Speech and Language Processing

Tim Lawson

November 16, 2023

These notes are based on Jurafsky and Martin 2023.

Contents

4 Naive Bayes, Text Classification, and Sentiment

4.1 Naive Bayes Classifiers

- Probabilistic, generative, and discriminative classifiers:
  - A probabilistic classifier outputs the probability that an instance belongs to a class.
  - Generative classifiers model the probability that a class generated an instance.
  - Discriminative classifiers learn the features that best discriminate between classes.
- Bayes' theorem, posterior and prior probabilities, and likelihoods:
  - Naive Bayes is a probabilistic classifier.
  - For a document d and classes $c \in C$, it returns the class $\hat{c}$ that has the maximum posterior probability given the document:

$$\hat{c} = \arg\max_{c \in C} P(c \mid d) \tag{4.1}$$

  - Bayes' theorem states that the posterior probability is:

$$P(c \mid d) = \frac{P(d \mid c)P(c)}{P(d)} \tag{4.2}$$

  - $P(d)$ is the prior probability of the document, which is the same for all classes. Therefore, we can return the class:

$$\hat{c} = \arg\max_{c \in C} P(d \mid c)P(c) \tag{4.3}$$

- Naive Bayes classifiers' assumptions:
  - A 'bag of words' is an unordered set of words. The 'bag of words' assumption is that the order of words in a document does not matter.
  - Naive Bayes (conditional independence) assumption.

4.2 Training the Naive Bayes Classifier

- Maximum likelihood estimate (frequency counts)
- Smoothing, unknown words, and stop words
- Example training algorithm

4.4 Optimizing for Sentiment Analysis

- For sentiment classification and other tasks, whether a token occurs in a document is more important than how many times it occurs. This is called binomial or binary naive Bayes.
- A simple way to handle negation is to prepend 'not' to tokens that occur between a negation token and the next punctuation mark. Negation can be handled more accurately with parsing.
- Sentiment lexicons are lists of tokens annotated with positive or negative sentiment. When the training set is sparse or not representative of the test set, lexicon features may generalize better.

4.5 Naive Bayes for other text classification tasks

Sets of tokens and non-linguistic features may be appropriate for other tasks. E.g., character or byte n-grams are effective for language identification. Feature selection is the process of selecting the most informative features.

4.6 Naive Bayes as a Language Model

Naive Bayes classifiers can use features that depend on multiple tokens or are non-linguistic. But naive Bayes classifiers that only use single-token features are similar to language models. Specifically, they are sets of class-specific unigram language models.

4.7 Evaluation: Precision, Recall, F-measure

- 'True' (human-defined) labels are called gold or gold-standard labels.
- A confusion matrix or contingency table represents the precision, recall, and accuracy of a classifier.
- The F-measure is one way to combine precision and recall into a single measure.
  - $F_\beta = \frac{(\beta^2+1)\cdot\text{precision}\cdot\text{recall}}{\beta^2\cdot\text{precision}+\text{recall}}$
  - $\beta < 1$ favours precision and $\beta > 1$ favours recall.
  - $\beta = 1$ equally balances precision and recall.
  - $F_{\beta=1}$ or $F_1 = \frac{2\cdot\text{precision}\cdot\text{recall}}{\text{precision}+\text{recall}}$ is the most common.
  - The F-measure is a weighted harmonic mean of precision and recall.
- Multi-class classification
  - In macroaveraging, we compute the performance for each class, then average over classes. A microaverage is dominated by the most frequent class.
  - In microaveraging, we compute the performance for all classes. A microaverage better reflects the class distribution, so it is more appropriate when performance on all classes is equally important.

4.8 Test Sets and Cross-validation

- Thus far, we have described a procedure wherein we split the data into training, test, and development sets.
- We can instead use all of the data for training and testing by using k-fold cross-validation:
  - Partition the data into k disjoint subsets (folds).
  - For each fold, train the classifier on the remaining $k-1$ folds and test it on the fold.
  - Average the test-set error rate over all folds.
- But then we can't examine the data to look for features, because we would be looking at the test set.
- Therefore, it is common to split the data into training and test sets, then use k-fold cross-validation on the training set.

4.9 Statistical Significance Testing

- How do we compare the performance of two systems A and B? The effect size $\delta(x)$ is the performance difference of A relative to B on a test set x. We want to know whether $\delta(x)$ is statistically significant.
- The null hypothesis H_0 is that $\delta(x) \leq 0$. We want to reject the null hypothesis to support the hypothesis that A is better than B.
- Let X be a random variable over all test sets. The p-value is the probability $P(\delta(X) \geq \delta(x) \mid H_0)$. A result is statistically significant if the p-value is below a threshold.
- Parametric tests like t-tests or ANOVAs make assumptions about the distributions of the test statistic. In NLP, we typically use non-parametric tests based on sampling:
  - Approximate randomization
  - The bootstrap test
- Paired tests compare two sets of aligned observations, e.g., the performance of two systems on a test set.

4.9.1 The Paired Bootstrap Test

- Bootstrapping is repeatedly sampling with replacement from a set of observations. Intuitively, we can create many virtual test sets from an observed test set by sampling from it.
- The test set is biased by $\delta(x)$ towards A (the expected value of $\delta(X)$ for the bootstrapped test sets is $\delta(x)$).

$$\text{p-value}(x) = \frac{1}{b}\sum_{i=1}^{b} \mathbb{I}(\delta(x_i) \geq 2\delta(x)) \tag{4.4}$$

4.10 Avoiding Harms in Classification

- Representational harms affect demographic groups, e.g., by perpetuating negative sterotypes.
- Another kind of harm is censorship, which may disproportionately affect minority groups, e.g., in toxicity detection.
- A model card documents its data sources, intended use, performance across demographic groups, etc.

5 Logistic Regression

Generative and discriminative classifiers

- A *generative* model uses a likelihood term $P(d \mid c)$. It expresses how to generate the data d from the class c.
- A *discriminative* model uses a posterior term $P(c \mid d)$. It expresses how to discriminate between classes c given the data d.

Components of a probabilistic machine-learning classifier

- A *feature representation* of the input $\vec{x}$.
- A *classification function* $f(\vec{x}) = \hat{y}$, e.g., sigmoid and softmax.
- An *objective function*, e.g., the cross-entropy loss function.
- An *optimization algorithm*, e.g., stochastic gradient descent.

5.1 The sigmoid function

Logistic regression learns a vector of *weights* and a *bias* term or intercept:

$$z = \left(\sum_{i=1}^{n} w_i x_i\right) + b \equiv \vec{w} \cdot \vec{x} + b \tag{5.1}$$

The *sigmoid* or logistic function $\sigma : \mathbb{R} \to [0, 1]$ is:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{5.2}$$

It is differentiable and squashes outliers towards 0 or 1.

For binary classification, we require that $P(y = 0) + P(y = 1) = 1$:

$$P(y = 1 \mid \vec{x}) = \sigma(\vec{w} \cdot \vec{x} + b) \tag{5.3}$$

$$P(y = 0 \mid \vec{x}) = 1 - \sigma(\vec{w} \cdot \vec{x} + b) = \sigma(-\vec{w} \cdot \vec{x} - b) \tag{5.4}$$

5.2 Classification with logistic regression

For binary classification, we call 0.5 the *decision boundary* of $P(y \mid \vec{x})$:

$$\text{decision}(\vec{x}) = \begin{cases} 1 & \text{if } P(y = 1 \mid \vec{x}) > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{5.5}$$

5.2.2 Other classification tasks and features

For some tasks, it helps to design combinations of features or *feature interactions*. These can be created automatically from *feature templates*. *Representation learning* tries to learn features automatically.

Scaling input features It is common to *standardize* features to have zero mean and unit variance. This transformation is called the *z-score*.

$$x'_{ij} = \frac{x_{ij} - \mu_i}{\sigma_i} \text{ where } \mu_i = \frac{1}{N}\sum_{j=1}^{N} x_{ij} \text{ and } \sigma_i = \sqrt{\frac{1}{N}\sum_{j=1}^{N}(x_{ij} - \mu_i)^2} \tag{5.6}$$

Alternatively, we can *normalize* features by $f : \mathbb{R} \to [0, 1]$:

$$x'_{ij} = \frac{x_{ij} - \min(x_i)}{\max(x_i) - \min(x_i)} \tag{5.7}$$

5.2.3 Processing many examples at once

Let $\{\vec{x}_j \mid j \in 1..N\}$ be a set of input feature vectors and $X = (x_{ij})$ a matrix where row i is the feature vector $\vec{x}_i$. Then the output vector is:

$$\vec{y} = X\vec{w} + b\vec{1} \tag{5.8}$$

5.2.4 Choosing a classifier

Naive Bayes assumes conditional independence, i.e., treats correlated features as independent. Logistic regression properly handles correlated features, so it generally works better on large datasets or documents. However, naive Bayes is easy to implement and fast to train, because it has no optimization step.

5.3 Multinomial logistic regression

Multinomial ($k > 2$) logistic regression is sometimes called softmax regression. It is an example of *hard* classification, i.e., it assigns a single class to each instance. A *one-hot vector* $\vec{y}$ has $y_i = 1$ and $y_j = 0 \; \forall \; i \neq j$.

5.3.1 Softmax

The *softmax* function is a generalisation of the sigmoid function to k classes:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} : i \in 1..k \tag{5.9}$$

5.3.2 Applying softmax in logistic regression

The probability that an input feature vector $\vec{x}$ belongs to class y_i is:

$$P(y_i = 1 \mid \vec{x}) = \text{softmax}(\vec{w}_i \cdot \vec{x} + b_i) = \frac{e^{\vec{w}_i \cdot \vec{x} + b_i}}{\sum_{j=1}^{k} e^{\vec{w}_j \cdot \vec{x} + b_j}} : i \in 1..k \tag{5.10}$$

The vector of output probabilities for the k classes is:

$$\hat{\vec{y}} = \text{softmax}(W\vec{x} + \vec{b}) \tag{5.11}$$

5.3.3 Features in multinomial logistic regression

In multinomial logistic regression, there is a weight vector and bias for each of the k classes. The weight vector $\vec{w}$ depends on the input feature vector and the output class, so it is sometimes written $f(x, y)$.

5.4 Learning in logistic regression

The learning task to learn the weight vector $\vec{w}$ and bias b that make the vector of output probabilities $\vec{\hat{y}}$ most similar to the true output $\vec{y}$ for the training data. Typically, the *distance* between them is measured, which is called the *loss* or cost function. The *cross-entropy* loss is commonly used for logistic regression and neural networks. The loss function is minimised by an optimisation algorithm. *Gradient descent*, e.g., stochastic gradient descent, is commonly used.

5.5 The cross-entropy loss function

A loss function that prefers that the true outputs are more likely is an example of *conditional maximum likelihood estimation*. It maximises the log probability of the true outputs given the input feature vectors.

The cross-entropy loss is:

$$\mathcal{L}(\vec{\hat{y}}, \vec{y}) = -\sum_{i=1}^{k} y_i \log \hat{y}_i \tag{5.12}$$

Derivation for binary regression For $k = 2$ classes, the probability that the output label is correct $P(y \mid x)$ is a Bernoulli distribution (it has two discrete outcomes):

$$P(y \mid x) = \hat{y}^y (1 - \hat{y})^{1-y} \tag{5.13}$$

The logarithm is a monotonic function. Therefore, instead of maximising the probability, we can maximise the log probability (or minimise the negative log probability):

$$-\log P(y \mid x) = -y \log \hat{y} - (1 - y) \log(1 - \hat{y}) \tag{5.14}$$

This is equivalent to 5.12 for $k = 2$.

5.6 Gradient descent

An optimisation algorithm finds the weights θ that minimise the loss function. In logistic regression, $\theta = (\vec{w}, b)$. For n instances, the weights that minimise the average loss are:

$$\vec{\hat{\theta}} = \arg\min_{\vec{\theta}} \frac{1}{n} \sum_{i=1}^{n} L(f(\vec{x}^{(i)}, \vec{\theta}), \vec{y}^{(i)}) \tag{5.15}$$

Gradient descent finds the minimum of a function by incrementing its parameters in the opposite direction in parameter space to the direction with the largest gradient. The increment is weighted by a *learning rate* η. In logistic regression, the loss function is convex (has at most one minimum) and gradient descent is guaranteed to find the global minimum.

The equation to update the weights is:

$$\vec{\theta}' = \vec{\theta} - \eta \nabla_{\vec{\theta}} L(f(x, \vec{\theta}), y) \tag{5.16}$$

5.6.1 The gradient for logistic regression

For binary logistic regression, the cross-entropy loss function is:

$$\mathcal{L}(\hat{y}, y) = -y \log \hat{y} - (1-y) \log(1-\hat{y}) \quad \text{where} \quad \hat{y} = \sigma(\vec{w} \cdot \vec{x} + b) \tag{5.17}$$

Its derivative with respect to w_i is:

$$\frac{\partial \mathcal{L}}{\partial w_j} = -y \frac{\partial \log \hat{y}}{\partial w_j} - (1-y) \frac{\partial \log(1-\hat{y})}{\partial w_j} \tag{5.18}$$

$$= -\frac{y}{\hat{y}} \frac{\partial \hat{y}}{\partial w_j} - \frac{1-y}{1-\hat{y}} \frac{\partial (1-\hat{y})}{\partial w_j} \tag{5.19}$$

$$= -\left(\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}} \right) \frac{\partial \hat{y}}{\partial w_j} \tag{5.20}$$

The derivative of the sigmoid function is:

$$\frac{\partial \sigma(z)}{\partial z} = \sigma(z)(1-\sigma(z)) \tag{5.21}$$

Therefore:

$$\frac{\partial \mathcal{L}}{\partial w_j} = -(y-\hat{y}) \frac{\partial (\vec{w} \cdot \vec{x} + b)}{\partial w_j} \tag{5.22}$$

$$= (\hat{y}-y) x_j \tag{5.23}$$

I.e., the gradient with respect to a weight w_j is the difference between the true y and the estimated $\hat{y}$ multiplied by the input feature x_j.

5.6.2 The stochastic gradient descent algorithm

The learning rate η is a hyperparameter. It is common to start with a higher learning rate and slowly decrease it.

5.6.4 Mini-batch training

6 Vector Semantics and Embeddings

- Distributional hypothesis (quotes)
- Word embeddings, static/contextualized
- Representation learning

6.1 Lexical Semantics

- Lemmas and senses, word sense disambiguation
- Synonymy, propositional meaning, principle of contrast
- Similarity and relatnedness (association)
- Semantic fields, topic models, LDA ...
- Semantic frames
- Connotations (affective meanings) and sentiment

6.2 Vector Semantics

- Combining vector representations and distributional semantics
- Embeddings (dense, e.g., word2vec) and vectors (sparse, e.g., tf-idf)

6.3 Words and Vectors

- Cooccurrence matrix
- E.g., term-document matrix
- Dimensionality
- Information retrieval
- Row and column vectors
- E.g., term-term matrix

6.4 Cosine for measuring similarity

- Dot/inner product and length
- Normalized dot product (cosine of angle)
- Cosine similarity

TF-IDF, (P)PMI, applications.

6.5 Word2vec

- Dense, short vectors
- Skip-gram with negative sampling (SGNS)
- Self-supervision

6.5.1 The classifier

- Logistic/sigmoid functon
- Independence assumption

6.5.2 Learning skip-gram embeddings

- Random initial embedding vectors and iteration
- Negative sampling and noise words
- Maximize similarity of positive target-context pairs and minimize similarity of negative pairs
- Stochastic gradient-descent
- Proof of derivatives and update equations
- Target and context embeddings
- Context window size

6.5.3 Other kinds of static embeddings

- fasttext and subwords
- GloVe (global vectors)

6.6 Visualizing Embeddings

- Most similar words
- Clustering
- Dimensionality reduction, e.g., t-SNE

6.7 Semantic properties of embeddings

- First- and second-order cooccurrence (syntagmatic and paradigmatic association)
- Analogy and relational similarity, parallelogram model
- Historical semantics

6.8 Bias and embeddings

- Allocational and representational harm
- Tendency to amplify bias
- Debiasing

7 Neural Networks

7.1 Units

Definition 7.1 (Computational unit)**.** *A computational unit is a function that takes an input vector $\vec{x}$, a weight vector $\vec{x}$, a scalar bias b, and a non-linear function f and produces an activation y:*

$$y = f(\vec{x} \cdot \vec{x} + b) \tag{7.1}$$

Non-linear functions Examples of non-linear functions are:

- the sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$;
- the hyperbolic tangent $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$; and
- the rectified linear unit (ReLU) $\max(0, z)$.

tanh is differentiable and maps outliers towards the mean, whereas ReLU is non-differentiable. For large z values, σ and tanh produce saturated y values, i.e., values that are nearly one have gradients close to zero, whereas ReLU has gradient one for positive z values. Gradients that are close to zero cannot be used for training (the *vanishing gradient* problem).

7.2 The XOR problem

A single computational unit (e.g., a *perceptron*) cannot compute simple functions of its inputs (e.g., XOR). This is because a perceptron is a linear classifier and XOR is not a linearly separable function. However, a layered network of units can compute functions like XOR.

7.3 Feedforward neural networks

A *feedforward* network is a layered network of units that are connected with no cycles. Networks with cycles are called *recurrent* networks. Feedforward networks are sometimes called *multi-layer perceptrons* but this is a misnomer unless the units are perceptrons.

A simple feedforward network has input units, hidden units, and output units. In a *fully-connected* layer of units, each unit takes as its input the outputs of all units in the previous layer.

Hidden layers A hidden layer is a set of computational units (definition 7.1) that takes an input vector $\vec{x}$, a weight matrix W, a bias vector $\vec{b}$, and a non-linear function f and produces an activation vector $\vec{h}$:

$$\vec{h} = f(W\vec{x} + \vec{b}) \tag{7.2}$$

The function f is applied element-wise to the vector $W\vec{x} + \vec{b}$.

Output layers In a feedforward network with a single hidden layer, the output layer is a function that takes an activation vector $\vec{h}$ and a weight matrix U and produces an intermediate output vector $\vec{z}$:

$$\vec{z} = U\vec{h} \tag{7.3}$$

Normalization functions A *normalization* function converts an activation vector into a vector that represents a probability distribution, e.g., softmax:

$$\vec{y} = \text{softmax}(\vec{z}) = \frac{e^{\vec{z}}}{\sum_{i=1}^{n} e^{z_i}} \tag{7.4}$$

Dimensionality The elements of feedforward neural network with a single fully-connected hidden layer have the following dimensions:

- $\vec{x} \in \mathbb{R}^{n_0}$ is a column vector $n_0 \times 1$;
- $\vec{h}, \vec{b}, \vec{z} \in \mathbb{R}^{n_1}$ are column vectors $n_1 \times 1$;
- $W \in \mathbb{R}^{n_1 \times n_0}$ is a matrix $n_1 \times n_0$;
- $U \in \mathbb{R}^{n_2 \times n_1}$ is a matrix $n_2 \times n_1$; and
- $\vec{y} \in \mathbb{R}^{n_2}$ is a column vector $n_2 \times 1$.

Non-linear activation functions If the activation function f is linear, then a multi-layer feedforward network is equivalent to a single-layer network (with a different weight matrix). Therefore, we use non-linear activation functions.

Replacing the bias vector Generally, we replace the bias vector with an additional unit in each layer whose activation is always one. The weights associated with this unit are the bias value b:

$$\vec{x} \in \mathbb{R}^{n_0} \rightarrow \vec{x} \in \mathbb{R}^{n_0+1},\ x_0 = 1$$
$$W \in \mathbb{R}^{n_1 \times n_0} \rightarrow W \in \mathbb{R}^{n_1 \times (n_0+1)},\ W_{i0} = b_i\ \forall\ i \in 0..n_0$$

7.4 Classification

A classifier could use hand-crafted features but most applications learn features from the data by representing words by embeddings. An input is generally represented by applying a *pooling* function to the embeddings of its words, e.g., the mean or element-wise maximum.

For a two-layer classifier, the equations to predict the output of a set of test instances are:

- n_i is the number of test instances;
- n_j is the dimensionality of the instance space;
- n_k is the number of nodes in the hidden layer;
- n_l is the number of classes;
- $X \in \mathbb{R}^{n_i \times n_j}$ is the set of test instances;
- $W \in \mathbb{R}^{n_k \times n_j}$ is the weight matrix;
- $B \in \mathbb{R}^{n_i \times n_k}$ is the bias matrix;

- $H = \sigma(XW^T + B) \in \mathbb{R}^{n_i \times n_k}$ is the hidden-layer activation matrix;
- $U \in \mathbb{R}^{n_l \times n_k}$ is the hidden-layer weight matrix;
- $Z = HU^T \in \mathbb{R}^{n_i \times n_l}$ is the intermediate output matrix; and
- $\hat{Y} = \text{softmax}(Z) \in \mathbb{R}^{n_i \times n_l}$ is the output matrix.

The use of an input representation is called *pretraining*. It is possible to train the classifier and the input representation jointly as part of an NLP task.

7.5 Language modelling

The task of *language modelling* is to predict upcoming words from prior context. Neural language modelling is an important task in itself and a precursor to many others. Generally, modern language models use architectures like recurrent neural networks or transformers.

7.5.1 Inference

Forward inference is the task of producing a probability distribution over the possible outputs, given an input. For language models, the inputs and outputs are words. A *one-hot vector* has a single element equal to 1 and the rest 0.

The equations to predict the output are:

- n_i is the context window size;
- n_j is the number of words in the vocabulary;
- n_k is the dimensionality of the embeddings;
- n_l is the number of nodes in the hidden layer;
- $\{x_{t-i} \mid i = 1..n_i\} \in \mathbb{R}^{n_j}$ are one-hot word vectors;
- $E \in \mathbb{R}^{n_k \times n_j}$ is the embedding matrix;
- $\vec{e}_i = E\vec{x}_i \in \mathbb{R}^{n_k}$ is the embedding of word x_i;
- $\vec{e} = [\vec{e}_1; \ldots; \vec{e}_{n_i}] \in \mathbb{R}^{n_i n_k}$ are the concatenated embeddings;
- $W \in \mathbb{R}^{n_l \times n_i n_k}$ is the weight matrix;
- $\vec{b} \in \mathbb{R}^{n_l}$ is the bias vector;
- $\vec{h} = \sigma(W\vec{e} + \vec{b}) \in \mathbb{R}^{n_l}$ is the hidden-layer activation vector;
- $U \in \mathbb{R}^{n_j \times n_l}$ is the hidden-layer weight matrix;
- $\vec{z} = U\vec{h} \in \mathbb{R}^{n_j}$ is the intermediate output vector; and
- $\vec{\hat{y}} = \text{softmax}(\vec{z}) \in \mathbb{R}^{n_j}$ is the output vector.

7.6 Training neural networks

A feedforward neural network is an example of a supervised machine learning model. A neural network is trained with:

- a loss function, e.g., the cross-entropy loss (section 7.6.1); and
- an optimisation algorithm, e.g., gradient descent.

For logistic regression, the derivative of the loss function can be computed directly (section 7.6.2). For neural networks, *backpropagation* (section 7.6.3) is necessary.

7.6.1 Cross-entropy loss

Definition 7.2 (Cross-entropy loss)**.** *Let $\vec{y}, \hat{\vec{y}} \in \mathbb{R}^k$ be the one-hot true output vector and predicted output vector, respectively. The cross-entropy loss of an instance with true class j is:*

$$L(\vec{y}, \hat{\vec{y}}) = -\sum_{i=1}^{k} y_i \log \hat{y}_i \tag{7.5}$$

$$= -\log \hat{y}_j \tag{7.6}$$

7.6.2 Gradient

For a neural network with a single hidden layer and sigmoid activation function, i.e., logistic regression, the derivative of the cross-entropy loss with respect to the weight w_i is:

$$\begin{aligned}\frac{\partial L(\vec{y}, \hat{\vec{y}})}{\partial w_i} &= (\hat{y} - y)\vec{x}_i \\ &= (\sigma(\vec{w} \cdot \vec{x} + b) - y)\vec{x}_i\end{aligned}$$

With a softmax activation function, i.e., multinomial logistic regression, the derivative of the cross-entropy loss with respect to the weight w_{ij} is:

$$\begin{aligned}\frac{\partial L(\vec{y}, \hat{\vec{y}})}{\partial w_{ij}} &= -(\vec{y}_i - \hat{\vec{y}}_i)\vec{x}_j \\ &= -\left(\vec{y}_i - \frac{\exp(\vec{w}_i \cdot \vec{x} + b_i)}{\sum_{k=1}^{l} \exp(\vec{w}_k \cdot \vec{x} + b_k)}\right)\vec{x}_j\end{aligned}$$

7.6.3 Backward differentiation

For a neural network with multiple hidden layers, we must compute the derivative with respect to the weights in each layer. This is achieved by backpropagation or backprop, which is a form of *backward differentiation* on the computational graph of the network.

Example TODO

7.6.4 Learning

For logistic regression, we can initialize the weights and biases to zero. For neural networks, we must initialize the weights and biases to small random numbers and normalize the inputs (zero mean and unit variance).

Regularization, e.g., *dropout*, is used to prevent overfitting. It is also important to tune *hyperparameters*, e.g., the learning rate, the mini-batch size, the model architecture, and the regularization.

8 Sequence Labelling

Parts of speech (POS) include nouns, verbs, pronounds, adverbs, conjunctions, participles, and articles. *Part-of-speech (POS) tagging* is the task of labelling each word in a sequence with a part-of-speech tag.

Named entities are referred to with proper names and include people, locations, and organisations. *Named-entity recognition (NER)* is the task of labelling each word in a sequence with a named-entity tag.

8.1 Word classes

8.2 Part-of-speech tagging

Part-of-speech tagging is a *disambiguation* task: words are ambiguous because they have multiple possible parts of speech. The task is to resolve these ambiguities by choosing the proper tags. Part-of-speech tagging algorithms are highly accurate.

8.3 Named-entity recognition

8.4 Hidden Markov models

Hidden Markov models (HMMs) are based on Markov chains. They are probabilistic: given a sequence of observations, they compute the probability distribution over possible sequences of labels and choose the best one.

Markov chains

Definition 8.1 (Markov assumption)**.** *Given a sequence of states $U = u_1 \dots u_{i-1}$, the probability of a state u_i depends only on the previous state u_{i-1}:*

$$P(u_i = s_j \mid u_1 \dots u_{i-1}) = P(u_i = s_j \mid u_{i-1}) \tag{8.1}$$

Definition 8.2 (Markov chain)**.** *A Markov chain is a tuple (S, T, P) where:*

- *$S = \{s_i \mid i = 1..n_s\}$ is a set of n_s states;*
- *$T \in \mathbb{R}^{n_s \times n_s}$ is a transition probability matrix where t_{ij} is the probability of transitioning from s_i to s_j, i.e., $\sum_{j=1}^{n_s} t_{ij} = 1 \ \forall \ i \in 1..n_s$;*
- *$P_0 = \{p_i \mid i = 1..n_s\}$ is the initial probability distribution where p_i is the probability that the first state is s_i, i.e., $\sum_{i=1}^{n_s} p_i = 1$.*

Hidden Markov models A HMM represents both observed and *hidden* events, which are not observed directly (like part-of-speech tags).

Definition 8.3 (Hidden Markov model)**.** *A hidden Markov model is a Markov chain (definition 8.2) where additionally:*

- *$W = w_1 \dots w_{n_w}$ is a sequence of n_w observations, each of which is taken from a vocabulary $V = \{v_i \mid i \in 1..n_v\}$; and*
- *$p_{ij} = P(w_j \mid s_i)$ is an observation likelihood or emission probability, i.e., the probability of observing w_j given s_i.*

As well as the Markov assumption (definition 8.1), a HMM assumes *output independence*: that the probability of an observation depends only on the current state, i.e., $P(w_i = v_j \mid u_1 u_2 \dots u_i) = P(w_i = v_j \mid u_i)$.

8.4.1 HMM tagging and decoding

The task of determining the most likely sequence of hidden states U given a sequence of observations W is called *decoding*. For part-of-speech tagging, this task is (applying Bayes' theorem):

$$\hat{u}_1 \hat{u}_2 \dots \hat{u}_{n_u} = \underset{u_1 u_2 \dots u_{n_u}}{\arg\max}\ P(u_1 u_2 \dots u_{n_u} \mid w_1 w_2 \dots w_{n_w}) \tag{8.2}$$

$$= \underset{u_1 u_2 \dots u_{n_u}}{\arg\max}\ \frac{P(w_1 w_2 \dots w_{n_w} \mid u_1 u_2 \dots u_{n_u}) P(u_1 u_2 \dots u_{n_u})}{P(w_1 w_2 \dots w_{n_w})} \tag{8.3}$$

$$= \underset{u_1 u_2 \dots u_{n_u}}{\arg\max}\ P(w_1 w_2 \dots w_{n_w} \mid u_1 u_2 \dots u_{n_u}) P(u_1 u_2 \dots u_{n_u}) \tag{8.4}$$

HMM tagging makes two more simplifying assumptions:

- The probability of a word depends only on its own tag (is independent of the other words in the sequence), i.e.,

$$P(w_1 w_2 \dots w_{n_w} \mid u_1 u_2 \dots u_{n_u}) \approx \prod_{i=1}^{n_w} P(w_i \mid u_i) \tag{8.5}$$

- The *bigram* assumption: the probability of a tag depends only on the previous tag (is independent of the other tags in the sequence), i.e.,

$$P(u_1 u_2 \dots u_{n_u}) \approx \prod_{i=1}^{n_u} P(u_i \mid u_{i-1}) \tag{8.6}$$

Substituting equations 8.5 and 8.6 into equation 8.4 gives:

$$\hat{u}_1 \hat{u}_2 \dots \hat{u}_{n_u} = \underset{u_1 u_2 \dots u_{n_u}}{\arg\max} \prod_{i=1}^{n_w} P(w_i \mid u_i) P(u_i \mid u_{i-1}) \tag{8.7}$$

where $P(w_i \mid u_i)$ is the *emission probability* and $P(u_i \mid u_{i-1})$ is the *transition probability* from definitions 8.3 and 8.2, respectively.

8.4.2 The Viterbi algorithm

The Viterbi algorithm is a dynamic programming algorithm that computes the most likely sequence of labels.

A matrix $X \in \mathbb{R}^{n_w \times n_s}$ is initialised with a column for each observation and a row for each state. An element x_{ij} is the probability that the HMM is in state s_j after observing $w_1 \dots w_i$.

The elements x_{ij} are computed by recursively taking the most likely paths that lead to their observations given the HMM λ.

$$x_{ij} = \max_{u_1 \dots u_{i-1}} P(u_1 \dots u_{i-1},\ w_1 \dots w_i,\ u_i = s_j \mid \lambda) \tag{8.8}$$

The most likely path to an element x_{ij} is the most likely path to an element for the *previous* observation $x_{i-1,k}$ whose state s_k maximises the the probability of transitioning into state s_j *and* observing w_i, i.e.:

$$x_{ij} = \max_{k \in 1..n_s} x_{i-1,k}\ t_{kj}\ p_{ji} \tag{8.9}$$

where:

- $x_{i-1,k}$ is the probability of s_k after observing $w_1 \dots w_{i-1}$;
- t_{kj} is the probability of transitioning from s_k to s_j; and
- p_{ji} is the probability of observing w_i given s_j.

8.5 Conditional random fields

8.6 Evaluation

Part-of-speech tagging is evaluated by *accuracy*. Named-entity recognition is evaluated by *recall*, *precision*, and F_1 *measure*. The statistical significance of a difference between the F_1 measures of two named-entity recognition systems is evaluated by the *paired bootstrap test* or similar. The segmentation component of named-entity recognition complicates evaluation.

9 RNNs and LSTMs

Language is inherently temporal. Some language-processing algorithms are temporal, e.g., the Viterbi algorithm (section 8.4.2), but others access the entire input sequence at once, e.g., bag-of-words models. Recurrent neural networks, e.g., LSTMs, represent prior context differently to n-gram models or feedforward neural networks: they do not require a fixed window size.

9.1 Recurrent Neural Networks

A *recurrent* neural network (RNN) has a cycle in its connections. Generally, RNNs are difficult to train but constrained architectures like *Elman* or *simple* recurrent networks have proved successful for language modelling tasks.

9.1.1 Inference

For a simple RNN with one hidden layer, inference at time t is computed by:

$$\vec{h}^{(t)} = f(W\vec{x}^{(t)} + U\vec{h}^{(t-1)}) \tag{9.1}$$

$$\vec{y}^{(t)} = g(V\vec{h}^{(t)}) \tag{9.2}$$

- $\vec{x}^{(t)} \in \mathbb{R}^{n_x}$ is the input vector at time t;
- $\vec{h}^{(t)} \in \mathbb{R}^{n_h}$ is the hidden-layer activation vector at time t;
- $\vec{y}^{(t)} \in \mathbb{R}^{n_y}$ is the output vector at time t;
- $W \in \mathbb{R}^{n_h \times n_x}$ is the weight matrix between the input and hidden layers;
- $U \in \mathbb{R}^{n_h \times n_h}$ is the weight matrix between the hidden layer and itself;
- $V \in \mathbb{R}^{n_y \times n_h}$ is the weight matrix between the hidden and output layers;
- $f : \mathbb{R}^{n_h} \to \mathbb{R}^{n_h}$ is the hidden-layer activation function; and
- $g : \mathbb{R}^{n_y} \to \mathbb{R}^{n_y}$ is the output-layer activation function.

The computation is incremental because the hidden-layer activation vector at time t depends on the hidden-layer activation vector at time $t-1$. The network is effectively *unrolled* over time, i.e., the vectors are copied for each time step, but the weight matrices are shared across time steps.

9.1.2 Training

Historically, RNNs were trained with *backpropagation through time*, a two-pass algorithm. But generally, a feedforward network is generated from the structure of an RNN for a given input sequence. Long input sequences can be divided into fixed-length subsequences.

9.2 Language modelling

9.2.1 Inference

For a simple recurrent language model with one hidden layer, inference at time t is computed by (section 9.1.1):

$$\vec{e}^{(t)} = E\vec{x}^{(t)} \tag{9.3}$$

$$\vec{h}^{(t)} = f(W\vec{e}^{(t)} + U\vec{h}^{(t-1)}) \tag{9.4}$$

$$\vec{y}^{(t)} = \text{softmax}(V\vec{h}^{(t)}) \tag{9.5}$$

- $X = \begin{bmatrix}\vec{x}_1 & \dots & \vec{x}_n\end{bmatrix} \in \mathbb{R}^{n_x \times n}$ is the input matrix, i.e., a sequence of n word embeddings represented by one-hot vectors of size n_x (the vocabulary size);
- $\vec{e}^{(t)} \in \mathbb{R}^{n_h}$ is the input embedding vector at time t; and
- $E \in \mathbb{R}^{n_h \times n_x}$ is the input embedding matrix.

9.2.2 Predicted probabilities

The output vector $\vec{y}^{(t)}$ is the predicted probability distribution over the vocabulary. The predicted probability that the next word is the i-th word is $y_i^{(t)}$, i.e., the i-th component of $\vec{y}^{(t)}$:

$$P(w_{t+1} = i \mid w_1 \dots w_t) = y_i^{(t)} \tag{9.6}$$

The predicted probability of a sequence of length n is the product of the probabilities of the words in the sequence:

$$P(w_1 \dots w_n) = \prod_{i=1}^{n} y_{w_i}^{(i)} \tag{9.7}$$

9.2.3 Training

The model is trained to minimise the difference between the predicted and correct probability distributions, which is measured by the cross-entropy loss function. The correct probability distribution is represented by a one-hot vector that is 1 for the next true word and 0 otherwise. Therefore, the cross-entropy loss is solely determined by the predicted probability of the next true word:

$$\mathcal{L}(\hat{\vec{y}}^{(t)}, \vec{y}^{(t)}) = -\log \hat{y}_{w_{t+1}}^{(t)} \tag{9.8}$$

At each time t of the input sequence, inference is performed with the true sequence $w_1 \dots w_t$ to generate the predicted probability distribution $\hat{\vec{y}}^{(t)}$ and the cross-entropy loss is computed from the predicted probability of the next true word $\hat{y}_{w_{t+1}}^{(t)}$. The procedure is repeated at time $t+1$, ignoring the predicted word at time t.

9.2.4 Weight tying

The output vector $\vec{y}^{(t)} \in \mathbb{R}^{n_x}$ of a language model has the same dimensions as the input vector. Therefore, the input embedding matrix $E \in \mathbb{R}^{n_h \times n_x}$ and the weight matrix between the hidden and output layers $V \in \mathbb{R}^{n_x \times n_h}$ are similar: both can be interpreted as word embeddings.

Weight tying is the practice of setting $V = E^T$ (section 9.2.1), which reduces the number of parameters of the model and improves its perplexity:

$$\vec{e}^{(t)} = E\vec{x}^{(t)} \tag{9.9}$$

$$\vec{h}^{(t)} = f(W\vec{e}^{(t)} + U\vec{h}^{(t-1)}) \tag{9.10}$$

$$\vec{y}^{(t)} = \text{softmax}(E^T\vec{h}^{(t)}) \tag{9.11}$$

9.3 Other tasks

9.3.1 Sequence labelling

In an RNN for sequence labelling (section 8), the input vectors are pre-trained embeddings and the output vectors are probability distributions over the labels. The RNN infers the most likely label at each time step and is trained as in section 9.2.3.

9.3.2 Sequence classification

In an RNN for sequence classification, the hidden layer for the last time step is treated as a compressed representation of the entire sequence. This representation can be input to a feedforward network for classification (section 7.4).

Alternatively, a *pooling* function can be used to represent the entire sequence by an aggregate of the hidden layers at each time step, e.g., the element-wise mean or maximum. In this instance, the RNN is trained *end-to-end*: the loss function is based only on the classification task.

9.3.3 Text generation

The approach of incrementally generating text from a language model by repeatedly sampling the next word from the predicted probability distribution conditioned on the previous samples is called *autoregressive* or causal language-model generation. Task-appropriate context can be provided by conditioning the model on a prefix as well as the previous samples.

9.4 Stacked RNNs

In sections 9.1 to 9.3, the input vectors are embeddings and the output vectors are probability distributions that predict words or labels. *Stacked* RNNs consist of multiple networks, where the output vectors of one network are the input vectors of the next.

Generally, stacked RNNs outperform single-layer networks. They seem to induce representations at different levels of abstraction in the different layers. However, they are more expensive to train.

9.5 Bidirectional RNNs

In sections 9.1 to 9.3, the networks use information from only the prior context (previous time steps). However, for some tasks, the entire sequence can be accessed at once, i.e., information from subsequent time steps could also be used. *Bidirectional* RNNs consist of two networks, one that processes the sequence from left to right and one that processes it from right to left.

The hidden-layer activation vectors of the two networks can be concatenated at each time step to produce a single vector that represents both contexts:

$$\vec{h}^{(t)} = \left[\vec{h}^{(t)}_{\text{left}} \quad \vec{h}^{(t)}_{\text{right}}\right] \tag{9.12}$$

Alternatively, the two vectors can be combined with a pooling function, e.g., element-wise addition or multiplication. Bidirectional RNNs have been successfully applied to sequence classification (section 9.3.2).

9.6 Encoder-decoder models

An encoder-decoder or *sequence-to-sequence* model translates an input sequence into an output of a different length. This model architecture is commonly applied to machine translation, summarisation, question answering, dialogue, etc.

The architecture has three components:

- an *encoder* takes an input sequence $\vec{x}_0 \cdots \vec{x}_n$ and generates a sequence of representations $\vec{h}^{(e)}_0 \cdots \vec{h}^{(e)}_n$;
- the sequence of representations generates a *context vector* $\vec{c}$; and
- a *decoder* takes a context vector $\vec{c}$ and generates a sequence of hidden states $\vec{h}^{(d)}_0 \cdots \vec{h}^{(d)}_n$ and an output sequence $\vec{y}_0 \cdots \vec{y}_n$.

The encoder and decoder variants of a recurrent network. The equations to compute the output y_t at time t are:

$$\begin{gathered}\vec{c} = \vec{h}^{(e)}_n, \quad \vec{h}^{(d)}_0 = \vec{c}, \quad \vec{h}^{(d)}_t = \text{RNN}(\hat{y}_{t-1}, \vec{h}^{(d)}_{t-1}, \vec{c}) \\ \vec{z}_t = f(\vec{h}^{(d)}_t), \quad \vec{y}_t = \text{softmax}(\vec{z}_t), \quad \hat{y}_t = \arg\max_{w \in \mathcal{V}} \vec{y}_t(w)\end{gathered} \tag{9.13}$$

Training Encoder-decoder models are trained end-to-end with pairs of input and output texts, e.g., aligned sentence pairs for machine translation. For inference, the decoder uses its estimated output $\hat{y}_t$ as the input to the next time step x_{t+1}, which leads the decoder to deviate from the true output. For training, it is common to use *teacher forcing*, i.e., for the decoder to use the true output y_t as the input to the next time step instead.

9.7 Attention

In a simple encoder-decoder model, the context vector $\vec{c}$ is the last hidden state of the encoder. The last hidden state may act as a *bottleneck*, i.e., it may not be a good representation of the entire input text. The *attention* mechanism is a way to allow the decoder to access information from all the hidden states of the encoder.

The context vector becomes a function of the encoder's hidden states:

$$\vec{c} = f(\vec{h}_1^{(e)}, \ldots, \vec{h}_n^{(e)}) \tag{9.14}$$

The weightings that the function applies to the hidden states 'pay attention to' the parts of the input sequence that are relevant to the output at a given time step. The decoder's hidden state is conditioned on the context vector:

$$\vec{h}_i^{(d)} = \text{RNN}(\hat{y}_{i-1},\, \vec{h}_{i-1}^{(d)},\, \vec{c}_{i-1}) \tag{9.15}$$

The relevance of the encoder states to the decoder state $\vec{h}_i^{(d)}$ is computed via a score $(\vec{h}_{i-1}^{(d)}, \vec{h}_j^{(e)})$. The simplest example of a scoring function is *dot-product* attention: $\vec{h}_{i-1}^{(d)} \cdot \vec{h}_j^{(e)}$. The scores are normalised with softmax to obtain a vector of weights:

$$\alpha_j^{(i)} = \text{softmax}(\vec{h}_{i-1}^{(d)} \cdot \vec{h}_j^{(e)}) = \frac{\exp(\vec{h}_{i-1}^{(d)} \cdot \vec{h}_j^{(e)})}{\sum_{k=1}^{n} \exp(\vec{h}_{i-1}^{(d)} \cdot \vec{h}_k^{(e)})} \tag{9.16}$$

Finally, given $\alpha_1^{(i)}, \ldots, \alpha_n^{(i)}$, the context vector is:

$$\vec{c}_i = \sum_{j=1}^{n} \alpha_j^{(i)} \vec{h}_j^{(e)} \tag{9.17}$$

A more complex scoring function has its own weights W_s:

$$\alpha_j^{(i)} = \text{softmax}(\vec{h}_{i-1}^{(d)} W_s \vec{h}_j^{(e)}) \tag{9.18}$$

The weights are learned in end-to-end training and allow the network to learn which aspects of similarity between the encoder and decoder hidden states are relevant to the task. They also allow the hidden states to have different dimensionality, unlike the dot-product scoring function.

Appendix

Gradient descent for RNNs

Recall that:

$$\vec{e}^{\,(t)} = E\vec{x}^{\,(t)}, \quad \vec{h}^{\,(t)} = \text{softmax}(W\vec{e}^{\,(t)} + U\vec{h}^{\,(t-1)}), \quad \vec{\hat{y}}^{\,(t)} = \text{softmax}(E^T\vec{h}^{\,(t)})$$

$$\mathcal{L}(\vec{\hat{y}}^{\,(t)}, \vec{y}^{\,(t)}) = -\log \hat{y}^{(t)}_{w_{t+1}}$$

- $\vec{x}^{\,(t)} \in \mathbb{R}^{n_x}$ is the input vector at time t;
- $\vec{e}^{\,(t)} \in \mathbb{R}^{n_h}$ is the input embedding vector at time t; and
- $\vec{h}^{\,(t)} \in \mathbb{R}^{n_h}$ is the hidden-layer activation vector at time t;
- $\vec{\hat{y}}^{\,(t)} \in \mathbb{R}^{n_y}$ is the output vector at time t;
- $E \in \mathbb{R}^{n_h \times n_x}$ is the embedding matrix;
- $W \in \mathbb{R}^{n_h \times n_x}$ is the weight matrix between the input and hidden layers;
- $U \in \mathbb{R}^{n_h \times n_h}$ is the weight matrix between the hidden layer and itself;
- $\text{softmax} : \mathbb{R}^{n_h} \to \mathbb{R}^{n_h}$ is the hidden-layer activation function; and
- $\text{softmax} : \mathbb{R}^{n_y} \to \mathbb{R}^{n_y}$ is the output-layer activation function.

In index notation:

$$e^{(t)}_i = E_{ij}x^{(t)}_j, \quad h^{(t)}_i = \text{softmax}(W_{ij}e^{(t)}_j + U_{ij}h^{(t-1)}_j), \quad \hat{y}^{(t)}_i = \text{softmax}(h^{(t)}_j E_{ji})$$

For an input sequence of length three:

$$e^{(0)}_i = E_{ij}x^{(0)}_j, \quad h^{(0)}_i = \text{softmax}(W_{ij}e^{(0)}_j), \qquad \hat{y}^{(0)}_i = \text{softmax}(h^{(0)}_j E_{ji})$$
$$e^{(1)}_i = E_{ij}x^{(1)}_j, \quad h^{(1)}_i = \text{softmax}(W_{ij}e^{(1)}_j + U_{ij}h^{(0)}_j), \quad \hat{y}^{(1)}_i = \text{softmax}(h^{(1)}_j E_{ji})$$
$$e^{(2)}_i = E_{ij}x^{(2)}_j, \quad h^{(2)}_i = \text{softmax}(W_{ij}e^{(2)}_j + U_{ij}h^{(1)}_j), \quad \hat{y}^{(2)}_i = \text{softmax}(h^{(2)}_j E_{ji})$$

$$\mathcal{L} = \sum_t \mathcal{L}(\vec{\hat{y}}^{\,(t)}, \vec{y}^{\,(t)}) = -\log \hat{y}^{(0)}_{w_1} - \log \hat{y}^{(1)}_{w_2}$$

References

Jurafsky, Dan and James H. Martin (2023). *Speech and Language Processing*. URL: https://web.stanford.edu/~jurafsky/slp3/ (visited on 01/25/2023).