COMP0081 Applied Machine Learning cheat sheet

Timuel the Great

February 2019

Contents

Introduction

Cheat sheet for the COMP0081 Applied Machine Learning exam. This cheat sheet is meant to be used in preparation for the exam, **not during** the exam. Expected background knowledge:

- A

Terminology

- **m**.

1 Competitive Data Science

1.1 Preprocessing

The first step in any data science project is **preprocessing**. This usually involves feature extraction - there are many types of features, but we will focus on **numerical** and **categorical** features. To make inference and computations simpler, we often convert categorical features into numerical features using the process called *encoding*. There are several popular types of encoding:

- **One-hot encoding:** Presents a simple way to make labels compatible with neural nets. If number of classes is big, resulting one-hot might be very large (even though they are sparse).
- **Label encoding:** Involves a single `label` field set to the ID of the class. Not suitable for NNs but works for tree models. Can encode unknown classes as -1. The issue here is that

there might be absolutely no correlation between label encoding and the target value, since our encoding logic did not rely on the target value at all. This brings us to the next type of encoding...

- **Mean encoding:** Sets the encoded value to be the ratio of how many examples with that class have label 1 vs. how many examples have label 0.

- Frequency encoding, target encoding, learning of embeddings, etc.

For **numerical features**, you can do: scaling, rank transformation, clipping, non-linear monotonic transformations like $\log$.

1.2 Competition metrics

Different competitions use different metrics. Participants should aim to optimize these metrics directly since they are used to measure the performance of each submission. Sadly, not all metrics can be optimized directly, and it's important to understand how different metrics deal with extreme values. **Classification metrics:**

- **Accuracy:** Fraction of correctly classified examples. This doesn't work well with imbalanced classes (classifier can always pick the most frequent class). It also only takes into account hard predictions, and not the confidence in each class. Finally, it's impossible to optimize directly with gradient-based methods.

- **Area Under Curve (AUC):** Plot True Positive Rate (TPR $= \frac{TP}{TP+FN}$) against False Positive Rate (FPR $= \frac{FP}{FP+TN}$) then calculate the area under the curve using integration or just area of rectangles. AUC provides an aggregate measure of performance across all possible classification thresholds. One way of interpreting AUC is as the probability that the model ranks a random positive example more highly than a random negative example. AUC is **scale invariant** and **classification-threshold-invariant**, note that these two properties are not always desirable.

- **Average $\log$-loss:** Computed using loss $-\frac{1}{N}\sum_{i=1}^{N}\left(y_i \log p_i + (1-y_i)\log(1-p_i)\right)$. This loss is directly optimized in many models.

Regression metrics:

- **Root Mean Square Error (RMSE):** MSE and RMSE are some of the most commonly used losses in regression. They are easy to optimize directly. The best constant prediction is the average value.

- **Mean Absolute Error (MAE):** Another common loss. Since absolute value is used instead of the square, this loss is less sensitive to the outliers. The best constant prediction is the median value.

- **R Squared (R^2):** R^2 is a loss similar to MSE but gives a more human-interpretable result. It is computed using $1 - \frac{MSE(\text{model})}{MSE(\text{baseline})}$, and it tells how far away our model is from some perfect baseline.

1.3 Train/test/validation strategies

It is important to validate the proposed model before moving it to production. This is done using test and validation sets, but one must think carefully about how these sets should be generated. There are multiple possible ways:

- **Random-split:** Assumes that data is i.i.d. and is the traditional option.
- **Time-based split:** Obvious choice for time-series data - don't want model to learn the past.
- **ID-based split:** If we want to generalise to unseen data, we should not put the same IDs into both training and test sets. Sometimes we have to learn the IDs during the competition.

1.4 Data leak

A data leak happens when some of the information about the test set is somehow included in the training set. Although data leak exploiting only happens during competitions, in real world problems we need to make sure our training data does not contain leaks or the final model will not generalise well to real world data.

2 Optimization

For symmetric A, the function of quadratics is:

$$f(x) = \frac{1}{2}x^T Ax - x^T b$$

2.1 Gradient descent and first order methods

When minimizing some function $f(x)$ using iterative non-linear optimization, at every step we want to move in the direction that decreases the value of $f(x)$. One way to do it is to use the gradient of the function - we step into the negative direction to the gradient with learning rate ϵ:

$$x_{k-1} = x_k - \epsilon \nabla f(x)$$

Gradient descent is easy to implement, but it's not very good - it only **improves the value only by a small amount** at every step, and if ϵ is large the solution may not converge at all. This method is also coordinate system dependant (unless we're talking about orthogonal transformations).

TODO: *Generalised gradient update*

2.1.1 Momentum

The idea behind gradient momentum is to use some sort of running average of the gradient to to updates. Let $\tilde{g}_k$ denote such running average and let $g(x_k)$ denote the gradient of f at x_k, i.e. $g(x_k) = \nabla f(x_k)$. We can then do update as follows:

$$\begin{aligned} \hat{g}_{k+1} &= \mu_k \hat{g}_k - \epsilon g(x_k) \\ x_{k+1} &= x_k + \hat{g}_{k+1} \end{aligned}$$

Momentum is useful to speed up convergence - where normal gradient descent "slows down" as it approaches the optimum, momentum-based descent will move at a slightly faster rate. Momentum is also helpful to deal with noisy gradients and is less likely to oscillate during convergence. Momentum can also avoid saddles. The momentum coefficient μ may need to be decreased as the number of iteration increases to improve convergence.

2.1.2 Nesterov's accelerated gradient (NAG)

NAG is similar to momentum but "speeds up" the gradient by evaluating g at a new point:

$$\hat{g}_{k+1} = \mu_k \hat{g}_k - \epsilon g(x_k + \mu_k \hat{g}_k)$$

The recommended value for μ is $\mu = 1 - \frac{3}{(k+5)}$. NAG has convergence rate $f(x^*) - f^* \leq \frac{c}{k^2}$ for some c, which is much faster than $\frac{1}{k}$ of normal gradient descent.

2.1.3 Line search and conjugate gradients

A simple way to speed up the convergence of standard gradient descent is to use **line search** - at every iteration, we pick some direction p_k, and we do line search along that direction by minimizing $F(\lambda) = f(x_k + \lambda p_x)$. This can potentially improve the solution much more per-iteration than gradient descent.

How do we pick the direction? The obvious solution is to set $p = -\nabla f(x_k)$, but that can cause zig-zagging behaviour in the quadratic case. We can use quadratic functions to gain some intuition:

$$f(x) = \begin{bmatrix} x_1 & x_2 & ... & x_n \end{bmatrix} A \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{bmatrix} - b^T \begin{bmatrix} x_1 \\ x_2 \\ ... \\ x_n \end{bmatrix}$$

Note that when A is **diagonal**, each term x_i^2 appears independently of all another terms. This means we could minimize each x_i separately to find the global minimum. Because of this, we want to find some transformation $x = P\hat{x}$ such that $P^T AP$ is a diagonal matrix - the **conjugate gradient** method achieves exactly that.

Vector $p_i, i = 1...n$ are conjugate to matrix A if it holds that:

$$p_i^T A p_j = \delta_{ij}\, p_i^T A p_i \qquad i, j = 1...n$$

Where δ_{ij} is the Kronecker delta function. Searching along direction p_i effectively diagonalises the problem. If we express the solution as $x = \sum_i \alpha_i\, p_i$, we get:

$$f(x) \;=\; \frac{1}{2} x^T A x - b^T x \;=\; \sum_{i=1}^{n} \left(\frac{1}{2} \alpha_i^2\, p_i^T A p_i - \alpha_i\, b^T p_i \right)$$

The conjugate gradient algorithm works as follows. We perform optimization iteratively. At every iteration, we pick a direction p_k that is conjugate to all previous p_i's, and perform line search along that direction. We thus obtain the improved solution $x_{k+1} = x_k + \alpha_k p_k$.

After convergence, the final solution will then look like $x_K = \sum_i \alpha_i p_i$ (if we set $x_1 = 0$ for simplicity). Note that, for an n-dimensional problem, the maximum number of conjugate vectors is n. Thus a quadratic with $\dim x = n$ is guaranteed to converge in n iterations, where each iteration takes $O(n^2)$ steps. In a general non-quadratic scenario, no such guarantee exists.

How do we choose p_k's? Let $g_k = \nabla f(x_k)$. We then use:

$$\begin{aligned} \beta_k &= \frac{\langle g_{k+1}, g_{k+1} \rangle}{\langle g_k, g_k \rangle} \\ p_{k+1} &= -\, g_{k+1} + \beta_k p_k \end{aligned}$$

Which is guaranteed to give us conjugate gradients for our matrix A. Another alternative is Polak-Ribiere formula:

$$\beta_k = \frac{\langle g_{k+1}, g_{k+1} \rangle - \langle g_{k+1}, g_k \rangle}{\langle g_k, g_k \rangle}$$

Conjugate gradient algorithm is very efficient on **sparse linear systems**. This is because we only ever need to do matrix-vector product calculation in CG, these calculation are faster with

sparse matrices than dense ones. If matrix is dense, your best course of action is probably to factor and solve the equation by backsubstitution. The time spent factoring a dense matrix is roughly equivalent to the time spent solving the system iteratively.

Other Gradient Descent methods requires computation of inverse (Newton) or can overshoot (Naive GD).

2.2 Higher order methods

2.2.1 Newton's method

Newton's method can be derived from the Talyor series approximation of the function (up to the second term), then taking the derivative to obtain the update:

$$x_{k-1} = x_k - \epsilon H_f^{-1} \nabla f$$

Newton's method converges in one step for quadratic problems. It also results in the same decrease in objective function invariant of change in the objective function. As for disadvantages, storing the Hessian matrix and solving linear system $H_f^{-1}\nabla f$ is very expensive. It is also not guaranteed to produce a downhill step - it only goes downhill when ϵ is small and H is positive definite.

There exist several Quasi-Newton methods that iteratively form the approximation for the inverse of the Hessian, such as the Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithm. L-BFGS is a version of BFGS that uses much less memory.

TODO: *Finish optimization chapter*

3 Ensembles of trees

Trees are often used in machine learning. The pros are that they are human-interpretable, don't require much data preprocessing and work with both numerical and categorical features. The cons are that optimal trees as hard to construct since short trees are weak predictors and deep trees tend to overfit.

Random forests (RFs) are easy to parallelise. They also provide estimation of generalization error for free, which is called the Out-Of-Bag (OOB) error. For each example, you compute the prediction using only the trees that were built without this example. RFs are less interpretable than single trees. The hyperparameters of RFs are number of samples for each tree, stopping criteria, and the number of features sampled at each node. If we're clever, we can find the best split for each iteration in $O(DN \log N)$ time.

Gradient boosting (GB) achieve good results by combining a lot of weak learners. We start by predicting the mean of the data, $F_0(x) = \bar{y}$. At every step k, we compute residuals $r_i^k = y_i - F_k(x_i)$, and fit a new model h_k to these residuals. We then find parameter α_k to form a new predictor $F_k(x) = F_{k-1}(x) + \alpha_k h_k(x)$.

We can see the connection to gradient descent if we take the derivative of MSE loss for this problem. That derivative is proportional to the residuals, so at every step we are decreasing gradient by fitting to residuals. Moreover, we can find pseudo-residual for any differentiable loss function $L(y, \hat{y})$ using:

$$r^k = \frac{\partial}{\partial \hat{y}} L(y, \hat{y}) \Big|_{\hat{y}=F_{k-1}(X)}$$

4 Matrix factorization

Low-rank factorization of matrices has several useful applications. For one thing, it lets us compress data - we can store $k(m+n)$ instead of mn numbers. Additionally, if the matrix A is a noisy version of some low-rank matrix, the PCA reconstruction can actually be more informative.

Useful in word vectors, topic modelling, and non-negative matrix factorization (NNMF) in source separation problem.

TODO: *Talk about word vectors*

4.1 PCA via SVD

PCA:

- Finds linear subspace s.t. projection of data onto that subspace maximizes the variance and thus reduces the sum of squared distances from the data points to the projections (can be done by a greedy algorithm).
- Solution found by taking first k eigenvectors of $X^T X$, which is also a part of SVD: $X^T X = ULV^T$. SVD also produces matrix U, which is useful for collaborative filtering - U are canonical products and V are canonical customers.

SVD:

- Can be computed efficiently using the power method - pick a x_0 uniformly at random, send it through the matrix repeatedly until convergence. When convergence is achieved, x_0^* is the largest eigenvector. Repeat this to get all eigenvectors. Convergence is determined by ratio σ_1/σ_2.
- Can be computed in $O(\min(mn^2, m^2 n))$. Industry methods can be used to compute it but they are more involved.
- Singular values are unique, but a single value can appear multiple times (meaning that dimensionality of the span of corresponding eigenvectors is more than two).
- **Good:** SVD is optimal in a sense that it's the best k-rank approximation of the matrix.
- **Bad:** Singular vectors are orthogonal, which is bad for topic analysis: all topics will be "distinct".
- **Bad:** Some entries are negative, which is bad for probabilistic interpretability.

4.2 Non-negative matrix factorization (NNMF)

NNMF outline:

- NNMF is concerned with finding a tall matrix W and a flat matrix H such that all entries are positive. The optimization problem is (in terms of Frobenius norm):

$$\min \|X - WH\|_F^2, \quad W \geq 0,\ H \geq 0$$

- The problem is NP-hard and ill-posed, need priors and regularization.
- Rank of the factorization needs to be learnt by trial and error.

The usual approach is to fix H, solve for W. Then fix W, solve H, and so on. The motivation is that each sub-problem is convex, and, in fact, independent in rows of W - can show this by fixing H and writing out $\|X - WH\|_F^2$.

4.2.1 NNMF Approach #1: Majorizataion-minimization framework

This topic is not mentioned in the syllabus.

4.2.2 NNMF Approach #2: Alternating Least Squares (ALS)

Can simply solve unconstrained optimization problem, then project using: $W = max(0, (XH^T)/(HH^T))$. Usually does not converge but is good for initializing values before switching to another algorithm.

4.2.3 NNMF Approach #3: Alternating Non-Negative Least Squares (NNLS)

Class of methods were sub-problems are solved exactly:

$$W = \text{argmin}_{W \geq 0} \|X - WH\|_F$$

Lots of methods to solve NNLS - active-set methods, projected gradients, Quasi-Newton. Also enhanced version (HALS) that converges faster. Result depends on initialization of W and H - many ways to initialize, e.g. SVD. Usually try several different initializations and use the ones that give best results (trial and error). Stopping criteria involve monitoring the objective function, optimality conditions and difference between successive iterations.

NNMF problems are NP-hard in the worst case, so many approximations are used (e.g. converging to local minima). To make the problem tractable, we need the **Separability assumption** (a.k.a. Anchor Words assumption): **Suppose matrix M admits factorization $M = AB$. For each column j in A, there exists a row i that is zero everywhere except A_{ij}.**

We can simplify our problem:

1. WLOG assume $\sum_j M_{ij} = 1$ for all rows i (otherwise we normalize, factorize and renormalize).
2. WLOG assume $\sum_j B_{ij} = 1$ for all rows i.
3. But this implies $\sum_j A_{ij} = 1$! Rows of M are linear combinations of rows of B, with coefficients from rows of A. Since rows of M and B sum up to 1 this must be a convex combination.
4. This means we're looking for a matrix B such that all rows of M are in convex hull of rows of B.

Note that **separability assumption** implies that there is an identity matrix embedded inside A. Since AB form M, this implies that k rows of M embedded in B! We can use a greedy algorithm to find these k rows - find a row that is in the convex hull of all other rows, then remove it.

5 Clustering

Used in unsupervised learning and semi-supervised learning.
Given data without labels, find compact descriptions (clusters) of data.

- Data in the same cluster have high similarity
- Data not in the same cluster have low similarity
- Most common (dis)similarity measure is the squared distance between points

Algorithms used in clustering includes K-means, Gaussians Mixture (Mixture of Gaussians) and Spectral Clustering.

5.1 K-Means

Imagine we have $x^1...x^n$ data vectors (n customers). We will assign each datapoint to a cluster, $x^n \rightarrow c(n)$ where $c(n) \in 1, ..., K$ is the cluster index of datapoint number n, so that the square loss $\sum_{n=1}^{N}(x^n - m^{c(n)})^2$ is minimized. Where $m^{c(n)}$ is the mean or centroid of nth cluster.

1: Initialise the centres $\mathbf{m}_i$, $i = 1, \ldots, K$.
2: **while** not converged **do**
3: For each centre i, find all the $\mathbf{x}^n$ for which i is the nearest centre.
4: Call this set of points $\mathcal{N}_i$. Let N_i be the number of datapoints in set $\mathcal{N}_i$.
5: Update the centre by taking the mean of those datapoints assigned to this centre:

$$\mathbf{m}_i^{new} = \frac{1}{N_i} \sum_{n \in \mathcal{N}_i} \mathbf{x}^n$$

6: **end while**

Figure 1: K-means algorithm

As the number of clusters K increases, the mean distance to clusters will decrease. By plotting number of clusters against mean distance to nearest clusters we can determine the best number of clusters as the elbow point in which the distance stops decreasing significantly.

There are 2 ways to assign points to clusters:

- **Hard assignment** allowing only one cluster per point
- **Soft assigment** allowing probability of belonging to different clusters per point (done with mixture models)

K-means is fast but it has some downsides:

- **Initialization dependent** hence we have to run it several times with different initialization and then find the clusterings that achieve lowest squared loss.

- **Sensitive to outliers**, one way to avoid this is used K-medians algorithm which works the same but uses median instead.
- **Doesn't work well with missing data** since representing missing data with 0 throws off k-means.
- Doesn't work when clusters are **non-spherical** and **have non-equal density**. We need to use spectral clustering to overcome this.
- **High dimensional data** requires expensive computational cost. (alternative is fastNN)
- **Sensitive to data representation:** If one attribute is distance in mm and the other is temperature in Celsius, the data will be overwhelmingly clustered into distance. Need to rescale data appropriately.
- **Need data to be categorised appropriately.**

5.2 Hierarchical K-Means

This algorithm has 2 approaches:

- **Top-down (Divisive):** clustering all data into a small number of groups, and then clustering the data in each group.
- **Bottom-up (Agglomerative):** Starts with singleton clusters. At each time-step, greedily merge 2 most similar clusters.

5.3 Mixture models

Powerful technique in unsupervised learning, can deal with categorical data, missing data (where k-means fails). We define the model as:

$$p(v) = \sum_{h=1}^{H} p(v|h)p(h)$$

Where v is the visible state, h is the hidden state. Mixture models have a natural interpretation in terms of clustering with each state of h corresponding to a cluster model $p(v|h)$. Like in Machine Vision, we can find $p(v|h)$ using maximum likelihood or Bayesian approach.

To learn the params of a mixture of models, define:

$$L(\theta) = \Pi_{n=1}^{N} p(v^n|\theta)$$

We maximise L w.r.t. θ using E-M algorithm, remember to do E-M multiple times as it is also initialisation dependent like k-means. **The best θ corresponds to the highest likelihood value.**

Mixture of models is good because we can use any distribution to model $p(x_i|h)$

- Use Gaussian in general but t-distribution can be good to deal with outliers.
- Train using E-M algorithm.
- Hold-out technique is commonly used for finding the number of mixture components.

5.4 Spectral Clustering

TODO: *Rewrite this part?*

Algorithm:

1. Input: Similarity matrix S, number of clusters k
2. Construct a similarity graph by one of the methods we describe below.
3. Compute the Graph laplacian L.
4. Compute the first k eigenvectors $v_1 ... v_k$ of L.
5. Compute the vectors $y_i \in R$, such that j-th component of i-th vector is the i-th coordinate of v_j.
6. Cluster the points y_i using k-means algorithm.

Build an undirected graph with data points (x_i) as vertices (v_i) and their similarity (or distance) as edges (using k-NN or ϵ-ball). Define clustering objective as a certain partition problem for this graph.

Note: Choice of similarity measure is important, so is the choice of similarity graph since computations are easier on k-nn and ϵ-ball. Normalised clustering incorporates within-cluster similarity in the objective function.

Points are clustered using their connectivity. Similar to K-means, no assumption is made about the clusters. Graph building techniques:

- **ϵ-ball** (aka ϵ-NN): Connect all the points with pairwise distances less than ϵ. Unweighted edges, use unit edge.
- **k-NN**: Connect v_i and v_j if either x_i is in the k nearest neighbours of x_j or x_j is among k nearest neighbours of x_i.
- **Mutual k-NN**: Connect v_i and v_j if either x_i is in the k nearest neighbours of x_j and x_j is among k nearest neighbours of x_i. In both k-NN cases we weight the edges by the similarity of the data points $w_{ij} = s_{ij}$
- **Fully connected graph:** Connect all the points with positive similarity and weight all the edges by s_{ij}

We then build a Graph Laplacian $L = D - W$ where D is the degree matrix with value $d_1 ... d_n$ at the diagonal and 0 everywhere else. W is the weight adjacency matrix where each w_{ij} denotes the weight between v_i and v_j, $w_{ij} = 0$ for $i = j$. Properties of L:

- $d_{ii} = \sum_{j=1}^{n} w_{ij}$
- L is symmetric and positive semi-definite
- $f^T L f = \frac{1}{2} \sum_{i,j=1}^{n} (x_i - x_j)^2$ for vector $f \in R^n$

- Smallest eigenvalue is 0 with the corresponding eigenvector being a constant, in SL this eigen vector is 1.
- L has n non-negative eigenvalues and $\lambda_1(=0) \leq \lambda_2 \leq ... \leq \lambda_n$
- **Normalised L (aka random walk L):** $D^{-1}L = I - D^{-1}W$

Laplacian and connected components: The multiplicity k of eigenvalue 0 of L equals the number of connected components $A_1...A_k$ in the graph. The eigenspace of eigenvalue 0 is spanned by the indicator vectors of those components. **TODO:** *Proof?*

The graph can be represented by an adjacency matrix which has the similarity between each vertex as its elements. Define a symmetric matrix of similarity of datapoints: $A_{i,j} = e^{-\lambda||x^i - x^j||^2}$, with some fixed λ. Define a Markov chain transition matrix with elements: $p(i|j) = \frac{A_{i,j}}{\sum_i A_{i,j}}$

- **Can deal with cases where k-means fails:** Outliers, non-spherical clustering, non-equal cluster density etc...
- **Sensitive to choice of similarity measure**
- If k-means was used for clustering the equilibrium distributions, we have to run multiple times because of initialisation dependancy.
- λ **has to be set by hand.**

5.5 Graph Cut

For two disjoint subsets A and B, we define graph cut as $cut(A, B) = \sum_{i \in A, j \in B} w_{ij}$

Normally we would use MinCut but unfortunately it does not always give us balanced partitions, therefore we have to use RatioCut or NCut (same as RatioCut but the denominator is $volume(A_i)$).

5.6 RatioCut

RatioCut is graph cut based on the number of vertices: $RatioCut(A_1...A_k) = \sum_{i=1}^{k} \frac{cut(A_i, \bar{A}_i)}{|A_i|}$
Where $\bar{A}_i$ is the complement of A_i and $|A_i|$ is the number of vertices in the set.

5.7 RatioCut relaxation for $k = 2$

RatioCut is related to unnormalised spectral clustering when $k = 2$. Suppose we have some partitions $(A_i, \bar{A}_i)$, define f_i s.t.

- $f_i = \sqrt{\frac{|\bar{A}_i|}{|A_i|}}$ if $v_i \in A_i$
- $f_i = -\sqrt{\frac{|A_i|}{\bar{A}_i}}$ if $v_i \in \bar{A}_i$

One can show that $f^T L f = 2|V| RatioCut(A_i, \bar{A}_i)$. Also $\sum_i f_i = 0$ and $||f||^2 = n$. From here we can rewrite RatioCut as for any f:

$$min_f f^T L f, \text{ subject to } f \perp 1, ||f|| = \sqrt{n}$$

The solution is the second eigenvector of L, we need to transform the solution back to the indicator vector using k-means clustering (this is analogous to unnormalised spectral clustering).

The same relaxation for NCut is analogous to normalised spectral clustering.

5.8 Random Walk

Spectral clustering can be viewed as finding the partition s.t. the random walk with transition probabilities proportional to edge weights stay long within the same cluster.

Transition matrix $P = D^{-1}W$, $p_{ij} = \frac{w_{ij}}{d_i}$. We can see that $L_{rw} = I - P$ (rw = random walk). If the graph is unique and non-bipartite then there is a unique stationary distribution $\pi = (\pi_1 ... \pi_n)$ given by $\pi_i = \frac{d_i}{vol(G)}$

Let G be connected non-bipartite graph, if we run random walk from x_0 in the stationary distribution, one can show that NCut and Random Walk are equivalent: $NCut(A, \bar{A}) = P(A, \bar{A}) + P(\bar{A}, A)$

6 Visualization/t-SNE

http://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf

6.1 Vanilla SNE (Hinton Roweis, 2002)

Stochastic neighbour embedding tries to approximate the distances in a high-dimensional space with a low-dimensional space. We solve the following problem by gradient descent (Top: Cost function, bottom: Gradient):

$$C = \sum_i KL(P_i||Q_i) = \sum_i \sum_j p_{j|i} log \frac{p_{j|i}}{q_{j|i}}$$

$$\frac{\delta C}{\delta y_i} = 2 \sum_j (p_{j|i} - q_{j|i} + p_{i|j} - q_{i|j})(y_i - y_j)$$

Where Q is a low-dimensional space and P is a high-dimensional space. $p_{j|i}$ and $q_{j|i}$ are normalized distances between points, defined as normalized RBF kernels

$$p_{j|i} = \frac{exp(-||x_i - x_j||^2/2\sigma_i^2)}{\sum_{k \neq i} exp(-||x_i - x_k||^2/2\sigma_i^2)}$$

$$q_{j|i} = \frac{exp(-||x_i - x_j||^2)}{\sum_{k \neq i} exp(-||x_i - x_k||^2)}$$

We just define $\sigma = \frac{1}{\sqrt{2}}$ for $q_{j|i}$.

For σ, we fudge: In denser regions, smaller σ is preferred, in sparse regions, we prefer a larger σ. SNE performs binary search on σ to produce P_i with user-defined perplexity.

(Perplexity: $Perp(P_i) = 2^{H(P)}$, with Shannon Entropy $H(P_i) = -\sum_j p_{j|i} log_2 p_{j|i}$; Can be interpreted as a "smooth measure of the number of neighbours")

6.2 t-Distributed SNE

Vanilla SNE has two problems:

1. Loss is not symmetric, making gradients more complex, slower to evaluate.
2. Outliers in high-dimensional space will have constant gradient since their low-dimensional location does not matter, because it is so far away
3. "Crowding problem": Low-dimensional reduction forces 'close' vectors to be even closer together in low dimensions, crushing together points in the center of the map

To solve (1) and (2), we define the joint distribution $p_{ij} = \frac{p_{j|i}+p_{i|j}}{2n}$. This ensures that $\sum_j p_{ij} > \frac{1}{2n}$ for all datapoints, implying that all points contribute to the cost function. Writing out the joint in full is done by changing the indices in the denominator:

$$p_{ji} = \frac{exp(-||x_i - x_j||^2/2\sigma_i^2)}{\sum_{k \neq i} exp(-||x_k - x_l||^2/2\sigma_i^2)}$$

$$q_{ji} = \frac{exp(-||x_i - x_j||^2)}{\sum_{k \neq l} exp(-||x_k - x_l||^2)}$$

Additionally, we define the symmetric loss[1] over the *single* KL divergence

$$C = KL(P||Q) = \sum_i \sum_j p_{ij} log \frac{p_{ij}}{q_{ij}}$$

This gives this elegant expression for the gradient

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)$$

To solve (3), we replace the Gaussian with a heavy-tailed distribution for the low-dimensional representation. This allows moderate distances in high-dimensional datapoints to be mapped to a much larger distance in the low-dimensional map. We can do this because we actually have an expression for the joint probability between two datapoints in both representations, not just some distance metric.

t-SNE uses a student-t distribution with one degree of freedom (same as Cauchy distribution), giving us

$$q_{ij} = \frac{(1 + ||y_i - y_j||^2)^-1}{\sum_{k \neq l} (1 + ||y_k - y_l||^2)^-1}$$

One degree of freedom gives us the property that the resulting map will be invariant to scale, because when $(y_i - y_j)^2$ is large the '1' term becomes irrelevant (the term approaches the inverse square law for large pairwise distances). Also, Student-t is cheaper to compute than the Gaussian. New gradient is same as the old one with an extra term attached to the end like a polyp:

$$\frac{\delta C}{\delta y_i} = 4 \sum_j (p_{ij} - q_{ij})(y_i - y_j)(1 + ||y_i - y_j||^2)^-1$$

6.2.1 Advantages of t-SNE gradient over SNE

1. Slightly dissimilar points are not repelled as much as in SNE.

2. Very dissimilar points that are close together do not get infinite repulsion.

[1] symmetric because $p_{ij} = p_{ji}$; $q_{ij} = q_{ji}$

6.2.2 XxXOptimization_TrixXxX_360noscope

1. Force points to stay close together at the start to make it easier to move around clusters of data. Do this by adding L2 loss proportional to the sum of squared distances of the map points from the origin

2. Similarly, multiply p_{ij} by some factor early on to encourage tightly coupled clusters

6.2.3 Interpretation in terms of forces

Recall our original gradient:

$$\frac{\delta C}{\delta y_i} = 2\sum_j (p_{j|i} - q_{j|i} + p_{i|j} - q_{i|j})(y_i - y_j)$$

Physically, the gradient may be interpreted as the force created by a set of springs between the map point y_i and all other map points y_j. All springs exert a force along the direction (y_i - y_j). The spring between y_i and y_j repels or attracts the map points depending on whether the distance between the two in the map is too small or too large to represent the similarities between the two high-dimensional datapoints. The force of the spring is proportional to its length, and also proportional to its stiffness, which is the mismatch $(p_{j|i} - q_{j|i} + p_{i|j} - q_{i|j})$ between the pairwise similarities of the data points and the map points.

Crowding Problem: The spring connecting datapoint i to each of these too-distant map points will thus exert a very small attractive force. Although these attractive forces are very small, the very large number of such forces crushes together the points in the center of the map.

6.3 t-SNE Complexity

$O(N^2)$ for each objective function calculation (= each iteration)

6.4 Landmark Datapoints and Random Walk Tricks

Idea: Replace $p(j|i)$ by approximation $p(j'|i')$, as a one-off preprocessing operation: Randomly select 'landmark' points, call such a point index i'. Now, we randomly sample j' according to $p(j|i = i')$ until we get some $j' \neq i'$. Rinse and repeat for the same i' to get good coverage of j', then normalize to get $p(j'|i)$. Then repeat this for many i'.

6.5 Vantage point trees to approximate input similarities

Vantage point trees: Same as KD-Tree but using spherical partitions, i.e. a set of intersecting spheres.

Set $p_{j|i}$ to zero for all points that are not part of u neighbors of x_i. Can now compute input similarities in $O(uNlogN)$ time.

6.6 Gradient Approximations

Recall gradient [2]

$$\frac{\delta C}{\delta y_i} = 4 \sum_{j \neq i} \frac{p_{ij} - q_{ij}}{1 + (y_i - y_j)^2} = 4 \sum_{j \neq i} (p_{ij} - q_{ij}) q_{ij} Z(y_i - y_j)$$

We can split this into attracting and repulsive forces

$$\frac{\delta C}{\delta y_i} = 4 \sum_{j \neq i} \frac{p_{ij} - q_{ij}}{1 + (y_i - y_j)^2} = 4(F_{attr} + F_{rep}) = 4(\sum_{j \neq i} p_{ij} q_{ij} Z(y_i - y_j) - \sum_{j \neq i} q_{ij}^2 Z(y_i - y_j))$$

F_{attr} is cheap to compute: Sum nonzero elements in p in $O(uN)$ time. F_{rep} is still $O(N^2)$

6.7 Barnes-Hut Approximation

Complexity: $O(NlogN)$.

Remember: Gradient gives representation as attractive/repulsive forces.

In physics, we need to compute gravitational attraction between N bodies. This problem is called N-body problem and has the Barnes-Hut approximation. Idea: Replace points far away by their center of mass.

We do this by constructing a Quad/Octtree and storing the center of masses at each cell. Then, we traverse the Quadtree and decide whether to use the actual points or the approximation. Use parameter θ to tune this: The larger θ, the more we rely on the approximation.

[2] This is the gradient as mentioned in the Barber slides. For some reason, he adds some variables..

7 Fast NN

Finding neighbours quickly is useful for k-NN and k-mean clustering. Computing nearest neighbours using the naive approach takes $O(DN)$ steps, which is prohibitively expensive for large data sets. There are several efficient approaches to finding neighbours. Note that in the worst case, all of the methods presented below have complexity equal or worse than simply calculating all distances and finding the smallest one.

7.1 Exploting the triangle inequality

Note that any valid distance metric (norm) $d(x, y)$ satisfies the following inequality:

$$d(x, y) \leq d(x, z) + d(z, y)$$

If $d(x, y) \leq \frac{1}{2}d(z, y)$, then it follows $d(x, y) \leq d(x, z)$. Refer to the diagram below for some good intuition behind this idea.

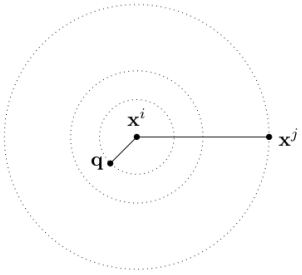

Figure 1: If we know that $\mathbf{x}^i$ is close to $\mathbf{q}$, but that $\mathbf{x}^i$ and $\mathbf{x}^j$ are not close, namely $d(\mathbf{q}, \mathbf{x}^i) \leq \frac{1}{2}d(\mathbf{x}^i, \mathbf{x}^j)$, then we can infer that $\mathbf{x}^j$ will not be closer to $\mathbf{q}$ than $\mathbf{x}^i$, *i.e.* $d(\mathbf{q}, \mathbf{x}^i) \leq d(\mathbf{q}, \mathbf{x}^j)$.

7.2 Orchard

Orchard method is useful when computing the distance between points is expensive. It uses triangle inequality to reduce the number of such calculations, but requires us to store a large inter-point distance matrix.

8 Fairness