Learning Notes on Computer Science and Artificial Intelligence

JIAN TONG
University of Edinburgh
October 17, 2018

Part I

Natural Language Processing

Chapter 1

Vector Semantics

1.1 Skip-gram

The training objective is to find word representations useful for predicting the **surrounding words** in a sentence or a document. Given a sequence of training words $w_1, w_2, w_3, ..., w_T$, the objective is to maximize the average log probability

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c}\log(w_{t+j}|w_t)$$

The basic Skip-gram formulation defines $p(w_{t+j}|w_t)$ using softmax function:

$$p(w_O|w_I) = \frac{\exp\left(\bar{v}_{w_O}^{\top} v_{w_I}\right)}{\sum_{w=1}^{W}\exp\left(\bar{v}_{w}^{\top} v_{w_I}\right)}$$

Here's the architecture of Skip-gram model, it should be noticed that the model is a **two-layer** network(hidden units and output units),which leads to one of its drawbacks — $p(w_{j+t}|w_t)$ is **asymmetric**, that is, $p(w_2|w_1)$ is not necessarily equal to $p(w_1|w_2)$. We need to separate the vector space between target words and context words. (**hidden units** and **output units**)

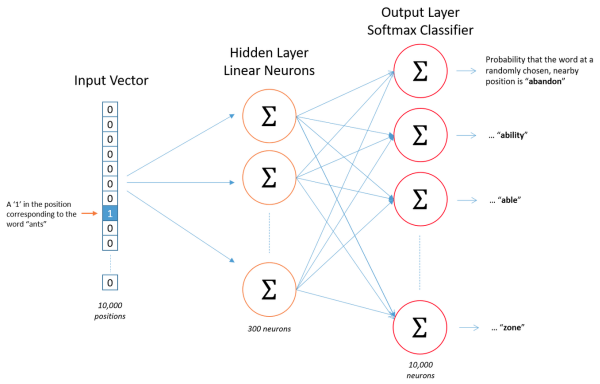

Figure 1.1: Neural network architecture of Skip-gram model

1.2 GloVe

GloVe or **Global Vector**, directly captures the global corpus statistics. The model attempts to solve two respective problems in word vector learning algorithms:

- **global matrix factorization**: do relatively poorly on the word analogy task, indicating a sub-optimal vector space structure.
- **local context window**: poorly utilize the statistics of the corpus since they train on separate local context windows instead of on global co-occurrence counts.

Let word-word co-occurrence counts be denoted by X_{ij}, denotes the number of times word j occurs in the context of word i. The **ratios of co-occurrence probabilities** rather than the **probabilities** themselves provide useful information.

Probability and Ratio	$k = solid$	$k = gas$	$k = water$	$k = fashion$
$P(k\|ice)$	1.9×10^{-4}	6.6×10^{-5}	3.0×10^{-3}	1.7×10^{-5}
$P(k\|steam)$	2.2×10^{-5}	7.8×10^{-4}	2.2×10^{-3}	1.8×10^{-5}
$P(k\|ice)/P(k\|steam)$	8.9	8.5×10^{-2}	1.36	0.96

Figure 1.2: ratio of co-occurrence probabilities

In the above example, only in ratio does noise from non-discriminative words like *water* and *fashion* cancel out, so the large values (much greater than 1) correlate well with properties specific to ice, and small values (much less than 1) correlate well with properties specific of steam.

1.2.1 Model Interpretation

The most general model takes the form, where $w \in \mathbb{R}^d$ are **word vectors** and $\tilde{w} \in \mathbb{R}^d$ are **separate context vectors**. Let $P_{ij} = P(j|i) = X_{i,j}/X_i$ be the probability that **word** j **appear in the context of word** i.

$$F(w_i, w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

Since vector spaces are inherently linear structures, the most natural way to do this is vector differences

$$F(w_i - w_j, \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

In order to make the left-hand side as a scalar

$$F((w_i - w_j)^\top \tilde{w}_k) = \frac{P_{ik}}{P_{jk}}$$

To remove the distinction between context vectors and word vectors, symmetry need to be restored. F is required to be homomorphism between the groups (ℝ, +) and (ℝ, $\times$), i.e.,

$$F((w_i - w_j)^\top \tilde{w}_k) = \frac{F(w_i^\top \tilde{w}_k)}{F(w_j^\top \tilde{w}_k)}$$

$$F(w_i^\top \tilde{w}_k) = P_{ik} = \frac{X_{ik}}{X_i}$$

The solution is $F = \exp$, or,

$$w_i^\top \tilde{w}_k = \log(P_{ik}) = \log(X_{ik}) - \log(X_i)$$

Noted that it would exhibit the exchange symmetry if not for the $\log(X_i)$ on the right-hand side. However, this term is independent of k so it can be absorbed into a bias b_i for w_i. Finally, adding an additional bias $\tilde{b}_k$ for $\tilde{w}_k$ restores the symmetry

$$w_i^\top \tilde{w}_k + b_i + \tilde{b}_k = \log(X_{ik})$$

Two tricks are introduced here as follows:

- **logarithm divergence**

- **weigh all co-occurrences equally**

One resolution is to include an additive shift in the logarithm, $\log(X_{ik}) \rightarrow \log(1 + X_{ik})$, which maintains the sparsity of X while avoiding divergence. But GloVe introduces a new weighting function $f(X_{ij})$ into the cost function to solve these two problems simultaneously

$$J = \sum_{i,j=1}^{V} f(X_{ij})(w_i^\top \tilde{w}_k + b_i + \tilde{b}_k - \log X_{ij})^2$$

$f(x) = (x/x_{max})^\alpha$ if $x \leq x_{max}$
$f(x) = 1$ otherwise

which has the following properties

1. $f(0) = 0$.
2. $f(x)$ should be non-decreasing so that rare co-occurrences are not overweighted.
3. $f(x)$ should be relatively small for large values of x, so that frequent co-occurrences are not overweighted.

1.2.2 Time Complexity

we conclude that the complexity of the model is much better than the **worst case** $O(V^2)$,and in fact it does somewhat better than the on-line window-based methods which scale like $O(|C|)$.

1.3 FastText

In many **morphologically rich** languages like Finnish, French and Spanish, it is possible to improve vector representations by using character level information.

Here's the model architecture of FastText, where the only difference between *fastText* and CBOW is the subword model. Given a word *where* and a character n-gram length equals to 3. We first add special boundary symbols $<$ and $>$ at the beginning and end of words, then taking all its character ngrams — where $\rightarrow$ <wh, whe, her, ere, re>, <where>.

The scoring function is then $s(w, c) = \sum_{g \in G_w} z_g^\top v_c$

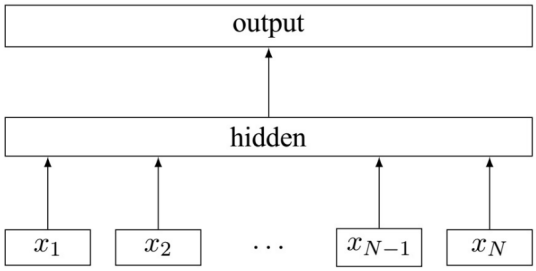

Figure 1.3: Model architecture of *fastText* for a sentence with N gram features $x_1, ..., x_N$. The features are embedded and average to form the hidden variable.

1.4 ELMo

ELMo or **Embeddings from Language Models** attempts to resolve two challenges:

1. complex characteristics of word use (e.g.,syntax and semantics)
2. how these uses vary across linguistic contexts

1.4.1 Bidirectional language models

Given a sequence of N tokens, $(t_1, t_2, ..., t_N)$, a forward language modeling the probability of t_k

$$p(t_1, t_2, ..., t_N) = \prod_{k=1}^{N} p(t_k | t_1, t_2, ..., t_{k-1})$$

whilst a backward model as

$$p(t_1, t_2, ..., t_N) = \prod_{k=1}^{N} p(t_k | t_{k+1}, t_{k+2}, ..., t_N)$$

A biLM combines both LMs and jointly maximizes the log likelihood as

$$\sum_{k=1}^{N} (\log p(t_k | t_1, t_2, ..., t_{k-1}; \Theta_x, \Theta_{LSTM}, \Theta_s) + p(t_k | t_{k+1}, t_{k+2}, ..., t_N; \Theta_x, \Theta_{LSTM}, \Theta_s))$$

1.4.2 ELMo

For each token t_k, a L-layer biLM computes a set of $2L + 1$ representations

$$R_k = \{x_K^{LM}, \overrightarrow{h}_{k,j}^{LM}, \overleftarrow{h}_{k,j}^{LM} | j = 1, ..., L\}$$

ELMo collapses all layers in R into a single vector, $ENMo_k = E(R_k; \Theta_e)$. In general, a task specific weighting of all biLM layers:

$$ELMo_k^{task} = E(R_k; \Theta^{task}) = \gamma^{task} \sum_{j=0}^{L} s_j^{task} h_{k,j}^{LM}$$

where s^{task} are softmax-normalized weights and scalar γ^{task} allows the task model to scale the entire ELMo vector.

Chapter 2

Chinese Word Segmentation

2.1 Dictionary-Based Method

Chapter 3

Neural Algorithms For Name Entity Recognition

3.1 LSTM-CRF Model

A hybrid tagging architecture combined LSTMs and CRFs is presented here. The LSTMs part always adopts a bi-directional LSTM module, which take as input a sequence of vectors $(x_1, x_2, ...x_n)$ and return another sequence $(h_1, h_2, ..., h_n)$. The representation of a word using this model is obtained by concatenating its left and right context representations, $h_t = [\overrightarrow{h_t}; \overleftarrow{h_t}]$.

3.1.1 CRF Tagging Model

Why do we need a CRF Tagging Module?

Considered a very simple — but surprisingly effective — tagging model is to use the h_t's as features to make **independent** tagging decisions for each output y_t. Despite the model's success in simple problems like **POS tagging**, its independent classification decisions are limited when there are **strong dependencies across output labels**. **NER** is one such task, since the "grammar" that characterizes interpretable sequences of tags imposes several hard constraints (e.g., I-PER cannot follow B-LOC) that **impossible to model with independence assumptions**.

Model detail

Instead of modeling tagging decisions independently, we model h_t jointly using a conditional random field. For an input sentence

$$X = (x_1, x_2, ..., x_n)$$

we consider P to be matrix of scores output by the bi-directional LSTM network. P is of size $n \times k$, where k is the number of distinct tags, and $P_{i,j}$ corresponds to the score of the j^{th} tag of the i^{th} word in a sentence. For a sequence of predictions

$$y = (y_1, y_2, ..., y_n)$$

we define its score to be

$$s(X, y) = \sum_{i=0}^{n} A_{y_i, y_{i+1}} + \sum_{i=1}^{n} P_{i, y_i} \tag{3.1}$$

where A is a matrix of transition scores such that $A_{i,j}$ represents the score of a transition from the tag i to tag j. (y_0 and y_n are the *start* and *end* tags of a sentence, that we add to the set of possible tags. A is therefore a square matrix of size $k + 2$)

Note: this part is actually very similar to the Hidden Markov Model(HMM), where P is the *emission matrix* and A is the *transmission matrix*, except that the scores are **unnormalized** which can not be formed as probabilities.

A softmax over all possible tag sequences yield a probability for the sequence y:

$$p(y|X) = \frac{e^{s(X,y)}}{\sum_{\tilde{y} \in Y_x} e^{s(X,\tilde{y})}} \tag{3.2}$$

During **training**, we maximize the log-probability of the correct tag sequence:

$$\begin{aligned} \log\left(p(y|X)\right) &= s(X, y) - \log\left(\sum_{\tilde{y} \in Y_x} e^{s(X,\tilde{y})}\right) \\ &= s(X, y) - \text{logadd}_{\tilde{y} \in Y_x} s(X, \tilde{y}) \end{aligned} \tag{3.3}$$

While **decoding**, we predict the output sequence that obtains the maximum score given by:

$$y^* = \text{argmax}_{\tilde{y} \in Y_x} s(X, \tilde{y}) \tag{3.4}$$

Both the above the summation in training and maximum a posteriori sequence y^* can be computed by using **dynamic programming**.

Parameterization and Training

The score associated with each tagging decision for each token ($P_{i,y}$) are defined to be the dot product between

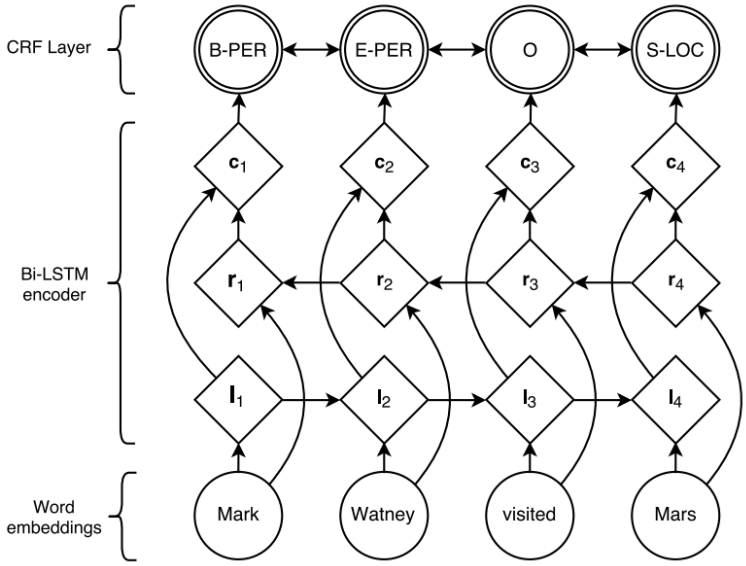

Figure 3.1: Main architecture of the LSTM-CRF network

- **embedding of a word-in-context**: computed with a bi-directional LSTM, that is, concatenating the forward and backward LSTM outputs and projecting onto a layer whose **size is equal to the number of distinct tags** (instead of using the softmax output from the above mentioned layer).
- **bigram compatibility scores** $A_{y,y'}$: computed in the CRF layer.

Note: Adding a hidden layer between c_i and CRF layer marginally improved the results.

3.2 LSTM-CNN-CRF Model

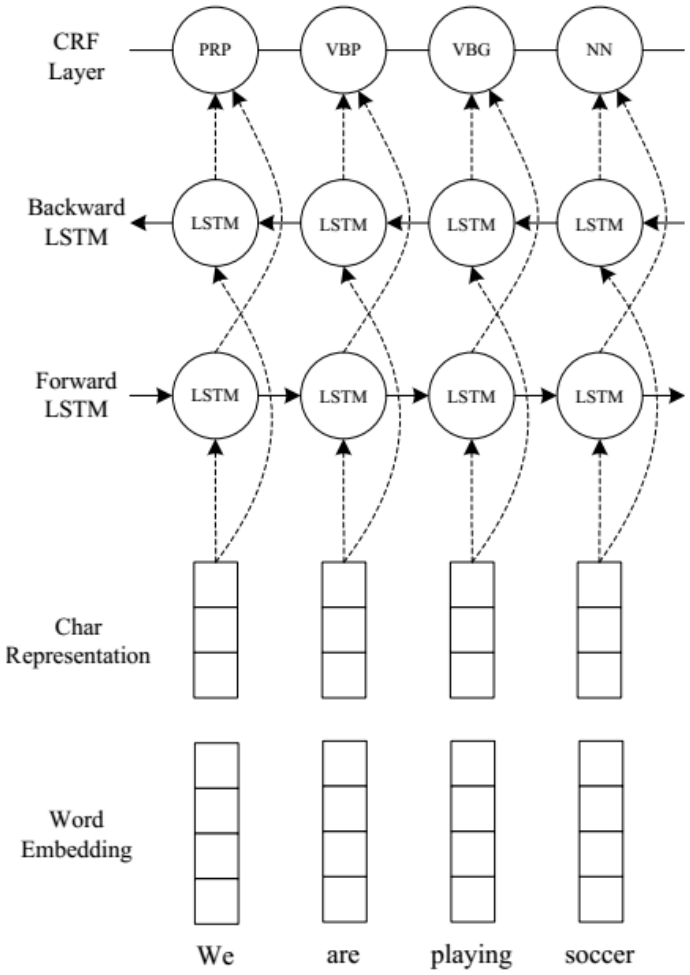

Figure 3.2: Main architecture of the LSTM-CNN-CRF network

Chapter 4

Topic Model

In *machine learning* and *natural language processing*, a **topic model** is a type of statistic model of discovering the abstract "topics" that occur in a collection of documents.

4.1 Probabilistic Latent Semantic Analysis

4.1.1 A Modern View of pLSA

In order to better understand the intuition behind the model, we need to make some assumptions. First, we assume a topic ϕ_k is a distribution over a fixed size of vocabulary V. In the original pLSA model, this distribution is not explicitly specified but the form is Multinomial distribution. Thus, ϕ_k is essentially a vector that each element $\phi_{(k,w)}$ represents the probability that term w is chosen by topic k, namely:

$$p(w|k) = \phi_{(k,w)}$$

and note $\sum_w^V \phi_{(k,w)} = 1$. Secondly, we also assume that a document consists of multiple topics. Therefore, there is a distribution θ_d over a fixed number of topics T for each document d. Similarly, original pLSA model does not have the explicit specification of this distribution but it is indeed a Multinomial distribution where each element $\theta_{(d,k)}$ in the vector θ_d represents the probability that topic k appears in document d, namely

$$p(k|d) = \theta_{(d,k)}$$

and also $\sum_k^T \theta_{(d,k)} = 1$. This is the prerequisite of the model.

pLSA can be considered as a **generative** model, where the generation process can be summarized as follows:

For each document d

- For each token position i
  - Choose a topic $z \sim \text{Multinomial}(\theta_d)$
  - Choose a term $w \sim \text{Multinomial}(\phi_z)$

and we can write the **probability a term** w appearing at position i in the document d as follows:

$$p(d_i = w|\Phi, \theta_d) = \sum_{z=k}^{T} \phi_{(z,w)}\theta_{(d,z)} \tag{4.1}$$

and the joint probability of the whole dataset **W** is:

$$\begin{aligned} p(\mathbf{W}|\Phi, \Theta) &= \prod_{d}^{D}\prod_{i}^{N_d}\sum_{z=k}^{T}\phi_{(z,w)}\theta_{(d,z)} \\ &= \prod_{d}^{D}\prod_{w}^{V}(\sum_{z=k}^{T}\phi_{(z,w)}\theta_{(d,z)})^{n(d,w)} \end{aligned} \tag{4.2}$$

where $n(d, w)$ is the number of times term w appearing in document d.

4.2 Latent Dirichlet Allocation

4.2.1 The Dirichlet and its Relation to the Multinomial

The Dirichlet distribution with parameter vector α of length K is defined as

$$Dirichlet(\theta; \alpha) = \frac{1}{B(\alpha)}\prod_{i=1}^{K}\theta_i^{\alpha_i - 1} \tag{4.3}$$

where $B(\alpha)$ is the multivariate Beta function, which can be expressed using gamma function as

$$B(\alpha) = \frac{\prod_{i=1}^{K}\Gamma(\alpha_i)}{\Gamma(\sum_{i=1}^{K}\alpha_i)} \tag{4.4}$$

Chapter 5

Knowledge Graph

5.1 TransE

5.2 TransD

5.3 TransR

5.4 TransH

Chapter 6

Question Answering

6.1 Summary

Most question answering systems focus on **factoid questions**, two basic approaches towards this problem:

- **IR-based question answering**
  - information retrieval techniques find relevant documents
  - **reading comprehension algorithms** to read these retrieved documents or passages
  - draw an answer directly from **spans of text**
- **Knowledge-based question answering**
  - build a semantic representation of the query to the logical representation
  - use meaning representation to query databases of facts

6.2 IR-based Question Answering

6.2.1 Question Processing

Extract the **query**: the keyword to match potential documents, along with additional **features**:

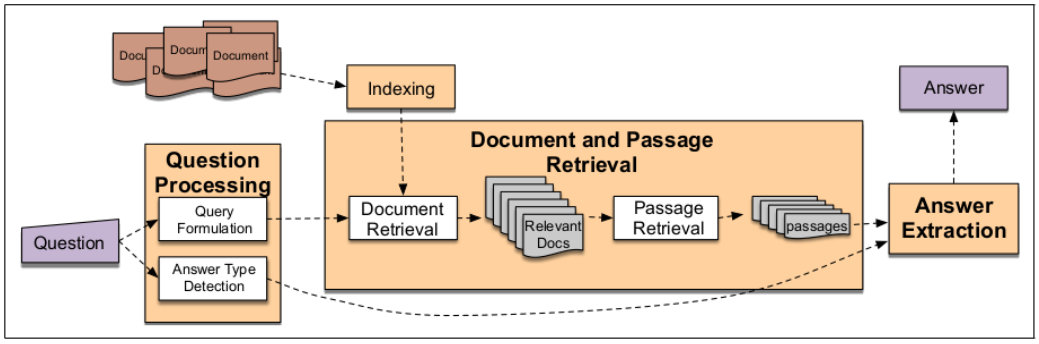

Figure 6.1: IR-based factoid question answering has three stages: question processing, passage retrieval, and answer processing.

- **answer type**: the entity type (person, location, time, etc.) of the answer.
- **focus**: the string of words in the question that are likely to be replaced by the answer in any answer string found.
- **question type**: is this a definition question, a math question, a list question?

Example:

- question: *Which US state capital has the largest population?*
- query: *US state capital has the largest population* item answer type: *city*
- focus: *state capital*

6.2.2 Query Formulation

Query formulation is a task of creating a query – a list of tokens – to send to information retrieval system. For QA from smaller sets of documents, **tf-idf cosine matching** and **query expansion** may be helpful.
Query reformulation rephrases the question to make it look like a substring of possible declarative answers. *"when was the laser invented?"* $\rightarrow$*"the laser was invented"*

6.2.3 Answer Types

Some systems make use of **question classification** to find the answer type. Each question can be labeled with a tag.

- **rule-based**:
- **supervised learning**:

6.2.4 Document and Passage Retrieval

1. IR engine ranks the documents by their relevance to the query.
2. QA systems divide the top n documents into smaller passages such as sections, paragraphs, or sentences.
3. Filter the passages by running answer type or name entity classification.
4. Supervised learning algorithms fully rank the rest passages with features like:

- The number of **named entities** of the right type in the passage
- The number of **question keywords** in the passage
- The longest exact sequence of **question keywords** that occurs in the passage
- The rank of the document from which the passage was extracted
- The **proximity** of the keywords from the original query to each other
- The number of **n-grams** that **overlap** between the passage and the question

6.2.5 Answer Extraction

In this step, a specific answer is extracted from the passage. This task is commonly modeled by **span labeling**: given a passage, identifying the **span** of the text which constitutes an answer.

Part II

Deep Learning

Chapter 7

Recurrent Neural Network

Chapter 8

Improving Neural Networks

8.1 Batch Normalization

8.1.1 Internal Covariate Shift

Internal covariate shift reduces the network learning efficiency since the *the change in the distributions of layers need to continuously adapt to the new distribution*.

Covriate shift extended to sub-network

Consider a network computing

$$l = F_2(F_1(u, \Theta_1), \Theta_2)$$

where F_1 and F_2 are arbitrary transformations, and the parameters θ_1, $\theta 2$ are to be learned so as to minimize the loss l. where F_1 and F_2 are arbitrary transformations, and the parameters Θ_1, Θ_2 are to be learned so as to minimize the loss l. Learning Θ_2 can be viewed as if the inputs $x = F_1(u, \Theta_1)$ are fed into the sub-network. $l = F_2(x, \Theta_2)$, thus a gradient step is exactly equivalent to that for a stand-alone network F_2 with input x.

As it is advantageous for the distribution of x to remain fixed over time, then Θ_2 does not have to readjust to compensate for the change in the distribution of x.

Help training process

Consider a layer with a sigmoid activation function, as $|x|$ increases, $g^{'}(x)$ tends to zero. In practice, the saturation problem and the resulting vanish-

ing gradients are amplified as the network depth increases. If we could ensure that **the distribution of nonlinearity inputs remains more stable as the network trains, then the optimizer would be less likely to get stuck in the saturated regime, and the training would accelerate**.

8.1.2 Towards Reducing Internal Covariate Shift

Let x be a layer input, treated as a vector, and χ be the set of these inputs over the training data set. The normalization can then be written as a transformation

$$\hat{x} = \text{Norm}(x, \chi)$$

which depends not only on the given **training example** x, but also on **all example** χ. For backpropagation, we would need to compute the Jacobians

$$\frac{\partial \text{Norm}(x, \chi)}{\partial x}, \frac{\partial \text{Norm}(x, \chi)}{\partial \chi} \tag{8.1}$$

Why not ignore the latter part?

Ignoring the latter term would lead to the explosion. For example, consider a layer with the input u that adds the learned bias b, and normalizes the result by subtracting the mean of the activation computed over the training data: $\hat{x} = x - \mathbb{E}[x]$ where $x = u + b$, $\chi = \{x_1, ..., x_N\}$ and $\mathbb{E}[x] = \sum_{i=1}^{N} x_i$.

If a gradient descent ignores the dependence of $\mathbb{E}[x]$ on b:

$$\begin{aligned} b &\leftarrow b + \Delta b \\ b &\propto -\partial l / \partial \hat{x} \end{aligned} \tag{8.2}$$

Then

$$u + (b + \Delta b) - \mathbb{E}(u + (b + \Delta b)) = u + b - \mathbb{E}[u + b]$$

The combination of the update to b and subsequent change in normalization led to no change in the output of the layer nor the loss. As the training continues, **b will grow indefinitely** while the **loss remains fixed**. This problem can get worse if the normalization not only centers but also scales the activations.

Simplification

The full whitening of each layer's inputs is costly (requires computing the covariance matrix $\text{Cov}[X] = \mathbb{E}_{x \in \chi} X X^\top - \mathbb{E}[X]\mathbb{E}[X]^\top$ and its inverse

square root, to produce the whitened activation) and not everywhere differentiable.

1. **normalize each scalar feature independently** instead of whitening the features in layer inputs and outputs jointly, by making it have the mean of zero and variance of 1. For a layer with d-dimensional input $x = (x^{(1)}...x^{(n)})$, we will normalize each dimension

$$\hat{x}^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{\text{Var}[x^{(k)}]}}$$

   where the expectation and variance are computed over the training data set.

2. In the batch setting where each training step use **mini-batches** in stochastic gradient training instead of the whole training set.

Preserve the network capacity

Noted that simply normalizing each input of a layer may change what the layer can represent. In order to maintain the **network capacity**, we make sure that *the transformation inserted in the network can represent the identity transform*.

To accomplish this, we introduce, for each activation $x^{(k)}$, a pair of parameters $\gamma^{(k)}$, $\beta^{(k)}$, which scale and shift the normalized value:

$$y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$$

These parameters are learned along with the original model parameters, and restore the representation power of the network. Indeed, by setting $\gamma^{(k)} = \sqrt{\text{Var}[x^{(k)}]}$ and $\beta^{(k)} = \mathbb{E}[x^{(k)}]$, we could recover the original activations, if that were the optimal thing to do.

8.1.3 Training Algorithm

Each normalized activation $^{(\hat{k})}$ can be viewed as an input to a sub-network composed of the linear transform $y^{(k)} = \gamma^{(k)}\hat{x}^{(k)} + \beta^{(k)}$, followed by the other processing done by the original network.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure 8.1: Batch Normalizing Transform, applied to activation x over a mini-batch.

8.1.4 Inference

The normalization of activations that depends on the mini-batch allows efficient training, but is neither necessary nor desirable during inference.

Once the network has been trained, we use the normalization

$$\hat{x} = \frac{x - \mathbb{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}}$$

using the population, rather than mini-batch, statistics. Neglecting ϵ, these normalized activations have the same mean 0 and variance 1 as during training. We use the unbiased variance estimate $\mathrm{Var}[x] = \frac{m}{m-1} \times E_\beta[\sigma^2_\beta]$, where the expectation is over training mini-batches of size m and σ^2_β are their sample variances.

Using moving averages instead, we can track the accuracy of a model as it trains. Since the means and variances are fixed during inference, the normalization is simply a linear transform applied to each activation. It may further be composed with the scaling by γ and shift by β, to yield a single linear transform that replaces $\mathrm{BN}(x)$.

Input: Network N with trainable parameters Θ;
subset of activations $\{x^{(k)}\}_{k=1}^{K}$

Output: Batch-normalized network for inference, $N_{\text{BN}}^{\text{inf}}$

1: $N_{\text{BN}}^{\text{tr}} \leftarrow N$ // Training BN network
2: **for** $k = 1 \ldots K$ **do**
3: Add transformation $y^{(k)} = \text{BN}_{\gamma^{(k)},\beta^{(k)}}(x^{(k)})$ to $N_{\text{BN}}^{\text{tr}}$ (Alg. 1)
4: Modify each layer in $N_{\text{BN}}^{\text{tr}}$ with input $x^{(k)}$ to take $y^{(k)}$ instead
5: **end for**
6: Train $N_{\text{BN}}^{\text{tr}}$ to optimize the parameters $\Theta \cup \{\gamma^{(k)}, \beta^{(k)}\}_{k=1}^{K}$
7: $N_{\text{BN}}^{\text{inf}} \leftarrow N_{\text{BN}}^{\text{tr}}$ // Inference BN network with frozen // parameters
8: **for** $k = 1 \ldots K$ **do**
9: // For clarity, $x \equiv x^{(k)}, \gamma \equiv \gamma^{(k)}, \mu_{\mathcal{B}} \equiv \mu_{\mathcal{B}}^{(k)}$, etc.
10: Process multiple training mini-batches $\mathcal{B}$, each of size m, and average over them:

$$\text{E}[x] \leftarrow \text{E}_{\mathcal{B}}[\mu_{\mathcal{B}}]$$
$$\text{Var}[x] \leftarrow \frac{m}{m-1}\text{E}_{\mathcal{B}}[\sigma_{\mathcal{B}}^2]$$

11: In $N_{\text{BN}}^{\text{inf}}$, replace the transform $y = \text{BN}_{\gamma,\beta}(x)$ with $y = \frac{\gamma}{\sqrt{\text{Var}[x]+\epsilon}} \cdot x + \left(\beta - \frac{\gamma\,\text{E}[x]}{\sqrt{\text{Var}[x]+\epsilon}}\right)$
12: **end for**

Figure 8.2: Batch normalization inference

8.2 Residual Network

Denoting the desired underlying mapping as $\mathcal{H}x$, we let the stacked nonlinear layers fit another mapping of $\mathcal{F}(x) = \mathcal{H}(x) - x$. The original mapping is recast into $\mathcal{F}(x) + x$. We hypothesize that it is easier to optimize the residual mapping than to optimize the original, unreferenced mapping. To the extreme, if an identity mapping were optimal, *it would be easier to push the residual to zero than to fit an identity mapping by a stack of nonlinear layers*.

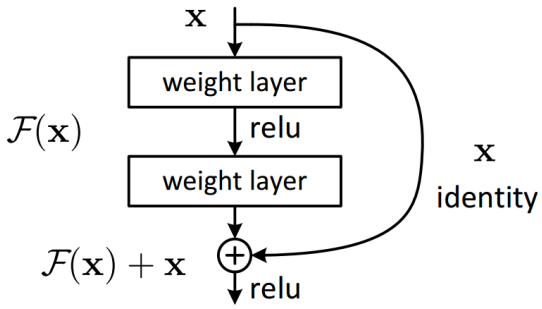

Figure 8.3: Residual learning: a building block.

8.2.1 Residual Learning

Chapter 9

Unsupervised Learning

9.1 Generative Adversarial Nets

9.1.1 Adversarial Nets

- **Generator**: $p_z(z) \rightarrow G(z;\theta_g) \rightarrow p_g$
  - $p_z(z)$: prior on input noise variables
  - $G(z;\theta_g)$: a differentiable function represented by a multilayer perceptron with parameters θ_g
  - p_g: generator's distribution
- **Discriminator**: $p_g \rightarrow D(x;\theta_d)$
- **Objective**: D and G play the following two-player minimax game with value function $V(G,D)$

  $$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

  s

9.1.2 Algorithm

for number of training iterations **do**
  for k steps **do**

- Sample minibatch of m noise samples $\{z^{(1)}, ..., z^{(m)}\}$ from noise prior $p_g(z)$.

- Sample minibatch of m examples $\{x^{(1)}, ..., x^{(m)}\}$ from data generating distribution $p_{data}(x)$
- Update the discriminator by **ascending** its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} [\log D(x^{(i)}) + \log (1 - D(G(z^{(i)})))]$$

end for

- Sample minibatch of m noise samples $\{z^{(1)}, ..., z^{(m)}\}$ xfrom noise prior $p_g(z)$.
- Update the generator by **descending** its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log (1 - D(G(z^{(i)})))$$

end for

9.1.3 Wasserstein GAN

9.2 Variational Auto-Encoder

VAE, similar with **GAN**, is an *unsupervised learning* method that aims at building a mapping between latent variable z and target data distribution x.

9.2.1 Problem scenario

Let us consider some dataset $X = \{x^{(i)}\}_{i=1}^{N}$ consisting of N i.i.d samples of some continuous or discrete variables x. We assume that the data are generated by some random process, involving an unobserved continuous random variable z.

1. a value $z^{(i)}$ is generated from some prior distribution $p_{\theta^*}(z)$
2. a value $x^{(i)}$ is generated from some conditional distribution $p_{\theta^*}(x|z)$

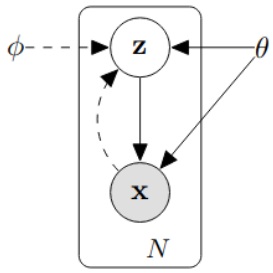

Figure 9.1: Directed graphical model under consideration for VAE

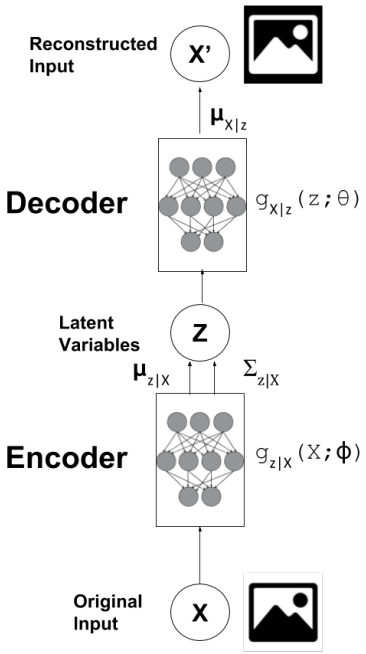

Figure 9.2: Vanilla Variational Autoencoder

Part III

Machine Learning

Chapter 10

Probability Distribution

Definition: *conjugate priors* —leads to posterior distributions having the same functional form as the prior.

10.1 Binary Variables

10.1.1 Bernoulli distribution

Considered a single binary random variable $x \in \{0, 1\}$. For example, flipping a coin, with $x = 1$ representing "heads", and $x = 0$ representing "tails". The probability distribution over x can therefore be written in the form

$$\text{Bern}(x|\mu) = \mu^x(1-\mu)^{1-x} \tag{10.1}$$

- $\mathbb{E}[x] = \mu$
- $\text{var}[x] = \mu(1-\mu)$

10.1.2 Binomial distribution

We can also work out the distribution of the number m of observations of $x = 1$, given that the data set has size N. This is called the *binomial distribution*

$$\text{Bin}(m|N,\mu) = \binom{N}{m}\mu^m(1-\mu)^{N-m} \tag{10.2}$$

- $\mathbb{E}[m] = N\mu$
- $\text{var}[m] = N\mu(1-\mu)$

10.1.3 Beta distribution

The **beta distribution** is introduced as a prior distribution $p(\mu)$ over the parameter μ for **binominal distribution** which keeps the *conjugacy* property.

$$\text{Beta}(\mu|a,b) = \frac{\Gamma(a+b)}{\Gamma(a)\Gamma(b)}\mu^{a-1}(1-\mu)^{b-1} \tag{10.3}$$

- $\mathbb{E}[\mu] = \frac{a}{a+b}$
- $\text{var}[\mu] = \frac{ab}{(a+b)^2(a+b+1)}$

The posterior distribution of μ is now obtained by multiplying the beta prior by the binomial likelihood function and normalizing, which has the form $p(\mu|m,l,a,b) \propto \mu^{m+a-1}(1-\mu)^{l+b-1}$

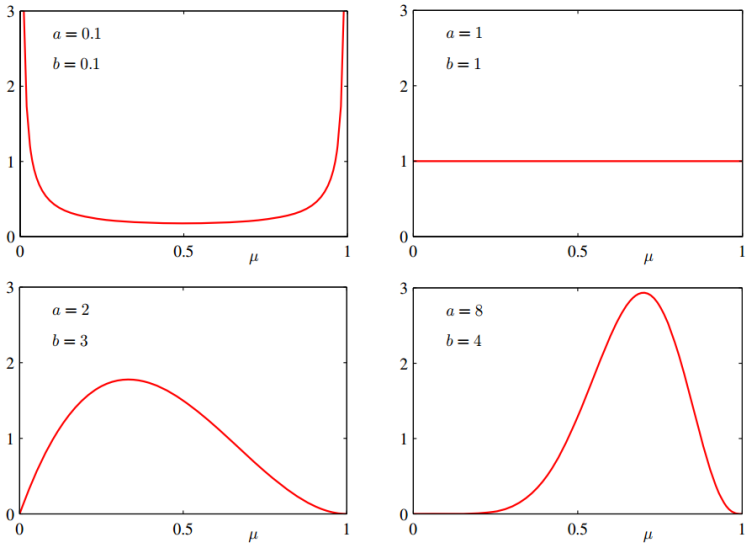

Figure 10.1: Plots of the beta distribution $\text{Beta}(\mu|a,b)$ as a function of μ for various values of the hyperparameters a and b.

10.2 Multinomial Variables

We encounter discrete variables that can take on one of K possible mutually exclusive states, then we replace x in *Bernoulli distribution* as a vector $\boldsymbol{x}$,

such as $x = (0, 0, 1, 0)^\top$. The distribution of x is given

$$p(x|\mu) = \prod_{k=1}^{K} \mu_k^{x_k} \tag{10.4}$$

This distribution can be regarded as a generalization of the Bernoulli distribution to more than two outcomes.

10.2.1 Multinomial distribution

We consider the joint distribution of the quantity $m_1, ..., m_K$, conditioned on the parameters μ and on the total number N of observations. m_i is the occurrence number of x_i.

$$\text{Mult}(m_1 m_2 ... m_K|\boldsymbol{\mu}, N) = \binom{N}{m_1, m_2, ...m_K} \prod_{k=1}^{K} \mu_k^{m_k} \tag{10.5}$$

10.2.2 Dirichlet

We now introduce a family of prior distribution for the parameters $\{\mu_k\}$ of the multinomial distribution, *Dirichlet* distribution:

$$\text{Dir}(\boldsymbol{\mu}|\boldsymbol{\alpha}) = \frac{\Gamma(\alpha_0)}{\Gamma(\alpha_1)...\Gamma(\alpha_K)} \prod_{k=1}^{K} \mu_k^{\alpha_k - 1} \tag{10.6}$$

where $\alpha_0 = \sum_{k=1}^{K} \alpha_k$

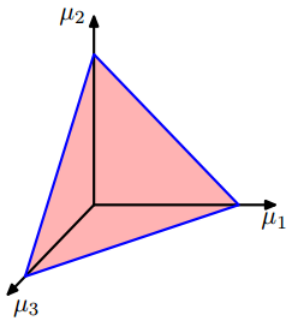

Figure 10.2: The Dirichlet distribution over three variables μ_1, μ_2, μ_3 is confined to a simplex sof the form shown, as a consequence of the constraints $0 \leq \mu_k \leq 1$ and $\sum_k \mu_k = 1$.

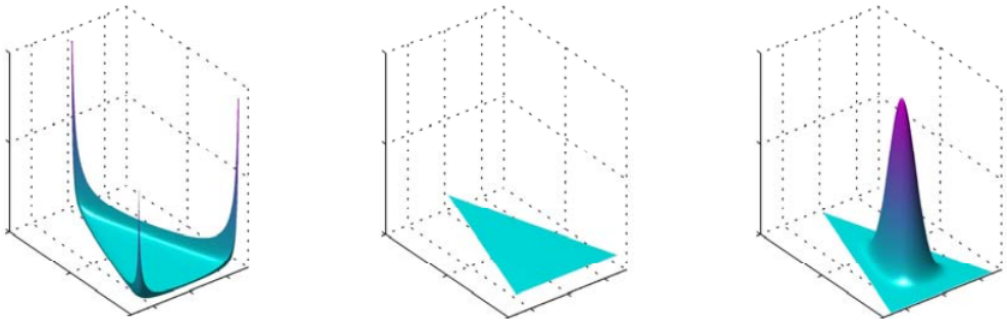

Figure 10.3: Plots of the Dirichlet distribution over three variables, where the two horizontal axes are coordinates in the plane of the simplex and the vertical axis corresponds to the value of the density.

Chapter 11

Linear Models

11.1 Logistic Regression

11.1.1 Logistic sigmoid function

$$f(x; w) = \sigma(w^\top x) = \frac{1}{1 + e^{-w^\top x}}$$

11.1.2 Loss function

The *Logistic Regression* model uses the interpretation of the function as a probability, $f(x; w) = p(y = 1|x, w)$. Maximum likelihood fitting of this model minimizes the negative log-probability of the training labels, which could be written as:

$$NLL = -\sum_{n=1}^{N} \log[\sigma(w^\top x^{(n)})^{y^{(n)}}(1 - \sigma(w^\top x^{(n)}))^{1-y^{(n)}}]$$

or

$$NLL = -\sum_{n:y^{(n)}=1} \log \sigma(w^\top x^{(n)}) - \sum_{n:y^{(n)}=0} \log\left(1 - \sigma(w^\top x^{(n)})\right)$$

or using labels $z^{(n)} \in \{-1, +1\}$ where $z^{(n)} = (2y^{(n)} - 1)$, and noticing $\sigma(-a) = 1 - \sigma(a)$:

$$NLL = -\sum_{n=1}^{N} \log \sigma(z^{(n)} w^\top x^{(n)})$$

11.1.3 Gradient

The derivative of the *logistic sigmoid* as follows:

$$\frac{\partial \sigma(a)}{\partial a} = \sigma(a)(1 - \sigma(a))$$

We use $\sigma_n = \sigma(z^{(n)} w^\top x^{(n)})$, then

$$\begin{aligned} \nabla_w NLL &= -\sum_{n=1}^{N} \nabla_w \log(\sigma_n) = -\sum_{n=1}^{N} \frac{1}{\sigma_n} \nabla_w \sigma_n \\ &= -\sum_{n=1}^{N} \frac{1}{\sigma_n} \sigma_n (1 - \sigma_n) \nabla_w z^{(n)} w^\top x^{(n)} \\ &= -\sum_{n=1}^{N} (1 - \sigma_n) z^{(n)} x^{(n)} \end{aligned} \tag{11.1}$$

Chapter 12

Support Vector Machine

Chapter 13

Dimensionality Reduction

13.1 Linear Discriminant Analysis

Linear Discriminant Analysis or **LDA** is most commonly used as dimensionality reduction technique. The intuition behind LDA is that we would like to *select the one of all possible lines that maximizes the separability of the scalars*.

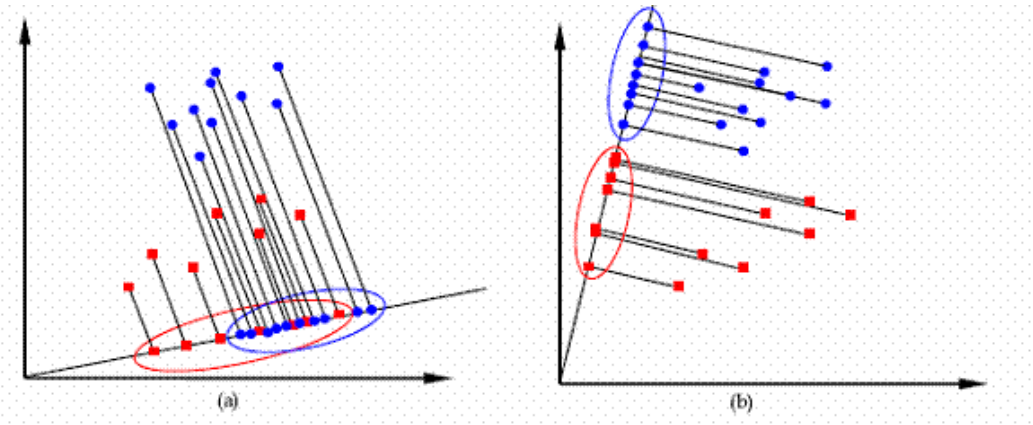

Figure 13.1: An illustration on the intuition behind LDA.

In order to find a good projection, we need to define a measure of separation between projections. The solution is to maximize a function that represents **the difference between the means, normalized by a measure of the within-class variability**.

Given a dataset $D = \{(x_i, y_i)\}_{i=1}^{m}, y_i \in \{0, 1\}$ and X_i, μ_i, Σ_i denotes the **dataset**, **mean vector** and **covariance matrix** with label $i \in \{0, 1\}$, then we

define the J is the objective function to be maximized:

$$\begin{aligned} J &= \frac{\left\| w^\top \mu_0 - w^\top \mu_1 \right\|_2^2}{w^\top \Sigma_0 w + w^\top \Sigma_1 w} \\ &= \frac{w^\top (\mu_0 - \mu_1)(\mu_0 - \mu_1)^\top w}{w^\top (\Sigma_0 + \Sigma_1) w} \end{aligned} \tag{13.1}$$

In order to find the optimum projection w^*, we need to express $J(w)$ as an explicit function of w, where S_w is called *within-class scatter matrix* and S_b is called *between-class scatter matrix*:

$$S_w = \Sigma_0 + \Sigma_1 = \sum_{x \in X_0} (x - \mu_0)(x - \mu_0)^\top + \sum_{x \in X_1} (x - \mu_1)(x - \mu_1)^\top$$

$$S_b = (\mu_0 - \mu_1)(\mu_0 - \mu_1)^\top$$

To find the maximum of $J(w)$, we notice that the optimum solution w^* is irrelevant to its magnitude, thus we assume $w^\top S_w w = 1$, then

$$\begin{aligned} \min_w \quad & w^\top S_b w \\ \text{s.t.} \quad & w^\top S_w w = 1 \end{aligned}$$

With Lagrangian multiplier, we get $S_b w = \lambda S_w w$, notice that the direction of $S_b w$ is always the same as $\mu_0 - \mu_1$, we assume $S_b w = \lambda(\mu_0 - \mu_1)$. As a result, we get

$$w = S_w^{-1}(\mu_0 - \mu_1) \tag{13.2}$$

13.2 Principal Components Analysis

13.2.1 Singular Value Decomposition

In linear algebra, the **singular-value decomposition** (**SVD**) is the generalization of the eigendecomposition of a **positive semidefinite normal matrix** (for example, a symmetric matrix with positive eigenvalues).

Formally, the singular-value decomposition of an $m \times n$ matrix $\mathbf{M}$ is a factorization of the form

$$\mathbf{M} = \mathbf{U}\mathbf{S}\mathbf{V}^\top \tag{13.3}$$

where

- $\mathbf{M}$: an $m \times m$ **unitary** matrix

- **S**: an $m \times n$ rectangular diagonal matrix with non-negative real numbers on the diagonal
- **V**: an $n \times n$ **unitary** matrix

13.2.2 Details

Principal Components Analysis or **PCA** reduces the dimensionality of an $N \times D$ data matrix **X**, by multiplying it by a $D \times K$ matrix **V**. To keep the maths simpler, we will assume for the rest of this note that the D-dimensional feature vectors are **zero mean** (the average of the numbers in each column of **X** is zero). We centre the data to have this property before doing anything else.

We compute the covariance matrix of the points. As we're assuming X is zero-mean, the covariance is $\Sigma = \frac{1}{N} X^\top X$.

$$\begin{aligned} \Sigma &= E(X^\top X) - \mu^\top \mu \\ &= E(X^\top X) \\ &\approx \frac{1}{N} X^\top X \end{aligned} \tag{13.4}$$

The covariance of multivariate Gaussians can be used to describe an ellipsoidal ball that summarizes how the data is spread in space.

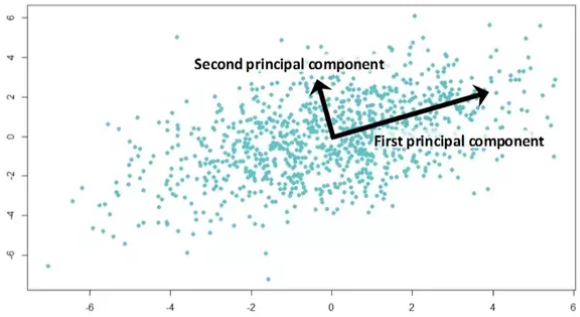

Figure 13.2: PCA example

Some axes of the ellipsoid are often very short, and the data is "squashed" into a ball that only significantly extends in some directions. The **eigenvectors** of the covariance matrix point along the axes of the ellipsoid, and the longest axes are the **eigenvectors** with **the largest eigenvalues**.

13.2.3 PCA and SVD

The truncated SVD view of PCA can find a low-dimensional vector representing either the rows or columns of a matrix. SVD finds both at once.

Singular Value Decomposition (SVD) factors a $N \times D$ matrix into a product of three matrices

$$\mathbf{X} \approx \mathbf{U}\mathbf{S}\mathbf{V}^\top$$

where

- $\mathbf{U}$ has size $N \times K$.
- $\mathbf{S}$ is a diagonal $K \times K$ matrix.
- $\mathbf{V}^\top$ has size $K \times D$.

The $\mathbf{V}$ matrix is the same as before, its columns (or the rows of $\mathbf{V}^\top$) contain eigenvectors of $\mathbf{X}^\top\mathbf{X}$. The columns of $\mathbf{U}$ contain eigenvectors of $\mathbf{X}\mathbf{X}^\top$. The rows of $\mathbf{U}$ give a K-dimensional embedding of the rows of $\mathbf{X}$. The columns of $\mathbf{V}^\top$ (or the rows of V) give a K-dimensional embedding of the columns of $\mathbf{X}$.

A truncated SVD is known to be the best low-rank approximation of a matrix (as measured by square error). PCA is the linear dimensionality reduction method that minimizes the least squares error in the distortion if we project back to the original space: $\mathbf{X} \approx \mathbf{X}\mathbf{V}\mathbf{V}^\top$

13.2.4 Further Details

PCA is sensitive to the scaling of the variables. This means that whenever the different variables have different units (like temperature and mass), PCA is a somewhat arbitrary method of analysis. One way of making the PCA less arbitrary is to use variables scaled so as to have **unit variance**, by **standardizing the data** and hence use the autocorrelation matrix instead of the autocovariance matrix.

Mean subtraction is **necessary** for performing classical PCA to ensure that the first principal component describes the direction of maximum variance. If mean subtraction is not performed, the first principal component might instead correspond more or less to the mean of the data. A mean of zero is needed for finding a basis that minimizes the mean square error of the approximation.

13.2.5 PCA vs. LDA

Both LDA and PCA are linear transformation techniques that are commonly used for dimensionality reduction. PCA can be described as an **"unsupervised"** algorithm while LDA is **"supervised"**.

13.2.6 PCA Whitening

The goal of whitening is to make the input less redundant, that is, the learning algorithms sees a training input where 1) the features are **less correlated** with each other, and 2) the features all have **the same variance**.

Rotating the data

As we mentioned in SVD that the matrix $\mathbf{V}$ is an unitary matrix, $x_{\text{rot}}^{(i)} = x^{(i)}\mathbf{V}$ rotates the original data into orthogonal basis, which satisfies the first requirement — the features are *uncorrelated* (linear uncorrelated).

Unit variance

To make each of our input features have unit variance, we can simply re-scale each feature $x_{\text{rot},i}$ by multiplying $\frac{1}{\sqrt{\lambda_i}}$ where λ_i is its corresponding eigenvalue (calculated in matrix $\mathbf{S}$):

$$x_{\text{PCAwhite}} = \frac{x_{\text{rot},i}}{\sqrt{\lambda_i}} \tag{13.5}$$

This data now has covariance equal to identity matrix $\mathbf{I}$, the difference components of x_{PCAwhite} are **uncorrelated** and have **unit variance**.

13.2.7 ZCA Whitening

Finally, it turns out that this way of getting the data to have covariance identity $\mathbf{I}$ isn't unique. Concretely, if $\mathbf{R}^\top$ is any orthogonal matrix, so that it satisfies $\mathbf{R}\mathbf{R}^\top = \mathbf{R}^\top\mathbf{R} = \mathbf{I}$, then $x_{PCAwhite}\mathbf{R}^\top$ will also have identity covariance. In **ZCA whitening**, we choose $\mathbf{R} = \mathbf{V}$:

$$x_{\text{ZCAwhite}} = x_{\text{PCAwhite}}V^\top \tag{13.6}$$

One defining property of ZCA transformation is that it results in whitened data that is **as close as possible** to the original data (in the least squares

sense). In other words, if you want to minimize $\left\|\mathbf{X} - \mathbf{X}A^\top\right\|^2$ subject to $XA^\top$ being whitened, then you should take $A = W_{ZCA}$. Here is a 2D illustration:

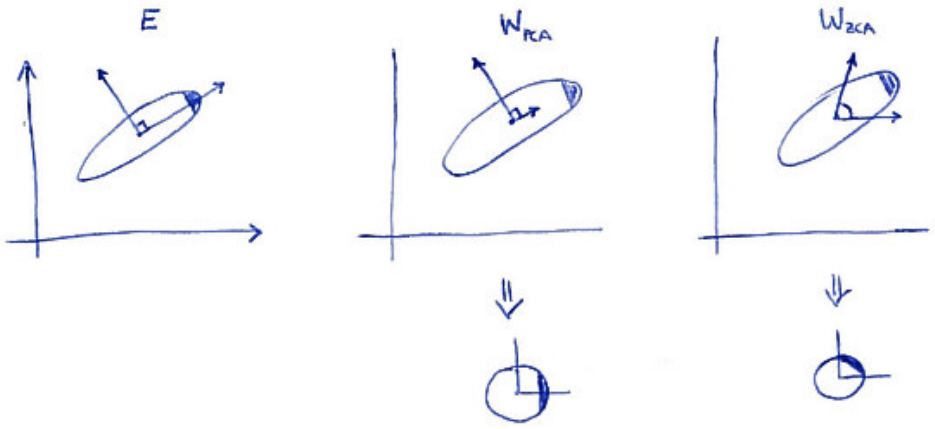

Figure 13.3: The difference of PCA and ZCA Whitening

Chapter 14

Independent Component Analysis

Independent Component Analysis or **ICA**, similar to PCA, this will find a new basis in which to represent the data, but with a very different goal.

14.1 Cocktail Party Problems

Here, n speakers are speaking simultaneously at a party, and any microphone placed in the room records only an overlapping combination of the n speakers' voices. But let's say we have n different microphones placed in the room, and because each microphone is a different distance from each of the speakers, it records a different combination of the speakers' voices. How can we separate out the original n speakers' voices.

Chapter 15

Graphical Models

15.1 Bayesian Networks

15.2 Markov Random Fields

15.2.1 Conditional Random Field

Chapter 16

Boosting

16.1 XGBoost

XGBoost stands for "Extreme Gradient Boosting", where the term "Gradient Boosting" originates from the paper *Greedy Function Approximation: A Gradient Boosting Machine*. The XGBoost is a decision tree ensemble model, whose objective function can be written in a general way as

$$obj = \sum_{i=1}^{n} l(y_i, \hat{y}_t^{(t)}) + \sum_{i=1}^{t} \Omega(f_i)$$

16.1.1 Additive Training

In order to find the **parameters** of trees, we need to learn those functions f_i, each containing the structure of the tree and the leaf scores. It is intractable to learn all trees at once, we use an additive strategy: fix what we have learned, and add one new tree at a time. We write the prediction value at step t as $\hat{y}_i^{(t)}$. Then we have

$$\begin{aligned}
\hat{y}_i^{(0)} &= 0 \\
\hat{y}_i^{(1)} &= f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i) \\
\hat{y}_i^{(2)} &= f_1(x_i) + f_2(x_i) = \hat{y}_i^{(1)} + f_2(x_i) \\
&\dots \\
\hat{y}_i^{(t)} &= \sum_{k=1}^{t} f_k(x_i) = \hat{y}_i^{(t-1)} + f_t(x_i)
\end{aligned} \tag{16.1}$$

It remains to ask: which tree do we want at each step? A natural thing is to add the one that optimizes our objective.

$$\begin{aligned} \text{obj}^{(t)} &= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t)}) + \sum_{i=1}^{t} \Omega(f_i) \\ &= \sum_{i=1}^{n} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + \text{constant} \end{aligned} \tag{16.2}$$

If we consider using mean squared error (MSE) as our loss function, the objective becomes

$$\begin{aligned} \text{obj}^{(t)} &= \sum_{i=1}^{n} (y_i - (\hat{y}_i^{(t-1)} + f_t(x_i)))^2 + \sum_{i=1}^{t} \Omega(f_i) \\ &= \sum_{i=1}^{n} [2(\hat{y}_i^{(t-1)} - y_i) f_t(x_i) + f_t(x_i)^2] + \Omega(f_t) + \text{constant} \end{aligned} \tag{16.3}$$

The form of MSE is friendly, with a first order term (usually called the residual) and a quadratic term. For other losses of interest (for example, logistic loss), it is **not so easy to get such a nice form**. So in the general case, we take the **Taylor expansion of the loss function up to the second order**:

$$\text{obj}^{(t)} = \sum_{i=1}^{n} [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) + \text{constant} \tag{16.4}$$

where the g_i and h_i are defined as

$$\begin{aligned} g_i &= \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}) \\ h_i &= \partial^2_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}) \end{aligned} \tag{16.5}$$

After we remove all the constants, the specific objective at step t becomes

$$\sum_{i=1}^{n} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \tag{16.6}$$

16.1.2 Model Complexity

We first refine the definition of the tree $f(x)$ as

$$f_t(x) = w_{q(x)}, w \in \mathbb{R}^T, q : \mathbb{R}^d \to \{1, 2, ..., T\}$$

Here w is the vector of scores on leaves, q is a function assigning each data point to the corresponding leaf, and T is the number of leaves. In XGBoost, we define the regularization term $\Omega(f)$ as follows:

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{16.7}$$

16.1.3 The Structure Score

We can write the objective value with the t-th tree as:

$$\begin{aligned} \text{obj}^{(t)} &\approx \sum_{i=1}^{n} [g_i w_{q(x_i)} + \frac{1}{2} h_i w_{q(x_i)}^2] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \\ &= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T \end{aligned} \tag{16.8}$$

where $I_j = \{i | q(x_i) = j\}$ is the set of indices of data points assigned to the j-th leaf. Notice that in the second line we have changed the index of the summation because all the data points on the same leaf get the same score. We could further compress the expression by defining $G_j = \sum_{i \in I_j} g_i$ and $H_j = \sum_{i \in I_j} h_i$:

$$\text{obj}^{(t)} = \sum_{j=1}^{T} [G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2] + \gamma T \tag{16.9}$$

In this equation, wj are independent with respect to each other, the form $G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2$ is quadratic and the best w_j for a given structure $q(x)$ and the best objective reduction we can get is:

$$\begin{aligned} w_j^* &= -\frac{G_j}{H_j + \lambda} \\ \text{obj}^* &= -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} + \gamma T \end{aligned} \tag{16.10}$$

The last equation measures how good a tree structure $q(x)$ is.

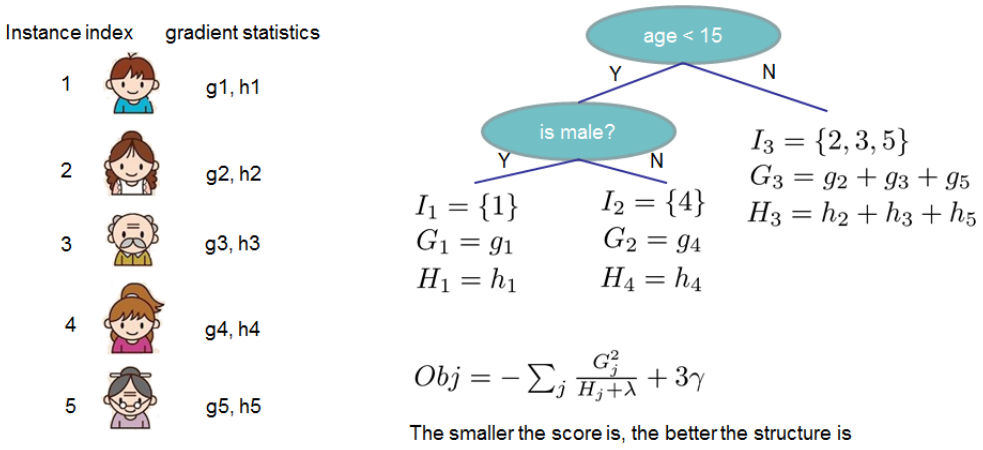

Figure 16.1: Example of how objective of XGBoost is calculated.

16.1.4 Learn The Tree Structure

Now that we have a way to measure how good a tree is, ideally we would enumerate all possible trees and pick the best one. In practice this is intractable, so we will try to optimize one level of the tree at a time. Specifically we try to split a leaf into two leaves, and the score it gains is

$$Gain = \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\right] - \gamma \tag{16.11}$$

This formula can be decomposed as

1. the score on the new left leaf
2. the score on the new right leaf
3. the score on the original leaf
4. the regularization on the additional leaf

We can see an important fact here: if the gain is smaller than γ, we would do better not to add that branch. This is exactly the **pruning** techniques in tree based models! By using the principles of supervised learning, we can naturally come up with the reason these techniques work :)

For real valued data, we usually want to search for an optimal split. To efficiently do so, we place all the instances in sorted order, like the following picture. A left to right scan is sufficient to calculate the structure score of all possible split solutions, and we can find the best split efficiently.

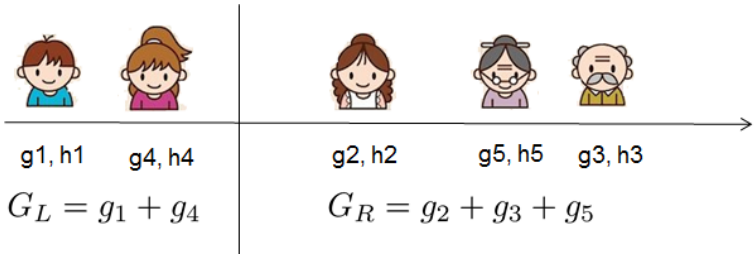

Figure 16.2: How to find splits in XGBoost.

Part IV

Algorithm

Chapter 17

Trie

Trie is a kind of digital search tree. Trie is an efficient indexing model, which is indeed also a kind of **deterministic finite automaton** (DFA).

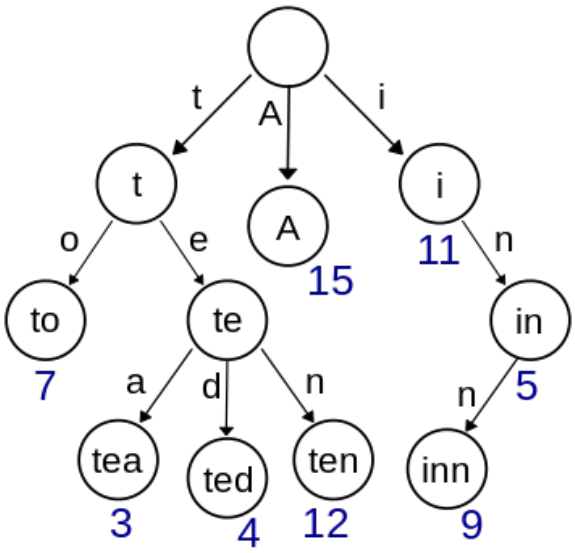

Figure 17.1: Trie

17.1 Tripple-Array Trie

17.1.1 Structure

The tripple-array structure is composed of:

- **base**: Each element in base corresponds to a node of the trie. For a trie node s, $base[s]$ is the starting index within the next and check pool for the row of the node s in the transition table.
- **next**: This array, in coordination with check, provides a pool for the allocation of the sparse vectors for the rows in the trie transition table. The vector data, that is, the vector of transitions from every node, would be stored in this array.
- **check**: This array works in parallel to next. It marks the owner of every cell in next. This allows the cells next to one another to be allocated to different trie nodes. That means the sparse vectors of transitions from more than one node are allowed to be overlapped.

Definition: For a transition from state s to t which takes character c as the input, the condition maintained in the tripple-array trie is:

$$\begin{aligned} check[base[s] + c] &= s \\ next[base[s] + c] &= s \end{aligned} \tag{17.1}$$

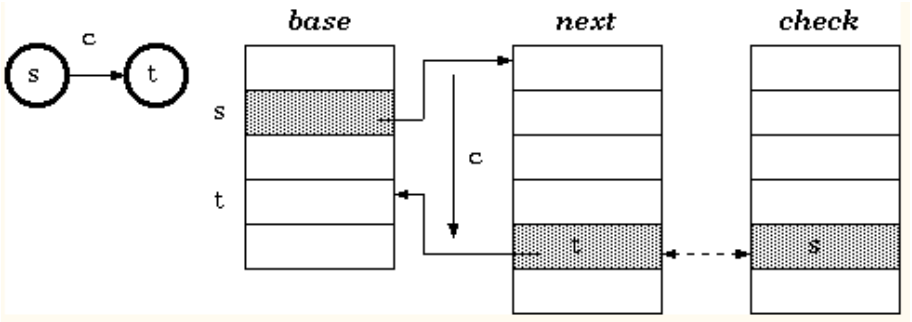

Figure 17.2: Tripple-array trie

17.1.2 Walking

$t \leftarrow base[s] + c$
if $check[t] = s$ **then**
  nextstate $\leftarrow next[t]$
else
  fail
end if

17.2 Double-Array Trie

The tripple-array structure for implementing trie appears to be well defined, but is still not practical to keep in a single file. The *next/check* pool may be able to keep in a single array of integer couples, but the base array does not grow in parallel to the pool, and is therefore usually split.

17.2.1 Structure

Instead of indirectly referencing through state numbers as in tripple-array trie, nodes in double-array trie are linked directly within the **base/check** pool. **Definition**: For a transition from state s to t which takes character c as the input, the condition maintained in the double-array trie is:

$$\begin{aligned} check[base[s] + c] &= s \\ base[s] + c &= t \end{aligned} \tag{17.2}$$

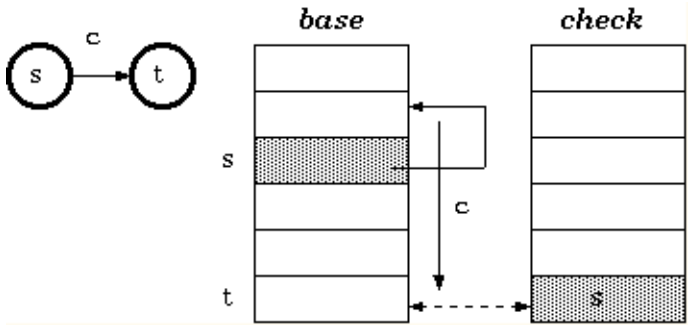

Figure 17.3: Double-array trie

17.2.2 Walking

$t \leftarrow base[s] + c$
if $check[t] = s$ **then**
  nextstate $\leftarrow t$
else
  fail
end if

17.3 An Efficient Language Model Using Double-Array Structure

Part V

Extreme Computing

Chapter 18

Extreme Computing Algorithms

18.1 Bloom Filter

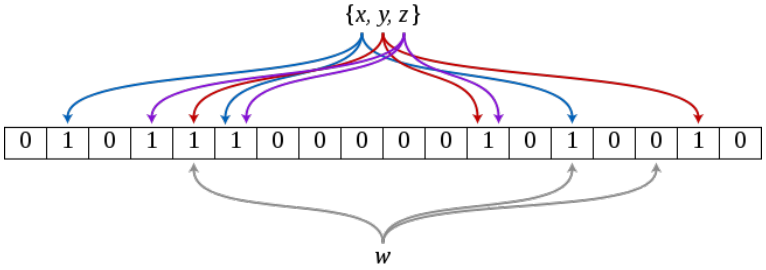

Figure 18.1: Bloom filter structure

The structure of a bloom filter can be explained here. Essentially, a bloom filter is implemented by a bit array with N entries and a set of k hash functions, $h_1, ..., h_k$. We assume the hash functions are independent and map a uniformly random selected key to every entry with equal probability. Initially, all entries in the bit array are then set to zero. Insertion and query operations define following actions:

- **Insert key** x: with k hash functions, we compute $h_1(x), ..., h_k(x)$ values and set the entries location in positions $h_1(x), ..., h_k(x)$ in the bit array with 1.
- **Query key** x: given the key w, we compute $h_1(w), ..., h_k(w)$, if all values of entries located in position $h_1(w), ..., h_k(w)$ are 1 then the bloom filter returns true. Otherwise, false is returned.

The **false positive rate** in the bloom filter depends on m (**the number of bits in array**), k (**the number of hash functions**) and n (**the number of inserted elements**).

the probability of a bit is not set to 1 by a specific hash function in one insertion is

$$1 - \frac{1}{m}$$

Note that hash functions are independent; thus, the probability of a bit not set to 1 by k hash functions is just

$$(1 - \frac{1}{m})^k$$

After n elements insertion, the probability becomes

$$(1 - \frac{1}{m})^{kn}$$

Now we consider testing a key w which is not in the set, then the probability that the k computed positions $h_1(w), ...h_k(w)$ by hash functions $h_1, ...h_k$ are all 1 is

$$(1 - [1 - \frac{1}{m}]^{kn})^k \approx (1 - e^{-\frac{kn}{m}})^k$$

18.2 Resevoir Sampling

Resovoir sampling enables sampling from a large dataset with one iteration, which is broadly applied in search engines.

18.2.1 Sample One Line

```
// Uniformly sample one line
import sys
import random

line_number = 0
for line in sys.stdin:
   if random.randint(0, line_number) == 0:
      resevoir = line.strip()
   line_number += 1
print(resevoir)
```

18.2.2 Proof

- **Base**: One line with probability 1.
- **Inductive**: Assume n lines were sampled with probability $\frac{1}{n}$ each. When the $n+1$th line is added, the resevoir is kept with probability $\frac{n}{n+1}$. Thus the first n lines each have probability

$$\frac{1}{n} \times \frac{n}{n+1} = \frac{1}{n+1}$$

  And the $n+1$th line also has probability $\frac{1}{n+1}$ by construction.

18.2.3 Sample Multiple Lines

```
// Uniformly sample multiple lines
import sys
import random

SAMPLES = 10
resevoir = []

# Fill resevoir
for i in range(SAMPLES):
   resevoir.append(sys.stdin.readline())
line_number = SAMPLES

# Substitute an entry with probability |samples|/|lines|
for line in sys.stdin:
   generated = random.randint(0, line_number)
   if generated < SAMPLES:
      resevoir[generated] = line
   line_number += 1
print("".join(resevoir), end="")
```

Chapter 19

Frequent Pattern Mining

"- Which pages are getting an unusual hit in the last 30 minutes?
- Which categories of items are now hot?

Frequent pattern mining refers to finding patterns that occur more frequently than a pre-specified threshold value. Frequent pattern mining over data streams differ from conventional one:

- can not afford multiple passes
  - minimize requirements in terms of memory
  - trade off between storage, complexity and accuracy
  - only get one look
- frequent items and itemsets are usually the final output

19.1 Lossy Counting

Step 1: Divide the incoming data stream into windows.

Step 2: Increment the frequency count of each item according to the new window values. After each window, decrement

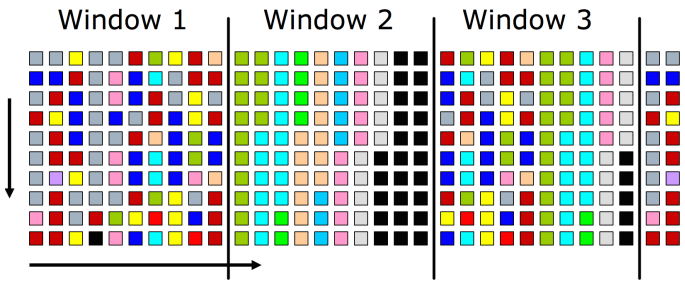

Figure 19.1: Split input stream into windows

all counters by 1.

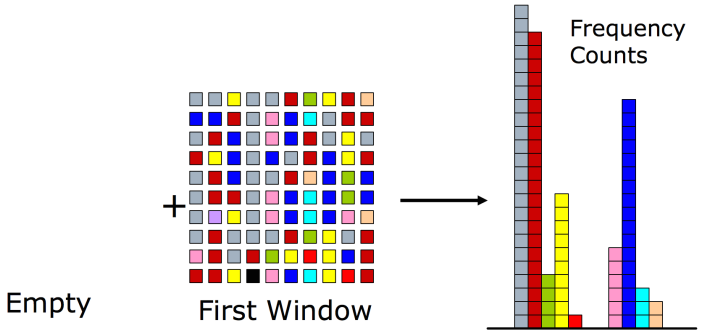

Figure 19.2: Increment frequency counts — At window boundary adjust counts

Step 3: Repeat — Update counters and after each window, decrement all counters by 1.

19.1.1 Implementation

1. user supplies two parameters support s and error ϵ
2. simple data structure, maintaining triplets of data items e, their associated frequencies f, and the maximum possible error Δ in f: (e, f, Δ).
3. The stream is conceptually divided into buckets of width $w = \frac{1}{\epsilon}$ — Each bucket labeled by a value $\frac{N}{w}$ where N starts

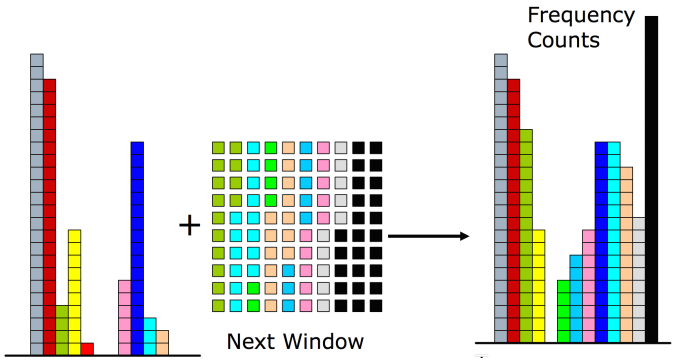

Figure 19.3: Update counters and at window boundary decrement all items by 1

from 1 and increases by 1.

4. For each incoming item, the data structure is checked
   - If an entry exists, increment frequency
   - Otherwise, add new entry with $\Delta = b_{current} - 1$ where $b_{current}$ is the current bucket label
5. When switching to a new bucket, all entries with $f + \Delta < b_{current}$ are released.

19.1.2 Output

The most frequently viewed items "survive". Given a frequency threshold f, a frequency error ϵ, and total number of elements N, then frequency error $\leq$ number of windows (ϵN). Thus, the output can be expressed as follows: Elements with count exceeding $fN - \epsilon N$.

Worst case we need $\frac{1}{\epsilon} * \log(\epsilon N)$ counters.

For example, we may want to print the Facebook pages of people who get hit more than 20%, with an error threshold

of 2%. For frequency $f = 20\%$, $\epsilon = 2\%$, all elements with true frequency exceeding $f = 20\%$ will be output — there are no false negatives. But we undercount. The output frequency of an element can be less than its true frequency by at most 2%. False positives could appear with frequency between 18% – 20%. Last, no element with frequency less than 18% will be output.

Given window of size $\frac{1}{\epsilon}$, the following guarantees hold:

- Frequencies are underestimated by at most $\epsilon * N$
- No false negatives
- False positives have true frequency of at least $f * N - \epsilon * N$

19.2 Sticky Sampling

This is another technique for counting but probabilistic. Here the space consumption is **independent** of the length of the stream N.

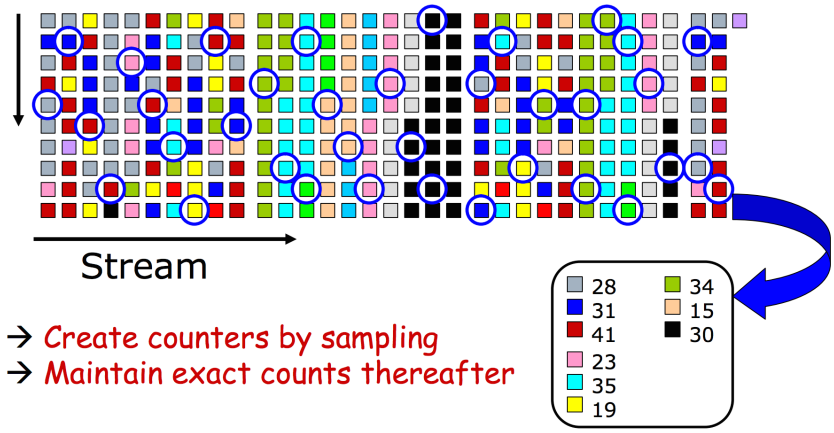

Figure 19.4: Sticky sampling

The user defines the frequency threshold f, the error ϵ but also the probability of failure δ.

The logic is the following: For each incoming item, if an entry exists we increase its frequency. Otherwise we may select the item based on a probability $\frac{1}{r}$. This is called sampling. We gradually increase the sampling rate (starting at $r = 1$) as more elements are processed. Upon a sampling rate change, we scan all existing entries. For each entry, we toss an unbiased coin and we decrease the item's frequency for each unsuccessful toss, until the coin toss becomes successful or the frequency becomes 0 at which we release the entry.

The sampling rate is adjusted according to the following formula

$$t = \frac{1}{\epsilon} * \log\left(\frac{1}{f * \delta}\right)$$

The first t elements are sampled at rate $r = 1$, the next $2t$ elements at rate $r = 2$, the next $4t$ at rate $r = 4$ etc.

19.2.1 Output

Same as lossy counting: Given a frequency threshold f, a frequency error ϵ, and total number of elements N, the output can be expressed as follows: Elements with count exceeding $fN - \epsilon N$.

Number of counters expected (probabilistically): $\frac{2}{\epsilon} * \log\left(\frac{1}{f*\delta}\right)$

It's worth noting that the number of counters here is independent of N, the length of the stream. The same guarantees as lossy counting apply here:

- Frequencies are underestimated by at most $\epsilon * N$
- No false negatives
- False positives have true frequency of at least $f * N - \epsilon * N$

19.3 Comparison

- Lossy counting is more accurate but Sticky Sampling requires **constant space**. Lossy Counting space requirements **increase logarithmically** with the length of the stream N.
- Sticky sampling remembers every unique element that gets sampled, whereas Lossy Counting chops off low frequency elements quickly leaving only the high frequency ones.
- Sticky sampling can support infinite streams while keeping the same error guarantees as Lossy Counting.

Part VI

Front-end Web Development

Chapter 20

HTML5

HTML5 is the World Wide Web's core markup language. Originally, HTML was primarily designed as a language for semantically describing scientific documents.

Bibliography