**Chapter 4: Data Preprocessing: Practical Issues:**

* You can’t train and test on the same data because then your model may **overfit to the model** and therefore perform well on the training set, but poorly on new data. Therefore, you should split your data set into **training samples and test samples**. (70:30, 80:20, etc.)
  + **Development samples** are also a section worth having, where you adjust and fine tune your model until it performs well on the development data.
* When features have **vastly different scale**, (e.g. 1 < x1 < 10 -vs- 1 < x2 < 10,000) you have two different options to ensure that the one with the larger scale does not dominate the model:
  + **Normalization:** For some set X, there is a minimum we will call s (small). There is a maximum we will call b (big). To normalize each item (Xn) in X, this is the formula:
    - ***XnNORMALIZED = (Xn – s) / (b - s)***.
    - This will ensure all values lie **between zero and one**.
  + Standardization: This method modifies the data to fit the normal curve where the **mean is zero** and the **standard deviation is one**. This is done by using this formula, for some set X where each item is Xn:
    - ***XnSTANDARDIZED = (Xn – mean(X)) / standard deviation(X)***
    - Can be better than normalization because it deals better with outliers, and works better with weights that are set to zero by default, since the means are also zero
* L1 and L2 regularization: **reducing model complexity**:
  + **L1 regularization:** This form of regularization strongly favors higher weights, meaning it essentially disregards smaller weights (which will be set to zero), effectively selecting which weights have an effect in predictions. Thus, L1 yields sparse solutions.
    - This is great for data sets with many extraneous features.
    - ***||w|| = sum (|all weights|)***
  + L2 regularization: This form of regularization does not show the selectivity of L1 regularization.
    - It words better for sets with fewer extraneous features.
    - ***||w|| = sum (all weights^2)***
  + The purpose of regularization is to **set weights to zero** to reduce the complexity of the model.

**Chapter 5: Dimensionality Reduction:**

* **Principal component analysis:** Reduces the number of dimensions by finding the direction with the most variation, then basing other directions off the first one, then picking a set number of directions (dimensions) to consider.
  + Your data should be **normalized/standardized** before reducing dimensions!
  + First, plot all the data. **Find the longest path through which to traverse the data.** This is your first dimension. You can stop here and make your data one-dimensional.
  + Then, your other directions (dimensions) are perpendicular to this first one.
  + The variation in each dimension is known as the **principle component (PC)**, so you would have PC1, PC2, and so on.
  + **E.g.** If you have 3 features, you started out with 3 dimensions. You find the longest path (largest variation), and call it the dimension x. You then define the dimensions y and z with respect to x (as in perpendicular to it). Then you can pick the one with the largest PC out of y and z, and use only x and y, or x and z to plot all the data in the new two dimensions.
    - This was going from 3 to 2 dimensions.You can do the same with 14 dimensions, obtain 14 PCs, then pick the top ***k*** PCs, and use those to plot the data in ***k*** dimensions.
  + If I’m making no sense, here’s a nice link: <http://setosa.io/ev/principal-component-analysis/>
* Creating a variance-covariance matrix:
  + Say you have 3 dimensions: d1, d2, and d3. You can do a pairwise comparison between all of them to obtain a matrix of covariance.
  + An algorithm that helps me understand the calculation:
    - Covariance between di and dj (coDiDj) = zero
    - For each item x in both di and dj:
      * coDiDj += (di[x] – mean(di)) \* (dj[x] – mean(dj))
    - return coDiDj //The final covariance between di and dj
  + You end up with this matrix using d1, d2, and d3:
    - Covariance (d1, d1) Covariance (d1, d2) Covariance (d1, d3)
    - Covariance (d2, d1) Covariance (d2, d2) Covariance (d2, d3)
    - Covariance (d3, d1) Covariance (d3, d2) Covariance (d3, d3)
  + **Positive covariance:** as one increases/decreases, the other follows.
  + **Negative covariance:** as one increase, the other decreases, and vice versa.
* As we incorporate more PCs into our model, we can explain more of the variance, but then each individual PC contributes less and less to the variance.

Chapter 6: Model Evaluation and Hyperparameter Tuning:

* **The holdout method:** This is what we have been doing for our assignments the past two times, where we have a training set, a development set, and a test set, where we train on the training set and refine our model based on its performance on the data set. Then obtain a final evaluation based on the test set.
* **K-fold cross validation:** Split the data into ***k*** sections. Iterate through each section, using it as the test set, and using the rest as the training set. Reduces estimate variance.
  + The smaller the dataset, the larger ***k*** should be.
  + **Leave-one-out cross validation:** where ***k*** is equal to the number of samples. Largest possible k value.
  + **Stratified k-fold cross validation:** Class proportions preserved for each fold, works better for imbalanced data.
* To find the best parameters, we could use brute force search methods. However, this is certainly going to be computationally expensive.
* Accuracy (correct predictions / total predictions) may not be the best way to assess evaluate a model.
  + Use counts of True Positive (TP), True Negative (TN), False Positive (FP), and False Negative (FP).
  + In these terms, **accuracy** is calculated as:
    - ***(TP + TN) / (TP + TN + FP + FN)***
  + **Error** is 1 – accuracy, or
    - ***(FP + FN) / (TP + TN + FP + FN)***
  + **F1 score:**
    - Precision: ***TP / (TP + FP) = P***
    - Recall: ***TP / (TP + FN) = R***
    - Finally, the F1 score is: ***(2 \* P \* R) / (P + R)***

**Chapter 8: Working with Text:**

* **Bag of words models:** Each word gets a number/index. Creates a sparse feature vector.
* **Unigrams:** Each individual word has its index and is treated separately.
* **Bigrams:** Pairs of words (W[i] + W[i + 1]) are treated as individual parts and are treated separately. Like ***k***-mers of nucleotides, where ***k*** is 2, for you bioinformatics people.
* **Natural Language Processing (NLP) tricks:** Remove noisy words/phrases (ones that do not contribute to predictions), stemming, lemmatization, etc. (I’m hoping we don’t have to know what these individual tricks are).

**Chapter 11: Data Clustering: K-means and Hierarchical Clustering:**

* Data clustering is an **unsupervised learning** problem.
* Clustering only uses data on similarity between samples. Good clustering means that there is high similarity within the same cluster, and low similarity across clusters.
* Similarity measures (inversely related to distance):
  + Euclidean distance: ***distance (x, y) = sum ((x[dim i] – y[dim i])^2)***
  + Manhattan distance: ***distance (x, y) = sum (|x[dim i] – y[dim i]|)***
  + Kernelized (non-linear) distance: ***distance (x, y) = look it up.***
* **Flat/Partitional clustering:** partitions are independent of each other.
* **Hierarchical clustering:** Partitions can be within each other, defining more refined and specific clusters. Something like a phylogenetic tree of species, where further branching specifies smaller more particular clusters (bioinfo people).
* **K-means:** from a set X, create ***k*** clusters. First pick k random samples from X, and set them as the initial cluster centers. The remaining samples in X are added one by one to the cluster they are closest to. Each added sample requires computing the cluster center again, as the mean of the samples within the cluster thus far.
  + K-means seeks to minimize the sums of distances of all samples from their assigned cluster centers. Minimize ***sum (sum (distance of all points in some cluster to the center) for all clusters).***
  + Find a good value for ***k*** by trying different values, and finding the “elbow point.”
  + Initialization values strongly affect K-means. One method is to pick the mean of the entire sample as the first cluster, then the farthest sample from that, then the next farthest sample from them both, and so on, ***k*** times.
  + Try different initialization methods and pick the best one.
  + **Limitations include:**
    - Bad with non-convex/round shapes of clusters.
    - Does hard assignment. Either you’re in the cluster or you’re not.
    - Bad with outliers. Try K-medians?
* Hierarchical clustering can be agglomerative or divisive:
  + **Agglomerative:** starts with samples as singletons, combines them until one all-encompassing cluster remains.
  + **Divisive:** Start with the enter data set in one cluster, remove the most outlying group one at a time, until singletons remain.
* Measuring similarity between clusters:
  + **Min-link:** shortest distance between opposite cluster members, results in large clusters.
  + **Max-link:** longest distance between opposite cluster members, results in small round clusters. ***Average-link is a compromise!***
* Flat -vs- hierarchical clustering:
  + ***Flat*** gives us one set of partitions. ***Hierarchical*** lets us select a resolution as which to view our partitions. Therefore, ***flat*** needs to be given the number of clusters beforehand, and ***hierarchical*** does not.
  + ***Hierarchical*** is slower. There is no consensus on which is generally better.

**Chapter 12: Artificial Neural Networks:**

* **Deep learning:** Algorithms used to train artificial neural networks (ANNs).
* Perceptron formula: ***Change in weight = learning coefficient \* (prediction[i] – correct prediction[i]) \* sample[i]***
* **Adaline:** weights adjusted only after every sample has been visited, which is one epoch. Multiple epochs are conducted.
* **SGD:** updates weights after n samples.
* **Multi-layer feedforward neural network:** There are layers of nodes, where each layer, except the output layer, has an additional activation node. Nodes from lower layers point to nodes in higher ones.
* MLP learning procedure
  + Starting at the input layer, forward propagate x(i).
  + Calculate the error that we will want to minimize.
  + Find its derivative with respect to each weight.
  + Update the weights.