PhD notes V: Statistics

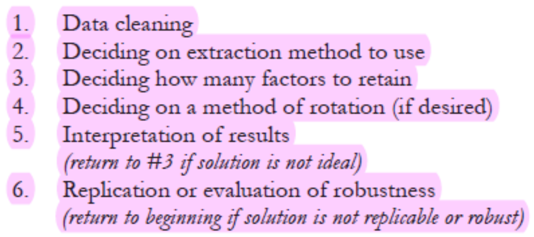
Contents

PhD notes V: Statistics

1. Exploratory factor analysis (EFA)

EFA examines all the pairwise relationships between individual variables and seeks to extract latent factors from the measured variables. **Multiple observed variables have similar patterns of responses because they are all associated with a latent variable (i.e. not directly measured).** It is a group of **extraction and rotation** techniques that are all designed to model unobserved or latent constructs. The goal is to explore whether your data fits a model that makes sense [1].

- Assumes and asserts that there are latent variables that give rise to the manifest (observed) variables, and the calculations and results are interpreted very differently in light of this assumption. Factor analysis also recognizes that model variance contains both shared and unique variance across variables.
- Examines only the **shared variance from the model each time a factor is created**, while allowing the unique variance and error variance to remain in the model. When the factors are uncorrelated and communalities are moderate, **PCA can produce inflated values of variance accounted for by the components** (Gorsuch, 1997; McArdle, 1990). Since factor analysis only analyzes shared variance, factor analysis should yield the same general solution (all other things being equal) while also avoiding the illegitimate inflation of variance estimates.
- Conversely to PCA, it has the option of acknowledging less than perfect reliability
- Each **factor captures a certain amount of the overall variance in the observed variables**, and the factors are always listed in order of how much variation they explain. **The eigenvalue is a measure of how much of the common variance of the observed variables a factor explains.** Any factor with an eigenvalue >1 explains more variance than a single observed variable. The factor loadings express the relationship of each variable to the underlying factor. Since factor loadings can be interpreted like standardized regression coefficients, one could also say that the variable income has a correlation of (factor loading) with Factor X.
- Steps to follow when conducting an EFA:



1. Data cleaning
2. Deciding on extraction method to use
3. Deciding how many factors to retain
4. Deciding on a method of rotation (if desired)
5. Interpretation of results
   (return to #3 if solution is not ideal)
6. Replication or evaluation of robustness
   (return to beginning if solution is not replicable or robust)

1.1. Extraction techniques

An extraction technique is one of a group of methods that examines the correlation/covariation between all the variables and seeks to "extract" the latent variables from the measured/manifest variables. Process of **reducing** the number of dimensions being analysed form the number of variables in the dataset into a smaller number of factors. Extraction of factors proceeds by first extracting the strongest factor that accounts for the most variance, and then progressively extracting successive factors that account for the most remaining variance. **ML is the preferred choices for when data exhibit multivariate normality and PAF for when that assumption is violated.**

Correlations	Are most commonly used in EFA as they are only influenced by the magnitude of the association of the two variables
Covariances	Are influenced by association, as well as the variance of each of the two variables in question.

(i) **Principal Axes Factoring extraction (PAF):**
   (a) Initial estimates of communality coefficients, which can be obtained either from a PCA or as a multiple regression equation predicting each variable from all other variables (multiple R^2) to provide starting values. Communality coefficients can be considered lower-bound estimates of score reliability to provide starting values (initial extraction), and also are the amount of variance accounted for in that variable by all other common factors combined.
   (b) Following this initial estimation, communality estimates are used to replace the diagonal elements of the correlation matrix (where PCA uses 1.0 on the diagonal elements signifying the expectation of perfect reliability of measurement). **This substitution is important, as it acknowledges the realistic expectation of imperfect measurement.** A new set of factors and communality coefficients are then estimated and the process is repeated iteratively until the communality coefficients stabilize- or change less than a pre-determined threshold.

(ii) **Maximum Likelihood (ML):** used in logistic regression, confirmatory factor analysis, structural equation modeling, etc. It seeks to extract factors and parameters that optimally reproduce the population correlation (or covariance) matrix.

(a) Starting with an assumption that individual variables are normally distributed (leading to multivariate normal distributions). If a certain number of factors are extracted to account for inter-relationships between the observed variables, then that information can be used to reconstruct a reproduced correlation matrix.

(b) The parameters chosen are tweaked iteratively in order to maximize the likelihood of reproducing the population correlation matrix- or to minimize the difference between the reproduced and population matrices.

(c) This technique is particularly sensitive to quirks in the data, particularly in "small" samples, so if the assumptions of normality are not tenable, this is probably not a good extraction technique.

(iii) **MINRES:** The minimum residual solution is an unweighted least squares solution that takes a slightly different approach. It uses the optim function and adjusts the diagonal elements of the correlation matrix to mimimize the squared residual when the factor model is the eigen value decomposition of the reduced matrix. MINRES and PA will both work when ML will not, for they can be used when the matrix is singular. At least on a number of test cases, the MINRES solution is slightly more similar to the ML solution than is the PA solution.

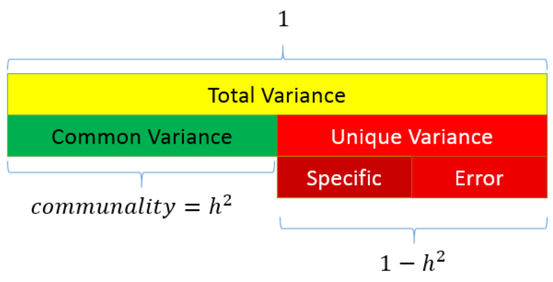
(iv) **Unweighted least squares and Generalized least squares:** use variations on the same process of ML; ULS is more robust to non-normal data / GLS weights variables with higher correlations more heavily which can contribute to sensitivity to problematic data.

(v) **Alpha factoring:** seeks to maximize the Cronbach's alpha estimate of the reliability of a factor. The difference between alpha extraction and other extraction techniques is the goal of the generalization. ML and other similar extraction techniques seek to generalize from a sample to a population of individuals, whereas alpha extraction seeks to generalize to a population of measures.

(vi) **Initial communalities vs extracted communalities:** we can expect to see differences in communalities and eigenvalues across extraction techniques. This is the proportion of each variable's variance that can be explained by the factors. It is also noted as h^2 and can be defined as the sum of squared factor loadings for the variables.

(a) **In PCA:** the initial communalities are always 1.00 for all variables.

(b) **In EFA:** initial communalities are estimates of the variance in each variable that will be accounted for by all factors, arrived at either by PCA analysis or as some form of multiple regression type analysis. For example, initial communalities are estimated by a multiple regression equation predicting each variable from all other variables (multiple R^2). Extracted communalities are calculated as the variance accounted for in each variable by all extracted factors. Looking at a table of factor loadings, with variables as the rows and factor loadings in columns, communalities are row statistics. Squaring and summing each factor loading for a variable should equal the extracted communality (within reasonable rounding error).



(vii) **Initial eigenvalues vs extracted eigenvalues vs rotated eigenvalues:** Eigenvalues are column statistics—again imagining a table of factor loadings, if you square each factor loading and sum them all within a column, you should get the eigenvalue for that factor (again within rounding error).

Thus, **eigenvalues are higher when there are at least some variables with high factor loadings, and lower when there are mostly low loadings.** Eigenvalues and communalities change from initial statistics to extraction (which are estimates and should be identical regardless of extraction method as long as the extraction method is a true factor analysis extraction, not PCA), which will vary depending on the mathematics of the extraction.

The **cumulative percent variance accounted for by the extracted factors will not change once we rotate the solution but the distribution of that percent variance will change as the factor loadings change with rotation**. Thus, if the extracted eigenvalues account for a cumulative 45% of the variance overall, once rotation occurs, the cumulative variance accounted for will still be 45%, but that 45% might be redistributed across factors.

1.1.1. How many factors should be extracted and retained?

(i) **Kaiser criterion:** proposed that an eigenvalue greater than 1 is a good lower bound for expecting a factor to be meaningful. Eigenvalue represents the sum of the squared loadings in a column (to get a sum of 1 or more, one must have rather large factor loadings to square and sum). Less impressive as more items are analyzed.

(ii) **Scree plot:** graph of eigenvalues; look for the natural elbow in the data where the slope of the curve changes (flattens).

(iii) **Parallel analysis:** generate random uncorrelated data and compare eigenvalues from the EFA to those from random data. Factors with eigenvalues significantly above the mean (or the 95 percentile) of the random eigenvalues should be retained.

(iv) **Minimum Average Partial (MAP):** involves partialing out common variance as each successive component is created; a familiar concept to those steeped in the traditions of multiple regression. As each successive component is partialed out, common variance will decrease to a minimum. At that point, unique variance is all that remains: that minimum point should be considered the criterion for the number of factors to extract.

1.2. Rotation in EFA

The goal is to clarify the factor structure and make the results of your EFA most interpretable. Rotation means that the axes are being rotated so that the clusters of items fall as closely as possible to them. There are several different rotation methodologies, falling into two general groups: orthogonal rotations and oblique rotations.

(i) **Orthogonal rotations** keep axes at a 90° angle, i.e., forcing the factors to be uncorrelated.
   (a) **Varimax:** rotation seeks to maximize the variance within a factor (within a column of factor loadings) such that larger loadings are increased and smaller are minimized.
   (b) **Quartimax:** tends to focus on rows, maximizing the differences between loadings across factors for a particular variable—increasing high loadings and minimizing small loadings.
   (c) **Equimax:** is considered a compromise between Varimax and Quartimax, in that it seeks to clarify loadings in both directions.

(ii) **Oblique rotations** allow angles that are not 90°, thus allowing factors to be correlated if that is optimal for the solution. When using oblique rotation we receive both a pattern matrix and structure matrix.
   (a) **Promax:** combination of an initial Varimax rotation to clarify the **pattern** of loadings, and then a procrustean rotation.
   (b) **Direct Oblimin**

All extracted factors are initially orthogonal, but remain so only as long as the rotation is orthogonal. However, even when the factors themselves are orthogonal, factor scores are often not uncorrelated despite the factors being orthogonal.

Factor matrix coefficients -> unrotated factor loadings (not of interest)

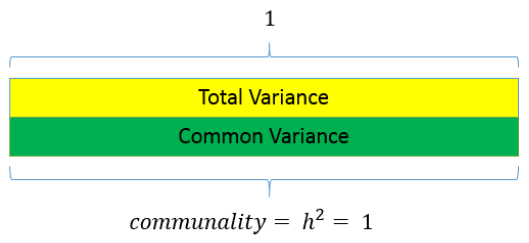
Pattern matrix coefficients ->

2. Principal Components Analysis (PCA)

Computes the analysis without regard to the underlying latent structure of the variables, using all the variance in the manifest variables. There is a fundamental assumption made when choosing PCA: the measured variables are themselves of interest, rather than some hypothetical latent construct (as in EFA). This makes PCA similar to multiple regression in some ways, in that it seeks to create optimized weighted linear combinations of variables.

- Assumes that all variables are measured without error (an untenable assumption in almost any discipline).
- Parameters are selected in an attempt to reproduce sample, rather than population characteristics (Thompson, 2004).

Unlike factor analysis, principal components analysis or PCA makes the assumption that there is no unique variance, the total variance is equal to common variance. Recall that variance can be partitioned into common and unique variance. If there is no unique variance then common variance takes up total variance (see figure below). Additionally, if the total variance is 1, then the common variance is equal to the communality.



3. Machine Learning: High bias - low variance

3.1. Introduction

ML software packages: scikit-learn, tensorflow, Pytorch, Keras

(i) Supervised learning: concerns learning from labeled data (for example, a collection of pictures labeled as containing a cat or not containing a cat). Common supervised learning tasks include classification and regression.

(ii) Unsupervised learning: is concerned with finding patterns and structure in unlabeled data. Examples of unsupervised learning include clustering, **dimensionality reduction**, and generative modeling.

(iii) Reinforcement learning: an agent learns by interacting with an environment and changing its behavior to maximize its reward.

3.2. Why is ML difficult?

A. Setting up a problem in ML
Ingredients:

- Dataset $D = (X, y)$, being X the matrix of independent variables and y the vector of dependent
- Model $f(x; \theta)$, is a function used to predict an output from a vector of input variables
- Cost function $C(y, f(X; \theta))$, allows to judge how well the model performs on the observations y. The model is fit by finding the value of θ that minimizes the cost function. One commonly used cost function is the squared error. Minimizing the squared error cost function is known as the method of least squares, and is typically appropriate for experiments with Gaussian measurement errors.

Recipe:

(i) Randomly divide the dataset D into two mutually exclusive groups D_{train} and D_{test} called the training and test sets.

(ii) The model is fit by minimizing the cost function using only the data in the training set $\hat{\theta} = argmin_\theta C(y_{train}; f(X_{train}; \theta))$. Finally, the performance of the model is evaluated by computing the cost function using the test set $C(y_{test}; f(X_{test}; \hat{\theta}))$. The value of the cost function for the best fit model on the training set is called the in-sample error $E_{in} = C(y_{train}; f(X_{train}; \theta))$ and the value of the cost function on the test set is called the out-of-sample error $E_{out} = C(y_{test}; f(X_{test}; \theta))$.

One of the most important observations we can make is that the out-of-sample error is almost always greater than the in-sample error, i.e. $E_{out} \leq E_{in}$.Splitting the data into mutually exclusive training and test sets provides an unbiased estimate for the predictive performance of the model; this is known as **cross-validation**.

B. Polynomial regression

It may be at first surprising that the model that has the lowest out-of-sample error E_{out} usually does not have the lowest in-sample error E_{in}. Therefore, if our goal is to obtain a model that is useful for prediction we may not want to choose the model that provides the best explanation for the current observations. **Jupyter notebook: ML IS DIFFICULT.ipynb**

At small sample sizes, noise can create fluctuations in the data that look like genuine patterns. Simple models (like a linear function) cannot represent complicated patterns in the data, so they are forced to ignore the fluctuations and to focus on the larger trends. Complex models with many parameters can capture both the global trends and noise-generated patterns at the same time. In this case, the model can be tricked into thinking that the noise encodes real information. **This problem is called "overfitting"** and leads to a steep drop-off in predictive performance. We can guard against overfitting in two ways: (1) we can use less expressive models with fewer parameters, or (2) we can collect more data so that the likelihood that the noise appears patterned decreases.

Bias-variance tradeoff: The simpler model has more "bias" but is less dependent on the particular realization of the training dataset, i.e. less "variance".

3.3. Basics of statistical learning theory

We begin with an unknown function $y = f(x)$ and fix a hypothesis set H consisting of all functions we are willing to consider, defined also on the domain of f. The function $f(x)$ produces a set of pairs (x_i, y_i), $i = 1, ..N$, which serve as the observable data. Once an appropriate error function E is chosen for the problem under consideration (e.g. sum of squared errors in linear regression), we can define two distinct performance measures of interest. The in-sample error, E_{in}, and the out-of-sample or generalization error, E_{out}. Models with large difference between E_{in} and E_{out} are said to overfit the data.

A. Basic intuitions from Statistical Learning Theory

The in-sample error will increase with the number of data points, because our models are not powerful enough to learn the true function we are seeking to approximate. In contrast, the out-of-sample error will decrease with the number of data points. As the number of data points gets large, the sampling noise decreases and the training data set becomes more representative of the true distribution from which the data is drawn. For this reason, in the infinite data limit, the in-sample and out-of-sample errors must approach the same value, which is called the **"bias"** of our model.

The bias represents the best our model could do if we had an infinite amount of training data to beat down sampling noise. The bias is a property of the kind of functions, or model class, we are using to approximate f(x). In general, **the more complex**

the model class we use, the smaller the bias. However, we do not generally have an infinite amount of data. For this reason, **to get best predictive power it is better to minimize the out-of sample error, E_{out}, rather than the bias. The bias always decreases with model complexity, but the variance, i.e. fluctuation in performance due to finite size sampling effects, increases with model complexity.**

B. Bias-variance decomposition

Consider a dataset $D = (X, y)$ consisting of the N pairs of independent-dependent variables. True data is generated from a noisy model $y = f(x) + \epsilon$, where ϵ is Gaussian with mean zero and σ_ϵ. Assume that we have a statistical procedure (e.g. least squares regression) for forming a predictor $f(x; \hat{\theta})$ that gives the prediction of our model for a new data point x.This estimator is chosen by minimizing a cost function which we take to be the squared error

$$C(y, f(X; \theta)) = \sum_i (y_i - f(x_i; \theta))^2. \tag{1}$$

The estimates for the parameters,

$$\hat{\theta_D} = arg_\theta min C(y, f(X; \theta)), \tag{2}$$

are a function of the dataset D. Denote as E_D the expected value over all the possible datasets obtained by drawing N samples from the true data distribution. E_ϵ is the expectation value over the noise. Therefore, the **bias**

$$Bias^2 = \sum_i (f(x_i) - E_D[f(x_i; \hat{\theta_D}])^2, \tag{3}$$

measures the deviation of the expectation value of our estimator from the true value. The **variance**

$$Var = \sum_i (E_D[(f(x_i; \hat{\theta_D} - E_D[f(x_i; \hat{\theta_D})])^2], \tag{4}$$

measures how much our estimator fluctuates due to finite-sample effects. Then,

$$E_{out} = Bias^2 + Var + Noise, \qquad Noise = \sum_i \sigma_\epsilon^2 \tag{5}$$

3.4. Gradient descent and its generalizations

(1) Dataset **X**
(2) Model $g(\theta)$, which is a function of the parameters θ
(3) Cost function $C(X, g(\theta))$, that allows to judge how well the model $g(\theta)$ explains the observations **X**
(4) The model is fit by finding the values of θ that minimize the cost function

$\longrightarrow$ **Iteratively adjust the parameters θ in the direction where the gradient of the cost function is large and negative.** The training procedure ensures the parameters flow towards a local minimum of of the cost function.

A. Gradient descent and Newton's method

The function we wish to minimize is the cost function, $E(\theta) = C(X, g(\theta)) = \sum_{i=1}^{n} e_i(x_i, \theta)$. For linear regression e_i is just the mean square-error for data point i; for logistic regression, it is the cross-entropy. **Jupyter notebook: GRADIENT DESCENT.ipynb**

Simplest GD algorithm

(i) **How:** Initialize to some value θ_0 and iteratively update the parameters according to

$$v_t = \eta_t \nabla_\theta E(\theta_t), \qquad \theta_{t+1} = \theta_t - v_t \tag{6}$$

where $\nabla_\theta E(\theta_t)$ is the gradient of $E(\theta_t)$ and we have introduced a **learning rate**, η_t, that controls how big a step we should take in the direction of the gradient at time step t. It is clear that for sufficiently small choice of the learning rate this methods will converge to a local minimum (in all directions) of the cost function. However, choosing a small η_t comes at a huge computational cost. The smaller η_t, the more steps we have to take to reach the local minimum. In contrast, if η_t is too large, we can overshoot the minimum and the algorithm becomes unstable (it either oscillates or even moves away from the minimum).

(ii) **Limitations:** (1) GD finds local minima of the cost function. Since the GD algorithm is deterministic, if it converges, it will converge to a local minimum of our energy function. Because in ML we are often dealing with extremely rugged landscapes with many local minima, this can lead to poor performance. (2) Gradients are computationally expensive to calculate for large datasets. (3) GD is very sensitive to choices of the learning rates. (4) GD treats all directions in parameter space uniformly. (5) GD is sensitive to initial conditions. One consequence of the local nature of GD is that initial conditions matter. Depending on where one starts, one will end up at a different local minimum. Therefore, it is very important to think about how one initializes the training process. (6) GD can take exponential time to escape saddle points, even with random initialization.

B. Variants of GD that address many of these shortcomings

(i) **Stochastic gradient descent (SGD) with mini-batches**: Stochasticity is incorporated by approximating the gradient on a subset of the data called a **minibatch**. The size of the minibatches is almost always much smaller than the total number of data points n, with typical minibatch sizes ranging from ten to a few hundred data points. If there are n points in total, and the mini-batch size is M, there will be n/M minibatches. Let us denote these minibatches by B_k where $k = 1, ..., n/M$. Thus, in SGD, at each gradient descent step we approximate the gradient using a single minibatch B_k,

$$\nabla_\theta E(\theta) = \sum_{i=1}^{n} \nabla_\theta e_i(x_i, \theta) \qquad \rightarrow \qquad \sum_{i \varepsilon B_k} \nabla_\theta e_i(x_i, \theta). \tag{7}$$

We then cycle over all k minibatches one at a time, and use the mini-batch approximation to the gradient to update the parameters θ at every step k. A full iteration over all n data points, in other words using all n/M minibatches, is called an epoch.

$$\nabla_\theta E^{MiniBatch}(\theta) = \sum_{i\varepsilon B_k} \nabla_\theta e_i(x_i, \theta) \tag{8}$$

$$v_t = \eta_t \nabla_\theta E^{MiniBatch}(\theta), \qquad \theta_{t+1} = \theta_t - v_t \tag{9}$$

In SGD, we replace the actual gradient over the full data at each gradient descent step by an approximation to the gradient computed using a minibatch. This has two important benefits: (1) it introduces stochasticity and decreases the chance that our fitting algorithm gets stuck in isolated local minima and (2) it significantly speeds up the calculation as one does not have to use all n data points to approximate the gradient.

(ii) **SGD with momentum**: One problem with gradient descent is that it has no memory of where the "ball rolling down the hill" comes from. This can be an issue when there are many shallow minima in our landscape. If we make an analogy with a ball rolling down a hill, the lack of memory is equivalent to having no inertia or momentum (i.e. completely overdamped dynamics). Without momentum, the ball has no kinetic energy and cannot climb out of shallow minima. Momentum becomes especially important when we start thinking about stochastic gradient descent with noisy, stochastic estimates of the gradient. In this case, we should remember where we were coming from and not react drastically to each new update: $0 \leq \gamma \leq 1$ serves as a memory of the direction we are moving in parameter space.

$$v_t = \gamma v_{t-1} + \eta_t \nabla_\theta E(\theta_t), \qquad \theta_{t+1} = \theta_t - v_t \tag{10}$$

The gradient is to be taken over a different mini-batch at each step; v_t is a running average of recently encountered gradients and $(1-\gamma)^{-1}$ sets the characteristic time scale for the memory used in the averaging procedure.

(iii) **Nesterov Accelerated Grdient (NAG)**: rather than calculating the gradient at the current parameters, $\nabla_\theta E(\theta_t)$, one calculates the gradient at the expected value of the parameters given our current momentum, $\nabla_\theta E(\theta_t + \gamma v_{t-1})$. Allows larger learning rate than GDM for the same choice of γ:

$$v_t = \gamma v_{t-1} + \eta_t \nabla_\theta E(\theta_t + \gamma v_{t-1}), \qquad \theta_{t+1} = \theta_t - v_t \tag{11}$$

(iv) **Second moment of the gradient**: The learning rate is limited by the steepest direction which can change depending on the current position in the landscape. To circumvent this problem, ideally our algorithm would keep track of curvature and take large steps in shallow, flat directions and small steps in steep, narrow directions. Second-order methods accomplish this by calculating or approximating the Hessian and normalizing the learning rate by the curvature. However, this is very computationally expensive for models with extremely large number of parameters. Ideally, we would like to be able to adaptively change the step size to match the landscape without paying the steep computational price of calculating or approximating Hessians.

(a) **RMSprop**:

$$g_t = \nabla_\theta E(\theta_t) \qquad s_t = \beta s_{t-1} + (1-\beta) g_t^2 \qquad \theta_{t+1} = \theta_t - \eta_t \frac{g_t}{\sqrt{s_t + \varepsilon}} \tag{12}$$

where β controls the avg time of the second moment (approx. 0.9), η_t is the learning rate to be 10^{-3} and $\varepsilon \approx 10^{-8}$ is a small regularization constant to prevent divergences. The learning rate is reduced in directions where the gradient is consistently large.

C. Practical tips

- **Randomize the data when making mini-batches**. It is always important to randomly shuffle the data when forming mini-batches. Otherwise, the gradient descent method can fit spurious correlations resulting from the order in which data is presented.
- **Standardize the data by subtracting the mean and normalizing the variance of input variables. Whenever possible, also decorrelate the inputs.** Thus, by standardizing the inputs, we are ensuring that the landscape looks homogeneous in all directions in parameter space. Since most deep networks can be viewed as linear transformations followed by a non-linearity at each layer, we expect this intuition to hold beyond the linear case.
- **Monitor the out-of-sample performance.** Always monitor the performance of your model on a validation set (a small portion of the training data that is held out of the training process to serve as a proxy for the test set). **If the validation error starts increasing, then the model is beginning to overfit**. Terminate the learning process. This early stopping significantly improves performance in many settings.
- **Adaptive optimization methods do not always have good generalization.**

3.5. Overview of Bayesian Inference

Bayesian methods are based on the fairly simple premise that probability can be used as a mathematical framework for describing uncertainty.

A. Bayes rule: To solve a problem using Bayesian methods, we have to specify two functions: the likelihood function $p(X|\theta)$, which describes the probability of observing a dataset X for a given value of the unknown parameters θ, and the prior distribution $p(\theta)$, which describes any knowledge we have about the parameters before we collect the data. The posterior distribution $p(\theta|X)$, describes our knowledge about the unknown parameter θ after observing the data X:

$$p(\theta|X) = \frac{p(X|\theta)p(\theta)}{\int d\theta' p(X|\theta')p(\theta')} \tag{13}$$

The likelihood function $p(X|\theta)$ is a common feature of both classical statistics and Bayesian inference, and is determined by the model and the measurement noise. Many common statistical procedures such as least-square fitting can be cast as Maximum Likelihood

Estimation (MLE). In MLE, one chooses the parameters $\hat{\theta}$ that maximize the likelihood (or equivalently the log-likelihood since log is a monotonic function) of the observed data

$$\hat{\theta} = arg_{\theta} max log p(X|\theta) \tag{14}$$

we choose the parameters that maximize the probability of seeing the observed data given our generative model. The prior distribution, by contrast, is uniquely Bayesian. There are two general classes of priors: if we do not have any specialized knowledge about θ before we look at the data then we would like to select an uninformative prior that reflects our ignorance, otherwise we should select an informative prior that accurately reflects the knowledge we have about θ.

Using an informative prior tends to decrease the variance of the posterior distribution while, potentially, increasing its bias. This is beneficial if the decrease in variance is larger than the increase in bias. In high dimensional problems, it is reasonable to assume that many of the parameters will not be strongly relevant. **Therefore, many of the parameters of the model will be zero or close to zero.** We can express this belief using two commonly used priors: the Gaussian prior

$$p(\theta|\lambda) = \prod_j \sqrt{\frac{\lambda}{2\pi}} e^{-\lambda\theta_j^2} \tag{15}$$

is used to express the assumption that many of the parameters will be small, and the Laplace prior

$$p(\theta|\lambda) = \prod_j \frac{\lambda}{2} e^{-\lambda|\theta_j|} \tag{16}$$

is used to express the assumption that many of the parameters will be zero.

B. Bayesian decisions In most cases we need to summarize our knowledge and pick a single "best" value for the parameters. In principle, the specific value of the parameters should be chosen to maximize a utility function.

C. Hyperparameters The Gaussian and Laplace prior distributions, used to express the assumption that many of the model parameters will be small or zero, both have an extra parameter λ. This hyperparameter or nuisance variable has to be chosen somehow. One standard Bayesian approach is to define another prior distribution for λ, usually using an uninformative prior, and to average the posterior distribution over all choices of λ. This is called a hierarchical prior.

3.6. Linear regression

The optimal choice of predictor depended on, among many other things, the functions used to fit the data and the underlying noise level.

Dataset with n samples: $D = \{y_i, x^{(i)}\}_{i=1}^{n}$, where $x^{(i)}$ is the i-th observation vector and y_i the response. Every sample has p features. The samples are generated via the true function f, where ε_i is kindof white noise with zero mean and finite variance:

$$y_i = f(x^{(i)}; w_{True}) + \varepsilon_i = w_{True}^T x^{(i)} + \varepsilon_i \tag{17}$$

we cast all the samples into a nxp matrix with the rows being observations $X_{i,:} = x^{(i)}$, and the columns being measured features $X_{:,j}$. In linear regression We don't know f, but we assume its form.

We want to find a function g with parameters w fit to the data D that can best approximate f (which is the true function). When this is done, meaning we have found a $\hat{w}$ such that $g(x; \hat{w})$ yields our best estimate of f, we can use this g to make predictions about the response y_0 for a new data point x_0. We define the L^p norm of a vector $x = (x_1, ..., x_d)$, $||x||_p = (|x_1|^p, ..., |x_d|^p)^{1/p}$.

A. Least square regression

Ordinary least squares linear regression **(OLS)** is defined as the minimization of the L_2 norm of the difference between the response y_i and the predictor $g(x^{(i)}; w) = w^T x^{(i)}$

$$\hat{w_{LS}} = (X^T X)^{-1} X^T y \tag{18}$$

Average generalization error

$$|E_{in} - E_{out}| = 2\sigma^2 \frac{p}{n} \tag{19}$$

for high-dimensional data (large p), the avg generalization error is large, thus meaning the model is not learning. One way to aminorate this is to use regularization.

(i) **Ridge regression** uses L_2 penalty: adding to the least squares loss function a regularizer defined as the L_2 norm of the parameter vector we wish to optimize over.

(ii) **LASSO** uses L_1 penalty: stands for "least absolute shrinkage and selection operator".

How different are the solutions found using LASSO and Ridge regression? In general, LASSO tends to give sparse solutions, meaning many components of $\hat{w}_{LASSO}$ are zero.

3.7. Logistic regression

A wide variety of problems, such as classification, are concerned with outcomes taking the form of discrete variables (i.e. categories).

(i) Define Logistic Regression and derive its corresponding cost function (Cross entropy) using Bayesian approach and discuss its minimization

(ii) Generalize Logistic Regression to the case of **multiple categories**: **SoftMax** regression

A. The cross-entropy as a cost function for logistic regression

Dependent variables: y_i, discrete and take values from $m = 0, .., M-1$, which enumerate the M classes.The goal is to predict the output classes from the design matrix X (nxp) made of n samples, each of which bears p features.

Perceptron: example of hard classification; each datapoint is assigned to a category ($y_i = 0$ or $y_i = 1$). Favorable in many cases, e.g. when dealing with noisy data.Sign function. **Soft classifier:** Logistic (sigmoid) function

$$\sigma(s) = \frac{1}{1+e^{-s}}, \qquad 1-\sigma(s) = \sigma(-s) \tag{20}$$

The probability that a datapoint x_i belongs to a category $y_i = \{0, 1\}$ is given by

$$P(y_i = 1|x_i, \theta) = \frac{1}{1+e^{-x_i^T\theta}}, \qquad P(y_i = 0|x_i, \theta) = 1 - P(y_i = 1|x_i, \theta) \tag{21}$$

where $\theta = w$ are the **weights we wish to learn from the data.**

We now define the **cost function for logistic regression** using **Maximum Likelihood Estimation (MLE)**. Recall, that in MLE we choose parameters to maximize the probability of seeing the observed data. The cost (error) function is just the negative log-likelihood (right-hand side is known as the **cross-entropy**).

$$C(w) = -l(w) = \sum_{i=1}^{n} -y_i log\sigma(X_i^T w) - (1-y_i)log[1-\sigma(x_i^T w)] \tag{22}$$

We usually supplement the cross-entropy with additional regularization terms L_1 and L_2.

B. Minimizing the cross-entropy: it is a convex function of the weights. Therefore, any local minimizer is a global minimizer.

$$0 = \nabla C(w) \tag{23}$$

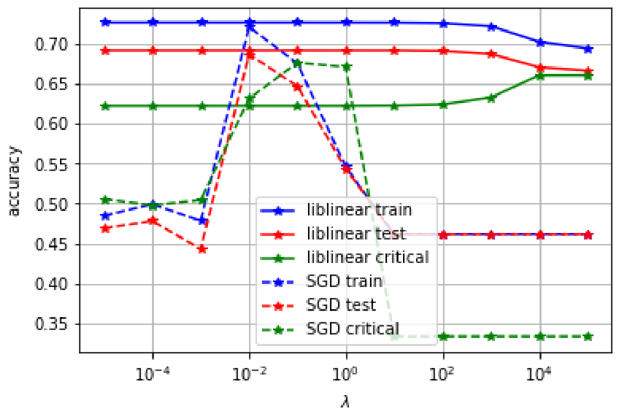
ISING model: classify, given an Ising state, if it corresponds to an ordered or disordered phase. To do so, use:

liblinear solver

Stochastic Gradient Descent to optimize the logistic regression cost function with L_2 regularization

```
from sklearn import linear_model
from sklearn.neural_network import MLPClassifier
# define logistic regressor
logreg=linear_model.LogisticRegression(C=1.0/lmbda,random_state=1,
                                 verbose=0,max_iter=1E3,tol=1E-5,
                                                  solver='liblinear')
# fit training data
logreg.fit(X_train, Y_train)
```

Comparing the accuracy on the training and test data, we evaluate the degree of overfitting: *Small degree of overfitting = training and test curves very close to each other for different λ parameters*



SUSY: use logistic regression in an attempt to find the relative probability that an event is from a signal or a background event $\rightarrow$ **TensorFlow**

C. SOFTMAX Regression: generalize logistic regression to multi-class classification. One approach is to treat the label as a vector y_i, namely a binary string of length M with only one component of y_i being 1 and the rest 0: $y_i = (1, 0, ..., 0)$ meaning the sample x_i belongs to class 1.

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(C=1e5,
                         multi_class='multinomial',
                              penalty='l2', solver='sag', tol=0.1)
clf.fit(X_train, y_train)
```

The probability of x_i being in class m' is given by the **SOFTMAX** function:

$$P(y_{im'} = 1|x_i, \{w_k\}_{k=0}^{M-1}) = \frac{e^{-x_i^T w_{m'}}}{\sum_{m=0}^{M-1} e^{-x_i^T w_m}}, \tag{24}$$

where $y_{im'}$ refers to the m'th component of the vector y_i. Therefore, the likelihood of this M-class classifier is simply:

$$P(D|\{w_k\}_{k=0}^{M-1}) = \prod_{i=1}^{n}\prod_{m=0}^{M-1} [P(y_{im} = 1|x_i, w_m)]^{y_{im}} \times [1 - P(y_{im} = 1|x_i, w_m)]^{1-y_{im}}, \tag{25}$$

from which we define the cost function

$$C(w) = -\sum_{i=1}^{n}\sum_{m=0}^{M-1} y_{im} log P(y_{im} = 1|x_i, w_m) + (1 - y_{im}) log(1 - P(y_{im} = 1|x_i, w_m)). \tag{26}$$

For $M = 1$ we recover the cross entropy for logistic regression.

3.8. Combining models

Use of ensemble methods that combine predictions from multiple, often weak, statistical models to improve predictive performance: random forests, boosted gradient trees (XGBoost), etc.

A. Bias-Variance tradeoff for an ensemble of classifiers: The bias-variance tradeoff summarizes the fundamental tension in machine learning between the complexity of a model and the amount of training data needed to fit it. Since data is often limited, in practice it is frequently useful to use a less complex model with higher bias - a model whose asymptotic performance is worse than another model - because it is easier to train and less sensitive to sampling noise arising from having a finite-sized training dataset (i.e. smaller variance).

Key property (!!): correlation between models that constitute the ensemble. The degree of correlation between models is important for two distinct reasons.

A. Holding the ensemble size fixed, averaging the predictions of correlated models reduces the variance less than averaging uncorrelated models.
B. In some cases, correlations between models within an ensemble can result in an increase in bias, offsetting any potential reduction in variance gained from ensemble averaging.

Bias-Variance tradeoff:

1. Bias-Variance tradeoff for a single model: data $X_L = \{(y_i, x_j), j = 1, .., N\}$, true data is generated from a noisy model $y = f(x) + \epsilon$, where ϵ is normaly distributed with mean zero

and std dev. σ_ϵ. Assume an statistical procedure (e.g. LeastSquare Reg.) for forming a predictor $\hat{g_L}(x)$ that gives the prediction of our model for a new data point x given that we trained the model using a dataset L. This estimator is chosen by minimizing a cost function which, for the sake of concreteness, we take to be the squared error

$$C(X, g(x)) = \sum_i (y_i - \hat{g_L}(x_i))^2. \tag{27}$$

Many datasets $\{L_i\} \to$ corresponding estimators $\hat{g_{L_j}}(x)$ differ from each other due to stochastic effects arising from sampling noise.

$$E_{L,\epsilon} = Bias^2 + Var + Noise, \qquad Noise = \sum_i \sigma_\epsilon^2 \tag{28}$$

$$Bias^2 = \sum_i (f(x_i) - E_L[\hat{g_L}(x_i)])^2, \quad Var = \sum_i (E_L[(\hat{g_L}(x_i) - E_L[\hat{g_L}(x_i)])^2]. \tag{29}$$

2. Bias-Variance tradeoff for an ensemble: So far the calculation for ensembles is almost identical to that of a single estimator. However, since the aggregate estimator is a sum of estimators, its variance implicitly depends on the correlations between the individual estimators in the ensemble.
   Aggregate ensemble predictor (given a learning algorithm A that generates a model $A(\theta, L)$:
   $$\hat{g}_L^A(x_i, \{\theta\}) = \frac{1}{M}\sum_{m=1}^{M} \hat{g}_L(x_i, \theta_m) \tag{30}$$
   $$Bias^2 = (f(x) - E_{L,\theta}[\hat{g_L}^A(x, \{\theta\})])^2, \tag{31}$$
   $$Var = (E_{L,\theta}[(\hat{g_L}^A(x, \{\theta\}) - E_{L,\theta}[\hat{g_L}(x, \{\theta\})])^2] = \rho(x)\sigma_{L,\theta}^2 + \frac{1-\rho(x)}{M}\sigma_{L,\theta}^2 \tag{32}$$
   Notice that by using large ensembles ($M \to \infty$), we can significantly reduce the variance, and for completely random ensembles where the models are uncorrelated ($\rho(x) = 0$), maximally suppresses the variance. Thus, using the aggregate predictor beats down fluctuations due to finite-sample effects. **The key is to decorrelate the models as much as possible while still using a very large ensemble**. When models in the ensemble are completely random, the bias of the aggregate predictor is just the expected bias of a single model. Thus, for a random ensemble one can always add more models without increasing the bias.
3. Ensembles: ensemble methods shortcomings
   (a) Statistical: When the learning set is too small, a learning algorithm can typically find several models in the hypothesis space H that all give the same performance on the training data. Provided their predictions are uncorrelated, averaging several models reduces the risk of choosing the wrong hypothesis.

(b) Computational: Many learning algorithms rely on some greedy assumption or local search that may get stuck in local optima. As such, an ensemble made of individual models built from many different starting points may provide a better approximation of the true unknown function than any of the single models.

(c) Representational: In most cases, for a learning set of finite size, the true function cannot be represented by any of the candidate models in H. By combining several models in an ensemble, it may be possible to expand the space of representable functions and to better model the true function.

We should try to **randomize ensemble construction as much as possible to reduce the correlations between predictors** in the ensemble. This ensures that **our variance will be reduced while minimizing an increase in bias due to correlated errors**. Second, the ensembles will work best for procedures where the error of the predictor is dominated by the variance and not the bias. Thus, these methods are especially well suited for unstable procedures whose results are sensitive to small changes in the training dataset.

B. BAGGing (Bootstrap AGGregation): given a very large dataset L, we can chop into smaller datasets M. If each partition is sufficiently large to learn a predictor, we can create an ensemble aggregate predictor composed of predictors trained on each subset of the data. **Bagging is effective on "unstable" learning algorithms where small changes in the training set result in large changes in predictions**. The contribution of all predictors is weighted equally in the bagged (aggregate) predictor.

Continuous predictors (regression-like): predictor is just the average of all the individual predictors.

Classification tasks: each predictor predicts a class label j; the predictor is just a majority vote of all the individual predictors.

Bagging with a perceptron (linear classifier) as the base classifier that constitutes the elements of the ensemble. It is clear that, although each individual classifier in the ensemble performs poorly at classification, bagging these estimators yields reasonably good predictive performance.

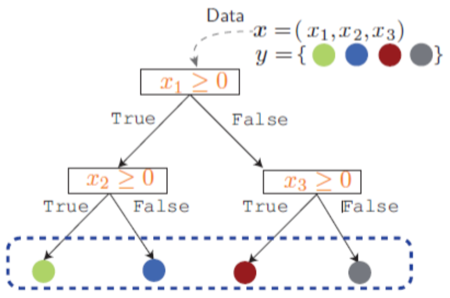
C. Boosting: Ensemble of weak classifiers $\{g_k(x)\}$ is combined into an aggregate, boosted classifier. **Unlike bagging**, each classifier is associated with a weight α_k that indicates how much it contributes to the aggregate classifier. It works best when we combine simple, high-variance classifiers into a more complex whole.

$$g_A(x) = \sum_{K=1}^{M} \alpha_k g_k(x) \tag{33}$$

Adaptive boosting (AdaBoost): The basic idea behind AdaBoost, is to form the aggregate classifier in an iterative process. Importantly, at each iteration we reweight the error function to "highlight" data points where the aggregate classifier performs poorly (so that in the next round the procedure put more emphasis on making those right.) In this way, we can successively ensure that our classifier has good performance over the whole dataset.

D. Random Forest: Ensemble method widely deployed for complex classification tasks. A random forest is composed of a family of (randomized) tree-based classifier decision trees. **Decision trees are high-variance, weak classifiers that can be easily randomized**, and as such, are ideally suited for ensemble-based methods.

(!!) A **decision tree** uses a series of questions to hierarchically partition the data. Each branch of the decision tree consists of a question that splits the data into smaller subsets, with the leaves (end points) of the tree corresponding to the ultimate partitions of the data. When using decision trees for classification, **the goal is to construct trees such that the partitions are informative about the class label**. It is clear that more complex decision trees lead to finer partitions that give improved performance on the training set. However, **this generally leads to over-fitting, limiting the out-of-sample performance**. For this reason, in practice almost all decision trees use some form of regularization (e.g. maximum depth for the tree) to control complexity and reduce over-fitting. **Decision trees also have extremely high variance, and are often extremely sensitive to many details of the training data**. This is not surprising since decision trees are learned by partitioning the training data. Therefore, individual decision trees are weak classifiers. However, these same properties make them ideal for incorporation in an ensemble method.



In order to create an ensemble of decision trees, we must introduce a randomization procedure:

1. Use bagging and simply "bag" the decision trees by training each decision tree on a different bootstrapped dataset - this does not constitute a random forest but rather a bagged decision tree.
2. Only use a different random subset of the features at each split in the tree. This "feature bagging" is the distinguishing characteristic of random forests: reduces correlations between decision trees that can arise when only a few features are strongly predictive of the class label.

3. Extremized random forests (ERFs) combine ordinary and feature bagging with an extreme randomization procedure where splitting is done randomly instead of using optimality criteria.

Random Forests - we use the sci-kit learn implementation of random forests. There are two main **hyper-parameters** that will be important in practice for the performance of the algorithm and the degree to which it overfits/underfits: the number of estimators in the ensemble and the depth of the trees used. The former is controlled by the parameter n_estimators whereas the latter (the complexity of the trees used) can be controlled in many distinct ways (min_samples_split, dictates how many samples need to be in each node of the classification tree, etc.). The bigger this number, the more coarse our trees and data partitioning.

Extremely fine trees: min_samples_split=2

Extremelly coarse trees: min_samples_split=10000

For more complicated datasets, how can we choose the right hyperparameters? **out-of-bag (OOB) estimates**. Whenever we bag data, since we are drawing samples with replacement, we can ask how well our classifiers do on data points that are not used in the training. This is the out-of-bag prediction error and plays a similar role to cross-validation error in other ML methods. Since this is the best proxy for out-of-sample prediction, we choose hyperparameters to minimize the out-of-bag error.

```
#This is the random forest classifier
from sklearn.ensemble import RandomForestClassifier
#This is the extreme randomized trees
from sklearn.ensemble import ExtraTreesClassifier
classifier=RandomForestClassifier #(or ExtraTreesClassifier)
for i, leaf_size in enumerate(leaf_size_list):
    # Define Random Forest Classifier
    myRF_clf = classifer(
        n_estimators=min_estimators,
        max_depth=None,
        min_samples_split=leaf_size, # minimum number of sample per leaf
        oob_score=True,
        random_state=0,
        warm_start=True # this ensures that you add estimators
        without retraining everything
    )
    for j, n_estimator in enumerate(n_estimator_range):
        print('n_estimators: %i, leaf_size: %i'%(n_estimator,leaf_size))
        myRF_clf.set_params(n_estimators=n_estimator)
        myRF_clf.fit(X_train, Y_train)
```

E. Gradient Boosted Trees and Extreme Gradient Boosting - XGBoost: use intuition from boosting and gradient descent (in particular Newton's method) to construct ensembles of decision trees. Like in boosting, the ensembles are **created by iteratively adding new decision trees to the ensemble**. In gradient boosted trees, one critical component is the cost function that measures the performance of our ensemble. At each step, **we compute the gradient of the cost function with respect to the predicted value of the ensemble and add trees that move us in the direction of the gradient**. Of course, this requires a clever way of mapping gradients to decision trees.
One nice aspect of XGBoost (and ensemble methods in general) is that it is easy to visualize **feature importances**. In XGBoost, there are some handy plots for viewing these (similar functions also exist for the scikit implementation of random forests). One thing we can calculate is the feature importance score (Fscore), which measures how many times each feature was split on. The higher this number, the more fine-tuned the partitions in this direction, and presumably the more informative it is for our classification task.

```
import xgboost as xgb
XGBclassifier=xgb.sklearn.XGBClassifier(nthread=-1,seed=1,n_estimators=1000)
xgb.plot_importance(XGBclassifier, ax=plt.gca())
```

3.9. Neural Networks

Conceptually, it is helpful to divide **neural networks** into four categories:

1. General purpose neural networks for **supervised learning**
2. Neural networks designed specifically for **image processing**, the most prominent example of this class being Convolutional Neural Networks (CNNs)
3. Neural networks for **sequential data** such as Recurrent Neural Networks (RNNs)
4. Neural networks for **unsupervised learning** such as Deep Boltzmann Machines

A. Basics: Neural networks (also called neural nets) are neural-inspired **nonlinear models for supervised learning**. As we will see, neural nets can be viewed as natural, more powerful extensions of supervised learning methods such as linear and logistic regression and soft-max methods.

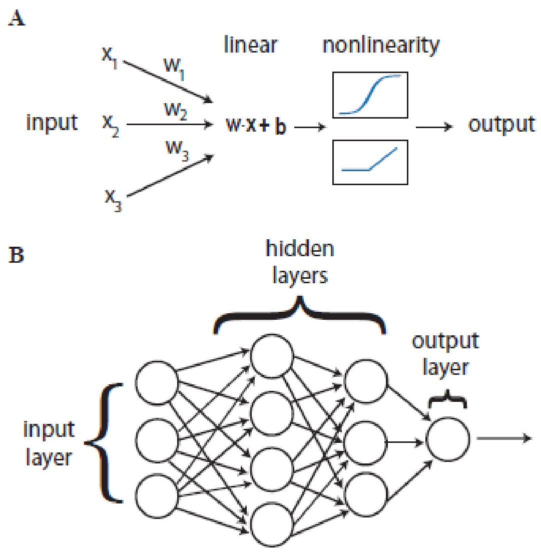
1. Basic bulding block (neuron): **Neuron** i takes a vector of d input features $x = (x_1, ..., x_d)$ and produces a **scalar output** $a_i(x)$. A neural network consists of many such neurons stacked into layers, with **the output of one layer serving as the input for the next**. The first layer in the neural net is called the **input layer**, the middle layers are often called **"hidden layers"**, and the final layer is called the **output layer**. The function a_i varies depending on the type of **non-linearity** used in the NN. However, a_i can be decomposed into a **linear operation** that **weights** the **relative importance** of the various inputs and a **non-linear transformation** $\sigma_i(z)$ which is usually the same for **all neurons**.

Linear transformation: takes the form of a **dot product** with a set of neuron-specific **weights** $w^{(i)} = (w_1^{(i)}, w_2^{(i)}, ..., w_d^{(i)})$ followed by re-centering with a neuron-specific **bias** $b^{(i)}$

$$z^{(i)} = w^{(i)} \cdot x + b^{(i)} = X^T \cdot W^{(i)}, \qquad X = (1, x), \quad W^{(i)} = (b^{(i)}, w^{(i)}). \tag{34}$$

The full **input-output function**

$$a_i(X) = \sigma_i(z^{(i)}). \tag{35}$$



Common choices of non-linearities $\sigma(z)$**:** we train NN with **gradient descent-based** methods $\rightarrow$ requires to take derivatives of the neural input-output function with respect to the weights $w^{(i)}$ and the bias $b^{(i)}$.

Step-functions (perceptrons)
Sigmoids (Fermi-functions)
Hyperbolic tangent
Rectified linear units (ReLUs)
Leaky rectified linear units (leaky ReLUs)
Exponential linear units (ELUs)

!! **Perceptrons** $\rightarrow$ derivative $= 0$ everywhere except where the input $= 0$. These behavior makes it impossible to train perceptrons using GD $\rightarrow$ use of hyperbolic tangent or sigmoid $\rightarrow$ when the input weights become large, the activation function saturates and the derivative of the output with respect to the weights tends to 0; these vanishing gradients make it harder to train NN $\rightarrow$ non-saturating activation function (**ReLUs** / **ELUs**), gradients stay finite even for large inputs.

2. Layering neurons to build deep networks (network architecture): In the simplest feed-forward networks, each neuron in the input layer of the neurons takes the inputs X and produces an output $a_i(X)$ that depends on its current weights. **The outputs of the input layer are then treated as the inputs to the next hidden layer**. This is usually repeated several times until one reaches the top or output layer. **The output layer is almost always a simple classifier**: **a logistic regression or soft-max function in the case of categorical data (i.e. discrete labels) or a linear regression layer in the case of continuous outputs.**

4. References

[1] Osborne, J. W. Best Practices in Exploratory Factor Analysis. Scotts Valley, CA: CreateSpace Independent Publishing (2014).

[2] Nielsen, Michael A (2015), Neural networks and deep learning (Determination Press).

[3] Goodfellow, Ian, Yoshua Bengio, and Aaron Courville (2016), Deep Learning (MIT Press) http://www.deeplearningbook.org.