ID	User ID	X_1	Group X_1
A	1	1	0.50
B	1	0	0.50
C	2	1	0.66
D	2	1	0.66
E	2	0	0.66
F	3	1	1.00

One idea to generate aggregations is to create a correlation matrix of all the features, pair each feature that it's least correlated with, and apply various aggregations on each pair.[xi]

43. Feature Hashing (Hashing Trick)
You can transform features into indices in a *n*-dimensional vector by applying a hash function to the values. The number of dimensions used for the vector is usually very large, around 2^{25}.

We first set the dimension *n* of the vector. Then we choose a hash function *h* that will take each value in input *X* and return *h*(value) so it returns a number between 0 and *n*-1. We then increment the value at the given index by one.

So, for example, if the input string is, "Hello World! Good Morning! What a Good Day!", we might end up with a sparse vector like this: [0 … 1 1 0 1 0 1 0 0 2…. 0 0 1] (2^{25} elements).

44. Memory Reduction
If you are working with large datasets, it might be useful to reduce the size of the dataset without losing information. You can label encode your categorical columns into integers and also reduce your numerical columns (e.g. by converting float64 → float32 and int64 → int32).

45. Splitting Features
A feature column can be made into two (or more) columns by splitting the value in the column. Suppose we are given the following string from a stock exchange of the format:
["ACCOUNT_NUM DATE TIME TICKER SHARE_PRICE"].

Suppose the data is ["0123456789 2020-06-01 12:54:21 AAPL 100 $320.21"]. We can split the information (which is one string) into multiple chunks: [0123456789, 2020-06-01, 12:54:21, "AAPL", 100, "$320.21"]. We can further split this data into maybe the first two digits of the account number, or even hour / minute / second, or perhaps dollars and cents.

46. Pseudo-Labeling
This is a semi-supervised approach[xii]. The idea behind pseudo-labeling is to use a small set of labeled data along with a large amount of unlabeled data to improve a model's performance.

Here are the following steps:

1) Train your model on the labeled data
2) Use the trained model to predict labels on a batch of unlabeled data (or pseudo-labels); try to aim for ¼ to ⅓ of the batch be pseudo-labels
3) Combine the real training labels / features with the pseudo-labels / features
4) Train the model again using this "augmented" dataset

47. Row Statistics
We can even use meta-features of the row to create features, such as the number of NaNs, negatives, or positives in a row. Can also use the row's mean, max, min, skewness, etc.

48. Derivative or Integral
For time series data, you can calculate the derivative or integral.

For example, if you have acceleration as a feature, you can take the derivative of acceleration to calculate jerk. Of if you have velocity as feature, you can take the integral to get the displacement.

Numerical Data

49. Interaction Features
Suppose you have features A and B. They can be combined together to create features A*B, A+B, A/B, A-B. Individually, the features may not correlate with the output variable but an interaction of the two might.

50. Normalizing / Standardizing
We can also transform data so that they lie on the same scale.

- Min-max Scaling: scales the data between [0, 1]; $x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$
- z-score Normalization: scales data so mean is 0 and variance is 1; $x_{new} = \frac{x - \bar{x}}{\sqrt{var(x)}}$
- Mean Normalization: scales data between [−1, +1] with mean 0; $x_{new} = \frac{x - \bar{x}}{x_{max} - x_{min}}$

51. Log Transform
One of the most common used transformations to engineer features. Why? Because it helps transform skewed data into less skewed data (more normal). Many machine learning models assume that the data is assumed to be a normal distribution. You can even perform log transform on the target variable and predict on the log instead of the raw predictions.

You can directly take the log of the value ($\log(y)$) or take the log of the value + n ($\log(y + n)$).

52. Box-Cox Transformation
A way to transform non-normal data into a more normal distribution. The transformation formula is: $x_{new} = \frac{x^{\lambda} - 1}{\lambda}$ if $\lambda \neq 0$ and $\log(\lambda)$ if $\lambda = 0$.

Note that this formula only works for positive data. Lambda values typically vary between -5 and 5- several values of lambda are tested and the best lambda is selected, depending on how well the results approximate a normal distribution.

53. Outlier Detection

Outliers in data are observations that lie an abnormal distance from other values. For example, if you have a feature column representing human heights and one of the data points was 240in (or 20ft), that data point is an outlier. Outliers may be the result of faulty data collection or entry, errors in the pipeline, or could just be variability in the data.

It's up to you to determine if the outlier makes sense and how to deal with it. You can detect outliers by examining the data visually by plotting the data, which is probably one of the best ways to go since the other detection methods are prone to making errors.

Other ways include using:

- Standard deviation: if value is greater than x * standard deviation, where x is usually between 2 and 4)
- Percentiles: data within the top or bottom x%
- Capping: any value over x just becomes x
- Frequency Removal: replace values that appear less than 0.5% of the time with another value like -9999
- Isolation Forest: the algorithm works by creating a bunch of decision trees and calculating the path length necessary to isolate an observation in the tree; the scores are then averaged; the main idea is that isolated observations, or anomalies, are easier to isolate because there are fewer conditions necessary to distinguish them from the normal cases

One important thing to note about finding outliers is that in most problems you want to remove them because they throw off your models. However, in certain problems (like fraud detection), you don't want to remove them since the problem is essentially detecting anomalies- the number of transactions that are fraud are extremely low relative to the number of total transactions. So, if you apply outlier detection methods, you might end up accidentally removing the fraud instances.

54. Discretization/Binning

Data can be thrown into various bins. In other words, data between the values:

- 0-10 → Best
- 10-20 → Good
- 20-30 → Ok
- etc…

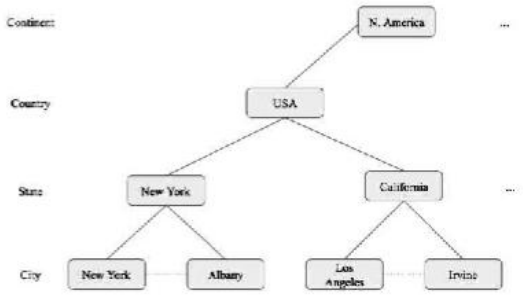
Discretization helps regularize the data and helps prevent overfitting.

55. Concept Hierarchy Generation

Nominal data can be generalized into a hierarchy. For example, street features can be generalized to higher-level concepts like city or country[xiii], or certain animals can be generalized to mammals, reptiles, etc.

For example:

- China / Thailand → Asia
- France / Germany→ Europe
- Canada / United States → North America
- etc.



56. Matrix Factorization

Matrix factorization (or decomposition) is the process of decomposing a matrix into two or more matrices such that when you multiply the decomposed matrices, you get back the original matrix.

The goal is to discover latent (or hidden) features between two entities. For example, a popular application is to build recommendation systems (Netflix or YouTube) and these systems use matrix factorization to identify hidden features between how a user rates an item.

Item Matrix

Movie 1	Movie 2	Movie 3	Movie 4
0.3	1.2	1.4	0.9
1.7	0.8	1.8	2.7

User Matrix

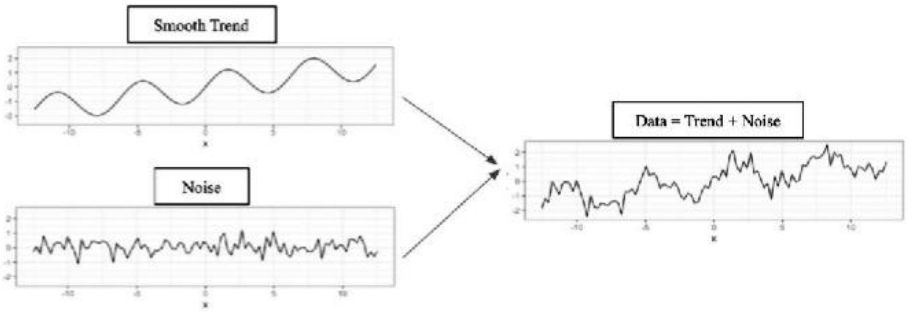
Bob	1.2	0.6
Mary	0.4	1.1
Joe	0.3	0.8
Emma	0.9	0.2

	Movie 1	Movie 2	Movie 3	Movie 4
Bob	4.0	2.0		2.0
Mary	1.0		3.0	
Joe	5.0		1.0	2.0
Emma		4.0	3.0	

Rating Matrix

Smoothing

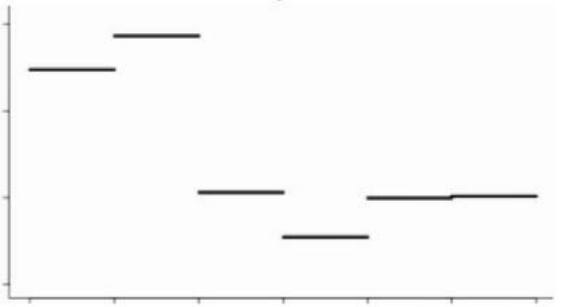
Sometimes the data is noisy and it's a good idea to detect the underlying trends in the data. Thus, we assume the trend is smooth and the noise is not. This is also known as curve fitting and low pass filtering.



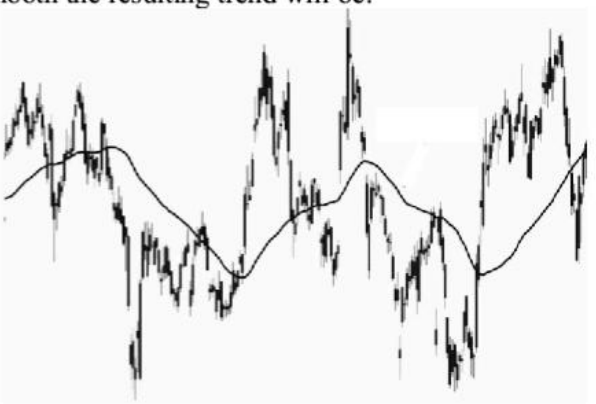
Smoothing techniques fall into two main categories: global and local. Global means identifying a trend over the entire series. Local refers to identifying trends on a smaller subset of the series and if done repeatedly over the entire series, we'll get an estimation of the underlying trend.

57. Bin Smoothing (Local)

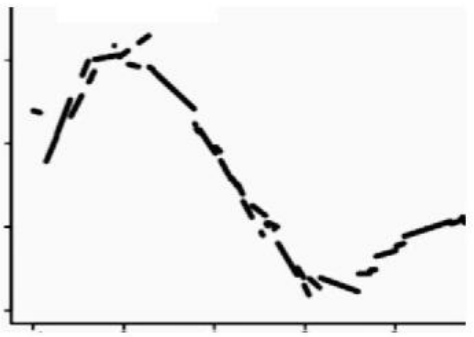
In this method, we smooth a data point by looking at the data points around it. We split the sorted data into discrete and disjoint bins and calculate the average. We can also use median or boundaries by taking the minimum and maximum value in a bin as bin boundaries and replace each bin value by the closest boundary value. The result will look like a step-function.



58. Simple Moving Average (Local)
Similar to bin smoothing, except that we take the mean of *n* nearby points. However, the choice of *n* determines how smooth the resulting trend will be.



59. Running Line (Local or Global)
Instead of taking the mean like moving average, we fit a linear regression through the nearby points. We can also just fit a linear regression (or polynomial or quadratic) through the entire series to obtain a global trend.



60. LOWESS / LOESS (Locally Weighted Scatter Plot Smooth)
Builds upon running line by using a weighted linear regression. It finds the *k* nearest neighbors (like *k*-NN) and fits linear regression using the selected data points; data points farther away are given less weight and data points closer are given more weight.

LOWESS uses a linear polynomial while LOESS uses a quadratic polynomial.

61. Savitzky-Golay Filtering
Can be thought of generalized moving average. Helps smooth the data by getting rid of small noisy fluctuations while retaining the signal's overall shape (area, position, and width of peaks). Each value of the series is replaced with a new value which is obtained by fitting a polynomial to 2n + 1 neighboring points (including the point to be smoothed).