COMP550 Notes

Chany Ahn

September 2021 - December 2021

Contents

1 NLP and CL

1.1 Domains of Language

- **Phonetics**: Study of speech sounds that make up language.
- **Phonology**: Study of rules that govern sound patterns and how they're organized.
- **Morphology**: Word formation and meaning.
- **Syntax**: Study of the structure of language.
- **Semantics**: Study of the meaning of language.
- **Pragmatics**: Study of the meaning of language in context.
  - **Deixis**: Interpretation of expressions can depend on extralinguistic context.
- **Discourse**: Study of the structure of larger spans of language.

2 Text Classification

2.1 Feature Extraction

- An input can be represented as a vector, $\vec{x}$, where the feature inputs can be binary: whether or no the word is in a given piece of text.
- **Lemma**: Removes affixes and recovers the lemma (form that you look up in the dictionary).
  - Achieved using finite state automata.
- **Stemming**: Cuts affixes to find the stem.
  - Porter stemming: An ordered list of of rewrite rules to approximately recover the stem of a word. The basic idea is to chop off and append endings back onto a word.

2.1.1 N-Grams

Sequences of *adjacent words*. Version of N-Grams are:

- Prescence or abscence of N-grams (0 or 1).
- Count of an N-gram.
- Proportion of the total document.
- Scaled versions of counts (ex. give common word lower weights and uncommon words larger weights).

2.1.2 POS Tags

Sequence of POS tags crudely captures syntatic patterns in text. The document must be pre-processed for their POS tags.

2.1.3 Stop Words

Common words may not be useful for some document classification tasks (highly task-dependent). Standardized list of such stop words are commonly removed.

3 Linear Classifiers

3.1 Type/Token Distinction

- **Type**: the identity of a word (i.e., count unique words).
- **Token**: an instance of a word (i.e., each occurrence is separate).

Text classifications usually deals with tokens and assumes a categorical distribution that is used to generate all of the tokens seen in a sample, conditioned on class y.

- Example: *yo buy my stuff yo*, class: *spam*.

$$P(spam)P(yo|spam)P(my|spam)P(stuff|spam)P(yo|spam)$$

3.2 Generative vs. Discriminative

- **Generative** models learn a distribution for all random variables involved: joint distribution, $P(\vec{x}, y)$.
- **Discriminative** models directly parameterize and learn $P(y|\vec{x})$ (better for text classification).

3.3 Naive Bayes

A probabilistic classifier that uses Bayes rule:

$$P(y|\vec{x}) = \frac{P(y)P(\vec{x}|y)}{P(\vec{x})} \tag{1}$$

Naive Bayes is a generative model:

- Probabilistic account of the data $P(\vec{x}, y)$.
- Naive Bayes assumes the dataset is generated in the following way, for each sample:
  1. Generate label by $P(y)$.
  2. Generate feature vector $\vec{x}$ by generating each feature independently, conditioned on y: $P(x_i|y)$.

So, the Naive Bayes assumption is:

$$P(\vec{x}, y) = P(y)\prod_i P(x_i|y) \tag{2}$$

3.3.1 Naive Bayes Model Parameters

The parameters to the model, θ, consist of:

- Parameters of prior class distribution $P(y)$.
- Parameters of each feature's distribution conditioned on class $P(x_i|y)$.

With discrete data, we assume that the distributions $P(y)$ and $P(x_i|y)$ are categorical distributions.

3.3.2 Training

Objective: pick θ to mazimize the likelihood of the training corpus, $\mathcal{D}$:

$$L^{NB}(\theta) = \prod_{(\vec{x},y)\in\mathcal{D}} P(\vec{x}, y; \theta) \tag{3}$$

$$= \prod_{(\vec{x},y)\in\mathcal{D}} P(y)\prod_i P(x_i|y) \tag{4}$$

This boils down to computing *relative frequencies*:

- $P(Y = y)$ should be set to proportion of samples that are within class y.
- $P(X_i = x|Y = y)$ should be set to proportion of samples with feature value x among samples of class y.

3.3.3 Inference

After training, we would like to classify a new instance (i.e., we want $P(y|\vec{x})$). This is easy to get from $P(\vec{x}, y)$:

$$
\begin{aligned}
P(y|\vec{x}) &= \frac{P(\vec{x}, y)}{P(\vec{x})} && (5) \\
&= \frac{P(y)\prod_i P(x_i|y)}{P(\vec{x})} && (6)
\end{aligned}
$$

Calculate $P(\vec{x})$ marginalize over random variable y by summing up numerator for all possible classes (Law of Total Probability).

3.4 Logistic Regression

Logit function (a.k.a., maximum entropy or MaxEnt classifier):

$$P(y|\vec{x}) = \frac{1}{Z} e^{a_1x_1+a_2x_2+\ldots+a_nx_n+b} \tag{7}$$

Gives continuous values in the interval $[0, 1]$, where Z is a normalizing constant to insure that it is a probability distribution. Typically used for classification problems (not really a regression model).

3.4.1 Logistic Function

Graphically, the function is as follows:

- y-axis: $P(y|\vec{x}) = \frac{1}{Z} e^{a_1x_1+a_2x_2+\ldots+a_nx_n+b}$
- x-axis: $a_1x_1 + a_2x_2 + \ldots + a_nx_n + b$

3.4.2 Features

Features depend on both the document and the proposed class.

3.4.3 Parameters in Logistic Regression

$$
\begin{aligned}
P(y|\vec{x}) &= \frac{1}{Z} e^{a_1x_1+a_2x_2+\ldots+a_nx_n+b} && (8) \\
\text{where } \theta &= \{a_1, a_2, \ldots, a_n, b\} && (9)
\end{aligned}
$$

Learning means to maximize the conditional likelihood of the training corpus.

$$
\begin{aligned}
L^{LR}(\theta) &= \prod_{(\vec{x},y)\in\mathcal{D}} P(y|\vec{x}; \theta) && (10) \\
\log L^{LR}(\theta) &= \sum_{(\vec{x},y)\in\mathcal{D}} \log P(y|\vec{x}; \theta) && (11)
\end{aligned}
$$

3.4.4 Optimizing the Objective

We want to maximize:

$$\begin{aligned}
\log L^{LR}(\theta) &= \sum_{(\vec{x},y)\in\mathcal{D}} \log P(y|\vec{x};\theta) && (12)\\
&= \sum_{(\vec{x},y)\in\mathcal{D}} \log\left(\frac{1}{Z}e^{a_1x_1+a_2x_2+\ldots+a_nx_n+b}\right) && (13)\\
&= \sum_{(\vec{x},y)\in\mathcal{D}} \left(\sum_i a_ix_i - \log Z\right) && (14)
\end{aligned}$$

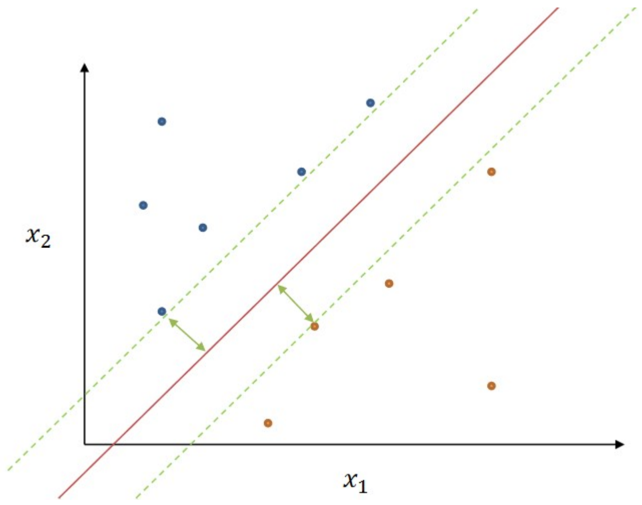
which is done using **gradient descent**.

3.5 Support Vector Machines

Let $\vec{x} \in \mathbb{R}^d$. A SVM learns a decision boundary as a line (or a hyperplane when there are > 2 features).

3.5.1 Margin

The hyperplane is chosen to maximize the margin to the nearest sample in each class. This method also deals with the fact that the samples *may not be linearly separable.*

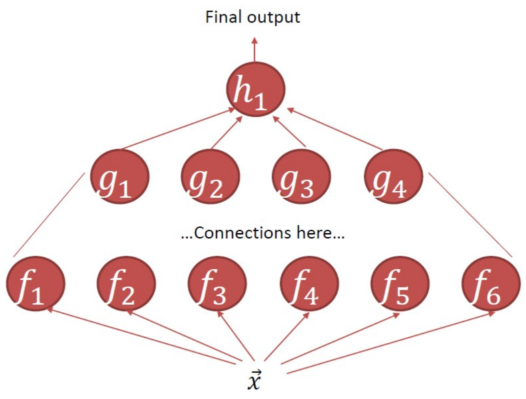


3.6 Perceptron

Closely related to logistic regression (differences in training and output interpretation):

$$f(\vec{x}) = \begin{cases} 1 & \text{if } \vec{w}\cdot\vec{x}+b > 0 \\ 0 & \text{otherwise} \end{cases} \tag{15}$$

3.6.1 Stacked Perceptrons



3.7 Artificial Neural Networks

Above is an example of an artificial neural network:

- Each unit is a neuron with many inputs and one output.
- The nucleus fires given input from other neurons.
- Learning occurs at the synapses that connect neurons, either by amplifying or attenuating signals.

Advantages:

- Can learn very complex functions.
- Many possible different network structures possible.
- Given enough training data, are currently achieving the best results in many NLP tasks.

Disadvantages:

- Training takes a long time.
- Often need a lot of training data to work well.

3.8 Other Classification Algorithms

- KNN
- Decision trees
- Transformation-based learning
- Random forests

4 Nonlinear Classifiers

Linear models cannot learn comlpex, non-linear functions from input features to output labels (without adding features).

- Ex: Starts with a capital AND not at beginning of a sentence $\rightarrow$ proper noun (this is tough to represent with a linear model).

4.1 (Artificial) Neural Network

A kind of learning model which automatically learns non-linear functions from input to output. From the biologically inspired metaphor:

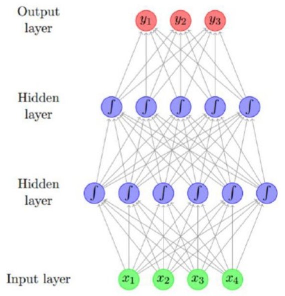
- Network of computational units called neurons.
- Each neuron takes scalar inputs and produces a scalar output. This is similar to a logistic regression model:

$$Neuron(\vec{x}) = g(a_1x_1 + \ldots + a_nx_n + b)$$

The network can, theoretically, compute any computable function, given enough neurons.

4.2 Feedforward Neural Networks

All connections flow forward (no loops); each layer of hidden units is fully connected to the next.



4.2.1 Inference in a FF Neural Network

Perform computations forwards through the graph:

$$\mathbf{h^1} = g^1(\mathbf{xW^1} + \mathbf{b^1}) \tag{16}$$
$$\mathbf{h^2} = g^2(\mathbf{xW^2} + \mathbf{b^2}) \tag{17}$$
$$\mathbf{y} = \mathbf{h^2W^3} \tag{18}$$

Note, each layer is represented as a vector; combining all the weights in a layer across the units into a weight matrix.

4.2.2 Activation Function

In one unit: *Linear comination of inputs and weight values* $\rightarrow$ *non-linearity.*

$$\mathbf{h^1} = g^1(\mathbf{xW^1} + \mathbf{b^1})$$

Some popular choices:

- Sigmoid function
- tanh function
- Rectifier/ramp function: $g(x) = \max(0, x)$

Why is non-linearity important?

4.3 Softmax Function

In NLP, we often care about discrete outcomes. Output layer can be constructed such that the output values sum to one: let $\mathbf{x} = x_1 \dots x_k$.

$$softmax(x_i) = \frac{exp(x_i)}{\sum_j^k exp(x_j)} \tag{19}$$

Interpretation: unit x_i represents probability that outcome is i. Essentially, the last layer is like a multinomial logistic regression.

4.4 Loss Function

A neural network is optimized with respect to a *loss function* which measures how much error it is making on predictions:

- $\mathbf{y}$: correct, gold-standard distribution over class labels.
- $\hat{\mathbf{y}}$: system predicted distribution over class labels.
- $L(\mathbf{y}, \hat{\mathbf{y}})$: loss function between the two.

A popular choice for classification is *cross entropy* (especially with a softmax output layer):

$$L_{ce}(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_i y_i \log(\hat{y}_i) \tag{20}$$

4.5 Training Neural Networks

Typically done by *gradient descent*, we find the gradient of the loss function wrt parameters of the network (i.e., the weights of each layer). Since the network can have a lot of parameters, we use *back-propagation* which is an efficient algorithm to compute the gradient wrt to all parameters.

This boils down to an efficient way to use the chain rule of derivatives to propagate the error signal from the loss function backwards through the network back to the inputs.

4.5.1 Gradient Descent

Descent vs. Ascent: The convention is to think about the problem as a minimization problem. As stated before, we must minimize the loss function

$$\theta \leftarrow \theta - \gamma(\nabla L(\theta)) \tag{21}$$

We do this by first initializing $\theta = \{\theta_1, \theta_2, \dots, \theta_k\}$ randomly. The do the following algorithm for a while:

- Compute $\nabla L(\theta)$ (through calculus)
- $\theta \leftarrow \theta - \gamma(\nabla L(\theta))$

Note: the gradient descent algorithm is computed over the entire training corpus.

- Sum over all samples in training corpus.
- Weight update *once* per iteration through the training corpus.

4.5.2 Stochastic Gradient Descent

SGD calculates the gradient over a small mini-batch of the training corpus and updates the weights. If the mini-batch size is one:

- Many weight updates per iteration through the training corpus.
- Usually results in much faster convergence to final solution, without loss in performance.

Overview:

- Inputs:
  - Function computed by neural network: $f(\mathbf{x}; \theta)$.
  - Training samples: $\{\mathbf{x}^k, \mathbf{y}^k\}$
  - Loss function: L
- Algorithm: Repeat for a while
  - Sample of training cases: $\{\mathbf{x}^k, \mathbf{y}^k\}$
  - Compute loss: $L(f(\mathbf{x}^k; \theta), \mathbf{y}^k)$ (forward pass).
  - Compute gradient $\nabla L(\mathbf{x}^k)$ wrt the parameters θ (In neural networks, use backpropagation).
  - Update $\theta \leftarrow \theta - \gamma(\nabla L(\theta))$

Example:
Forward Pass

$$\mathbf{h^1} = g^1(\mathbf{xW^1} + \mathbf{b^1})$$
$$\mathbf{h^2} = g^2(\mathbf{xW^2} + \mathbf{b^2})$$
$$f(\mathbf{x}) = \mathbf{y} = g^3(\mathbf{h^2}) = \mathbf{h^2W^3}$$

Loss function: $L(\mathbf{y}, \mathbf{y}^{gold})$. Remember that we must save the values for $\mathbf{h^1}, \mathbf{h^2}, \mathbf{y}$.
Backpropagation

$$f(\mathbf{x}) = g^3(g^2(g^1(\mathbf{x})))$$

Need to compute: $\frac{\partial L}{\partial \mathbf{W}^3}, \frac{\partial L}{\partial \mathbf{W}^2}, \frac{\partial L}{\partial \mathbf{W}}$. By calculus and chain rule:

- $\frac{\partial L}{\partial \mathbf{W}^3} = \frac{\partial L}{\partial g^3} \frac{\partial g^3}{\partial \mathbf{W}^3}$
- $\frac{\partial L}{\partial \mathbf{W}^2} = \frac{\partial L}{\partial g^3} \frac{\partial g^3}{\partial g^2} \frac{\partial g^2}{\partial \mathbf{W}^2}$
- $\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial g^3} \frac{\partial g^3}{\partial g^2} \frac{\partial g^2}{\partial g} \frac{\partial g}{\partial \mathbf{W}}$

4.6 Word Representations

Previous word representations created vectors that held counts for each word found in the corpus: $[w_1\ w_2\ w_3\ w_4] =$ [count$_1$ count$_2$ count$_3$ count$_4$].

A more typical choice is to associate each word type with a fixed-dimensional vector which are model parameters: (don't really get this part)

$$w_1\ [0.3, 0.5, 0.6]$$
$$w_2\ [-0.1, 0.2, 0.7]$$

4.7 Sentence Representations

To represent an input sentence or document, need to combine the input word vector representations:

- Simple vector addition:
  this is a sentence $\rightarrow v^{\text{this}}\ v^{\text{is}}\ v^{\text{a}}\ v^{\text{sentence}}$ (look-up layer)
  $s = v^{\text{this}} + v^{\text{is}} + v^{\text{a}} + v^{\text{sentence}}$
- Component-wise vector multiplication
  $s = v^{\text{this}} \odot v^{\text{is}} \odot v^{\text{a}} \odot v^{\text{sentence}}$

There a many, sophisticated possible options.

4.8 Hardware For NN

Common operations in inference and learning:

- Matrix multiplication
- Component-wise operations (e.g., activation functions)

These operations are highly parallelizable. Graphical processing units (GPUs) are specifically designed to perform this type of computation efficiently.

4.9 Summary

Advantages:

- Learn relationships between inputs and outputs.
  - Complex features and dependencies between inputs and states over long ranges with no fixed horizon assumptions (i.e., non-Markovian).
  - Reduces need for feature engineering.
  - More efficient use of input data via weight sharing.
- Highly flexible, generic architecture.
  - **Multi-task learning**: jointly train model that solves multiple tasks simultaneously.
  - **Transfer learning**: Take part of a neural network used for an initial task, use that as an initialization for a second, related task.

Challenges: complex models may need a lot of training data. There are many fiddly hyperparameters to tune, little guidance on how to do so, except empirically or through experience:

- Learning rate, number of hidden units, number of hidden layers, how to connect units, non-linearity, loss function, how to sample data, training procedure, etc.

It can be difficult to interpret the output of a system.

- Why did the model predict a certain label? We would have to examine weights in the network.
- Important to convince people to act on the outputs of the model!

4.10 NNs for NLP

NN have taken over mainstream NLP, and most empirical work at recent conferences use them in some way. Some interesting research questions:

- How to use linguistic structure (e.g., word senses, parses, other resources) with NNs, either as input or output?
- When is linguistic feature engineering a good idea, rather than just throwing more data with a simple representation for the NN to learn the features?
- Multitask and transfer for NLP.
- Defining and solving new, challenging NLP tasks.

4.11 Evaluation Measures

To measure the performance of your classifier, the simplest option is **accuracy**:

$$\frac{\#\text{ correct}}{\#\text{ samples in test set}} \tag{22}$$

This is not always good. Consider the following case: Suppose that only one of every twenty emails is a *spam* email. What would the accuracy of the classifier that always predicts *non-spam* be? (What's the answer?)

4.11.1 Precision and Recall

$$\textbf{Precision} = \frac{\#\text{ correct}}{\#\text{ predicted}} \tag{23}$$

$$\textbf{Recall} = \frac{\#\text{ correct}}{\#\text{ of that class}} \tag{24}$$

4.11.2 Combining Precision and Recall

$F1$ is the harmonic mean of precision and recall:

$$F1 = \frac{2 * P * R}{(P + R)} \tag{25}$$

Can combine P, R, $F1$ for each class:

- **Macro-average**: take the average after computing P, R, $F1$ for each class.
  - Weights each class equally. Good if all classes are equally important
- **Micro-average**: take the sum of the counts first, then compute P, R, $F1$.
  - Weights each sample equally.

4.12 Confusion Matrix

It is often helpful to visualize the performance of a classifier using aconfusion matrix (horizontal represents the predicted class, and vertical represents the actual class):

	C_1	C_2	C_3	C_4
C_1	count	count	count	count
C_2	count	count	count	count
C_3	count	count	count	count
C_4	count	count	count	count

We prefer if most of the cases fall into the diagonal entries.

5 N-Grams