Advanced Methods in Text Analytics
Exercise 1: Machine Learning Basics
Solutions

Daniel Ruffinelli

FSS 2025

1 Machine Learning Basics

(a) (i) $\boldsymbol{x}$ is an input vector, typically representing a data point in some dataset. These vectors are often referred to as feature vectors, and their corresponding vector space $\mathbb{R}^n$ as a feature space. These names are derived from each $\boldsymbol{x}_i$ being referred to as a *feature*, which models use as input during training and inference. Thus, a feature vector is the set of features we use to represent our datapoints when modeling our task of choice.

(ii) The output of our function, used for inference or to make predictions with the model.

(iii) This is read as "function $\boldsymbol{h}$ of $\boldsymbol{x}$ *given* parameters $\boldsymbol{\theta}$".

(iv) n can take any value, typically in $\mathbb{N}$. This value is typically a design decision, e.g. the number of features we manually crafted, the number of PCA components we use for dimensionality reduction, or the size of each hidden layer in a feed-forward neural network (more on this in future lectures/tutorials).
m can also take any value in $\mathbb{N}$, i.e. $\boldsymbol{y} \in \mathbb{R}^d$, where d is not necessarily equal to n. This is less a design decision and more a property of the task we are modeling. E.g. for binary classification, we typically have $m = 1$, whereas for a classification problem with k classes, $m = k$.

(b) *Classification* when output is categorical, e.g. part-of-speech (POS) tagging or next-word prediction. *Regression* when the output is a real number, e.g. in automated essay scoring.

(c) Learning refers to the process of estimating the parameters $\boldsymbol{\theta}$ of function f in order to solve an optimization problem, usually the minimization of some training objective. The goal is not to solve this optimization problem as best as possible, but to perform well on unseen data, i.e. to perform a task well *generally*.

(d) Parameters that we do not learn, but manually set. These are often not parameters of the model, but they related to the training setting. For example, using a bias or not with linear classifiers, the choice of a loss function, and even the choice of a model, can all be seen as hyperparameters. We typically find useful values for hyperparameters by conducting grid search or random search, although more involved methods exist.

(e) This related to the goal of machine learning, which is for models to generalize well to *unseen* data. To that end, datasets are usually split into *training* and *test* splits, so that we may estimate the parameters of our models on training data, but evaluate our models

performance on unseen data using the test set. However, we typically have to make a lot of decisions when training a model, e.g. which model to use, how to set the various hyperparameters in our setting, etc. Generally, we would like to know how these decisions impact our model performance, but if we compare two different hyperparameter settings on test data, e.g. whether to use a bias or not, then we would identify which one works best on that test data, which would make that data no longer unseen (we would already know something about it, e.g. which settings work best for it).

Validation split exists to allow for the model selection process without compromising the test data split. That is, this is the evaluation data we can use to train various models under different settings and choose the settings that yield the best performing model on the validation set, to then finally report its performance on the test set, which has remained unseen throughout the entire process (typically by training on the training and validation sets combined to get more data). Note that these splits are typically constructed by taking an existing dataset, shuffling its examples and then randomly sampling examples for each split without replacement, in order to ensure all splits come from the same distribution.

(f) We say underfitting occurs when a model is too simple to perform well on a task, even on the data it is trained on. Conversely, we say overfitting occurs when a model is clearly capable of performing well on a task to the point that it learns all the detail and noise in the data it was trained on, while still not performing well on unseen data.

We can measure underfitting by looking at the model's performance on the training set, e.g. compute the accuracy of the training set in a classification setting. We can measure overfitting by looking at a model's performance on unseen data, e.g. a validation split and contrasting it with its performance on the training set. High performance during training but low performance on unseen data is a clear indication of overfitting. In practice, this is not so straightforward to see, as difference in performance across training and validation are common, so a baseline is required to determine whether the difference is significant/problematic.

2 Log-linear Classifiers

(a) Given that logistic regression is a linear classifier, a logistic regression model that takes an n-dimensional vector as input, and does not use a bias, can be formally described with the following linear combination:

$$p(y = 1 \mid x) = \sigma\left(\sum_{i=1}^{n} w_i \cdot x_i\right) \tag{1}$$

$$p(y = 0 \mid x) = 1 - p(y = 1), \tag{2}$$

where $w_i \in \mathbb{R}$ is the parameter (weight) that corresponds to input feature x_i. Being a linear combination of the input features, this model has as many parameters as there are features in the input vector.

(b) We have:

$$p(y = 1 \mid x) = \sigma(\boldsymbol{w}^\top \boldsymbol{x}). \tag{3}$$

(c) We have:

$$p(y = 1 \mid x) = \sigma\left(\sum_{i=1}^{n}(w_i \cdot x_i) + b\right) \tag{4}$$

$$p(y = 0 \mid x) = 1 - p(y = 1), \tag{5}$$

where $b \in \mathbb{R}$ is the bias term.

We can still write this in vectorized form, but to do this comfortably we make use of what we call a *bias feature* $x_0 = 1$, which acts as the coefficient for the bias weight in the linear combination. That is:

$$p(y = 1 \mid x) = \sigma(\boldsymbol{w}^\top \boldsymbol{x}). \tag{6}$$

where $\boldsymbol{x} \in \mathbb{R}^{n+1}$, i.e. $\boldsymbol{x} = (x_o, x_1, \ldots, x_n)$.

(d) We have:

$$f(y = 1 \mid \boldsymbol{x}) = \exp(\boldsymbol{w}^\top \boldsymbol{x}) = e^{\boldsymbol{w}^\top \boldsymbol{x}}, \tag{7}$$

$$f(y = 0 \mid \boldsymbol{x}) = 1 - f(y = 1 \mid \boldsymbol{x}). \tag{8}$$

(e) For $c \in C$ classes, we have:

$$f(y = c \mid \boldsymbol{x}) = \exp(\boldsymbol{w}_c^\top \boldsymbol{x}) = e^{\boldsymbol{w}_c^\top \boldsymbol{x}}, \tag{9}$$

where $\boldsymbol{w}_c$ are the parameters used for making predictions for class c. That is, this general log-linear model has as many parameters as there are input features *for each class* $c \in C$.

(f) We can accomplish this by normalizing the output of the model, so that it represents a probability distribution over the classes. Concretely, we have:

$$p(y = c \mid \boldsymbol{x}) = \frac{\exp(\boldsymbol{w}_c^\top \boldsymbol{x})}{\sum_{c' \in C} \exp(\boldsymbol{w}_{c'}^\top \boldsymbol{x})}, \tag{10}$$

where C is the set of classes in our classification task.

Eq. 10 is known as a *softmax function*, which gives the name to this multi-class log-linear model: softmax regression. Note that given input vector $\boldsymbol{x}$ and class c, the function returns a single probability value. That means that, for a given input vector $\boldsymbol{x}$, we can get an entire "softmax distribution" over classes by computing this function for all classes $c \in C$. Specifically, we can compute $\boldsymbol{w}_c^\top \boldsymbol{x}$ for all $c \in C$ and store these results in a C-dimensional vector $\boldsymbol{l} \in \mathbb{R}^C$. Then, by normalizing each entry in that vector as with Eq. 10, we get a probability vector $\boldsymbol{y} \in \mathbb{R}^C$ that contains the distribution over all classes, i.e. where all of its entries add up to 1. That is:

$$\boldsymbol{y} = softmax(\boldsymbol{l}), \tag{11}$$

where the i-th component of $\boldsymbol{y}$ is given by:

$$\boldsymbol{y}_i = \frac{e^{l_i}}{\sum_{j=1}^{C} e^{l_j}}. \tag{12}$$

This is typically the way the softmax function is represented in vectorized form. This is a very interpretable output for a classification model.

(g) Let $\boldsymbol{W} \in \mathbb{R}^{n \times C}$ be parameters of our model such that column $\boldsymbol{w}_i$ corresponds to class $i \in C$. Using our vectorized softmax function from the previous question, we have:

$$\boldsymbol{Y} = softmax(\boldsymbol{X}\boldsymbol{W}), \tag{13}$$

where we assume our softmax operation is applied row-wise.

(h) There are a few reasons for this, but in short, there are some computational reasons, and some statistical reasons. The computational reasons are (i) it simplifies the math and therefore computations, e.g. when computing likehoods of multiple data points, and (ii) it helps to avoid numerical underflow issues that can occur when multiplying many probabilities together. The main statistical reason is that it makes the model more interpretable, as it can be related to concepts like odds and independence.

3 Training Log-linear Classifiers

(a) We have:

$$p(X \mid \boldsymbol{w}) = p(x_1 \mid \boldsymbol{w})p(x_2 \mid \boldsymbol{w}) \dots p(x_n \mid \boldsymbol{w}) = \prod_{i=1}^{n} p(x_i \mid \boldsymbol{w}). \tag{14}$$

(b) We have:

$$p(X \mid \boldsymbol{w}) = \prod_{x_p \in X_{pos}} p(x_p \mid \boldsymbol{w}) \prod_{x_n \in X_{neg}} \left(1 - p(x_n \mid \boldsymbol{w})\right). \tag{15}$$

(c) We have:

$$\mathcal{L}(\boldsymbol{w} \mid X) = \prod_{x_p \in X_{pos}} p(x_p \mid \boldsymbol{w}) \prod_{x_n \in X_{neg}} \left(1 - p(x_n \mid \boldsymbol{w})\right). \tag{16}$$

(d) We have:

$$\begin{aligned}
\mathcal{LL}(\boldsymbol{w} \mid X) &= \ln p(X \mid \boldsymbol{w}) && (17)\\
&= \ln \left(\prod_{i=1}^{n} p(x_i \mid \boldsymbol{w}) \right) && (18)\\
&= \ln \left(\prod_{x_p \in X_{pos}} p(x_p \mid \boldsymbol{w}) \prod_{x_n \in X_{neg}} \left(1 - p(x_n \mid \boldsymbol{w})\right) \right) && (19)\\
&= \sum_{x_p \in X_{pos}} \ln p(x_p \mid \boldsymbol{w}) + \sum_{x_n \in X_{neg}} \ln \left(1 - p(x_n \mid \boldsymbol{w})\right). && (20)
\end{aligned}$$

This is the general expression we want to maximize when training log-linear models with MLE using the i.i.d. assumption. Specifically, we want to find the value of $\boldsymbol{w}$ that maximizes Eq. 20.

(e) We have:

$$\mathcal{LL}(\boldsymbol{w} \mid X) = \sum_{x_p \in X_{pos}} \ln \sigma(\boldsymbol{w}^\top \boldsymbol{x}) + \sum_{x_n \in X_{neg}} \ln \Big(1 - \sigma(\boldsymbol{w}^\top \boldsymbol{x})\Big) \tag{21}$$

$$= \sum_{x_p \in X_{pos}} \ln \frac{e^{\boldsymbol{w}^\top \boldsymbol{x}_p}}{1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_p}} + \sum_{x_n \in X_{neg}} \ln \left(1 - \frac{e^{\boldsymbol{w}^\top \boldsymbol{x}_n}}{1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_n}}\right) \tag{22}$$

$$= \sum_{x_p \in X_{pos}} \ln \frac{e^{\boldsymbol{w}^\top \boldsymbol{x}_p}}{1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_p}} + \sum_{x_n \in X_{neg}} \ln \left(\frac{1}{1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_n}}\right) \tag{23}$$

$$= \sum_{x_p \in X_{pos}} \Big(\boldsymbol{w}^\top \boldsymbol{x}_p - \ln(1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_p})\Big) - \sum_{x_n \in X_{neg}} \ln \Big(1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_n}\Big). \tag{24}$$

(f) Let $\nabla = \nabla_{\boldsymbol{w}}$. We have:

$$\nabla \mathcal{LL} = \nabla \left(\sum_{x_p \in X_{pos}} \Big(\boldsymbol{w}^\top \boldsymbol{x}_p - \ln(1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_p})\Big) - \sum_{x_n \in X_{neg}} \ln \Big(1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_n}\Big) \right) \tag{25}$$

$$= \nabla \sum_{x_p \in X_{pos}} \Big(\boldsymbol{w}^\top \boldsymbol{x}_p - \ln(1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_p})\Big) - \nabla \sum_{x_n \in X_{neg}} \ln \Big(1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_n}\Big) \tag{26}$$

$$= \sum_{x_p \in X_{pos}} \Big(\boldsymbol{x}_p - \nabla \ln(1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_p})\Big) - \sum_{x_n \in X_{neg}} \nabla \ln \Big(1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_n}\Big) \tag{27}$$

$$= \sum_{x_p \in X_{pos}} \left(\boldsymbol{x}_p - \frac{e^{\boldsymbol{w}^\top \boldsymbol{x}_p}}{1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_p}} \boldsymbol{x}_p\right) - \sum_{x_n \in X_{neg}} \left(\frac{e^{\boldsymbol{w}^\top \boldsymbol{x}_n}}{1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_n}} \boldsymbol{x}_n\right) \tag{28}$$

$$= \sum_{x_p \in X_{pos}} \left(1 - \frac{e^{\boldsymbol{w}^\top \boldsymbol{x}_p}}{1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_p}}\right) \boldsymbol{x}_p - \sum_{x_n \in X_{neg}} \left(\frac{e^{\boldsymbol{w}^\top \boldsymbol{x}_n}}{1 + e^{\boldsymbol{w}^\top \boldsymbol{x}_n}} \boldsymbol{x}_n\right) \tag{29}$$

$$= \sum_{x_p \in X_{pos}} \Big(1 - \sigma(\boldsymbol{w}^\top \boldsymbol{x}_p)\Big) \boldsymbol{x}_p - \sum_{x_n \in X_{neg}} \sigma(\boldsymbol{w}^\top \boldsymbol{x}_n) \boldsymbol{x}_n. \tag{30}$$

(g) For SGD, see Algorithm 1. For BGD, see Algorithm 2.

Algorithm 1 Stochastic Gradient Descent

procedure GRADIENT DESCENT(Dataset X, Objective function J, Model $p_{\boldsymbol{w}}$, Learning rate lr, Number of Epochs N)
  $\boldsymbol{w} \leftarrow random()$ // random weight initialization
  for $i \in \{1, \dots N\}$ **do** // we iterate over epochs
    for $x_i \in X$ **do** // we iterate over the dataset
      $g_i \leftarrow \nabla_{\boldsymbol{w}} J(p_{\boldsymbol{w}}(x_i))$
      $\boldsymbol{w}_{i+1} \leftarrow \boldsymbol{w}_i - lr \cdot g_i$ // update rule
    end for
  end for
  Return $p_{\boldsymbol{w}}$
end procedure

Note that we still rely on gradients that need to be computed every time we update our parameters. But we don't compute gradients manually when working with these models.

Algorithm 2 Batch Gradient Descent

procedure GRADIENT DESCENT(Dataset X, Objective function J, Model $p_{\boldsymbol{w}}$, Learning rate lr, Number of Epochs N, Batch Size b)
  $\boldsymbol{w} \leftarrow random()$ // random weight initialization
  for $i \in \{1, \dots N\}$ **do**
    for $batch = [x_1, \dots, x_b]$ where $batch$ is sampled without replacement from X **do**
      $g_i \leftarrow \nabla_{\boldsymbol{w}} J(p_{\boldsymbol{w}}(batch))$
      $\boldsymbol{w}_{i+1} \leftarrow \boldsymbol{w}_i - lr \cdot g_i$ // update rule
    end for
  end for
  Return $p_{\boldsymbol{w}}$
end procedure

This is generally handled by tools like Autograd. But it's important to understand the process underlying the training of models, despite it being automatic, so we can reason about model performance as a function of its training process.

(h) For SGD, see Algorithm 3. For BGD, see Algorithm 4.

Algorithm 3 Stochastic Gradient Descent with Shuffling

procedure GRADIENT DESCENT(Dataset X, Objective function J, Model $p_{\boldsymbol{w}}$, Learning rate lr, Number of Epochs N)
  $\boldsymbol{w} \leftarrow random()$ // random weight initialization
  for $i \in \{1, \dots N\}$ **do** // we iterate over epochs
    for $x_i \in X$ **do** // we iterate over the dataset
      $g_i \leftarrow \nabla_{\boldsymbol{w}} J(p_{\boldsymbol{w}}(x_i))$
      $\boldsymbol{w}_{i+1} \leftarrow \boldsymbol{w}_i - lr \cdot g_i$ // update rule
    end for
    shuffle(X)
  end for
  Return $p_{\boldsymbol{w}}$
end procedure

Algorithm 4 Batch Gradient Descent with Shuffling

procedure GRADIENT DESCENT(Dataset X, Objective function J, Model $p_{\boldsymbol{w}}$, Learning rate lr, Number of Epochs N, Batch Size b)
  $\boldsymbol{w} \leftarrow random()$ // random weight initialization
  for $i \in \{1, \dots N\}$ **do**
    for $batch = [x_1, \dots, x_b]$ where $batch$ is sampled without replacement from X **do**
      $g_i \leftarrow \nabla_{\boldsymbol{w}} J(p_{\boldsymbol{w}}(batch))$
      $\boldsymbol{w}_{i+1} \leftarrow \boldsymbol{w}_i - lr \cdot g_i$ // update rule
    end for
    shuffle(X)
  end for
  Return $p_{\boldsymbol{w}}$
end procedure

While GD considers the entire dataset when computing gradients, and SGD treats all

examples equally, BGD uses a subset of the dataset to compute each gradient. Shuffling the dataset at the end of each epoch is a common practice to avoid using the same subset of examples for each BGD update, as this may bias the training process in favor of some examples over others. BGD is by far the most method used in NLP training. It has been empirically shown that SGD and BGD can converger faster than GD, but they can also be more unstable.