CS 188 Introduction to Artificial Intelligence
Spring 2023 Note 23

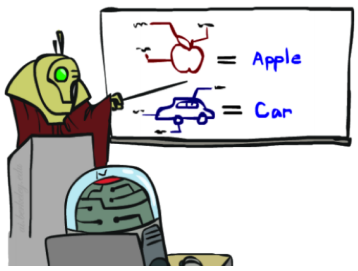
These lecture notes are based on notes originally written by Nikhil Sharma and on the textbook *Artificial Intelligence: A Modern Approach*.

Last updated: January 15, 2023

Machine Learning

In the previous few notes of this course, we've learned about various types of models that help us reason under uncertainty. Until now, we've assumed that the probabilistic models we've worked with can be taken for granted, and the methods by which the underlying probability tables we worked with were generated have been abstracted away. We'll begin to break down this abstraction barrier as we delve into our discussion of **machine learning**, a broad field of computer science that deals with constructing and/or learning the parameters of a specified model given some data.

There are many machine learning algorithms which deal with many different types of problems and different types of data, classified according to the tasks they hope to accomplish and the types of data that they work with. Two primary subgroups of machine learning algorithms are **supervised learning algorithms** and **unsupervised learning algorithms**. Supervised learning algorithms infer a relationship between input data and corresponding output data in order to predict outputs for new, previously unseen input data. Unsupervised learning algorithms, on the other hand, have input data that doesn't have any corresponding output data and so deal with recognizing inherent structure between or within datapoints and grouping and/or processing them accordingly. In this class, the algorithms we'll discuss will be limited to supervised learning tasks.



(a) Training

(b) Validation

(c) Testing

Once you have a dataset that you're ready to learn with, the machine learning process usually involves splitting your dataset into three distinct subsets. The first, **training data**, is used to actually generate a model mapping inputs to outputs. Then, **validation data** (also known as **hold-out** or **development data**) is used to measure your model's performance by making predictions on inputs and generating an accuracy score. If your model doesn't perform as well as you'd like it to, it's always okay to go back and train again, either by adjusting special model-specific values called **hyperparameters** or by using a different learning algorithm altogether until you're satisfied with your results. Finally, use your model to make predictions on the third and final subset of your data, the **test set**. The test set is the portion of your data that's never seen by

your agent until the very end of development, and is the equivalent of a "final exam" to gauge performance on real-world data.

In what follows we will be covering some foundational machine learning algorithms, such as Naive Bayes, Linear Regression, Logistic Regression, and the Perceptron algorithm.

Naive Bayes

We'll motivate our discussion of machine learning with a concrete example of a machine learning algorithm. Let's consider the common problem of building an email spam filter which sorts messages into spam (unwanted email) or ham (wanted email). Such a problem is called a **classification problem** – given various datapoints (in this case, each email is a datapoint), our goal is to group them into one of two or more **classes**. For classification problems, we're given a training set of datapoints along with their corresponding **labels**, which are typically one of a few discrete values.

As we've discussed, our goal will be to use this training data (emails, and a spam/ham label for each one) to learn some sort of relationship that we can use to make predictions on previously unseen emails. In this section we'll describe how to construct a type of model for solving classification problems known as a **Naive Bayes Classifier**.

To train a model to classify emails as spam or ham, we need some training data consisting of preclassified emails that we can learn from. However, emails are simply strings of text, and in order to learn anything useful, we need to extract certain attributes from each of them known as **features**. Features can be anything ranging from specific word counts to text patterns (e.g. whether words are in all caps or not) to pretty much any other attribute of the data that you can imagine.

The specific features extracted for training are often dependent on the specific problem you're trying to solve and which features you decide to select can often impact the performance of your model dramatically. Deciding which features to utilize is known as **feature engineering** and is fundamental to machine learning, but for the purposes of this course you can assume you'll always be given the extracted features for any given dataset. In this note, $\mathbf{f}(\mathbf{x})$ refers to a feature function applied to all inputs $\mathbf{x}$ before putting them in the model.

Now let's say you have a dictionary of n words, and from each email you extract a feature vector $F \in \mathbb{R}^n$ where the i^{th} entry in F is a random variable F_i which can take on a value of either a 0 or a 1 depending on whether the i^{th} word in your dictionary appears in the email under consideration. For example, if F_{200} is the feature for the word *free*, we will have $F_{200} = 1$ if *free* appears in the email, and 0 if it does not. With these definitions, we can define more concretely how to predict whether or not an email is spam or ham – if we can generate a joint probability table between each F_i and and the label Y, we can compute the probability

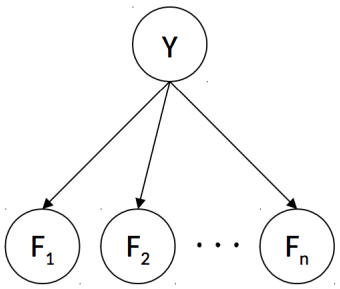
any email under consideration is spam or ham given it's feature vector. Specifically, we can compute both

$$P(Y = spam|F_1 = f_1, \ldots, F_n = f_n)$$

and

$$P(Y = ham|F_1 = f_1, \ldots, F_n = f_n)$$

and simply label the email depending on which of the two probabilities is higher. Unfortunately, since we have n features and 1 label, each of which can take on 2 distinct values, the joint probability table corresponding to this distribution mandates a table size that's exponential in n with 2^{n+1} entries - very impractical! This problem is solved by modeling the joint probability table with a Bayes' net, making the critical simplifying assumption that each feature F_i is independent of all other features given the class label. This is a very strong modeling assumption (and the reason that Naive Bayes is called *naive*), but it simplifies inference and usually works well in practice. It leads to the following Bayes' net to represent our desired joint probability distribution.



Note that the rules of d-separation delineated earlier in the course make it immediately clear that in this Bayes' net each F_i is conditionally independent of all the others, given Y. Now we have one table for $P(Y)$ with 2 entries, and n tables for each $P(F_i \mid Y)$ each with $2^2 = 4$ entries for a total of $4n+2$ entries - linear in n! This simplifying assumption highlights the trade off that arises from the concept of **statistical efficiency**; we sometimes need to compromise our model's complexity in order to stay within the limits of our computational resources.

Indeed, in cases where the number of features is sufficiently low, it's common to make more assumptions about relationships between features to generate a better model (corresponding to adding edges to your Bayes' net). With this model we've adopted, making predictions for unknown data points amounts to running inference on our Bayes' net. We have observed values for $F_1, \cdots F_n$, and want to choose the value of Y that has the highest probability conditioned on these features:

$$\begin{aligned} prediction(f_1, \cdots f_n) &= \operatorname*{argmax}_y P(Y = y|F_1 = f_1, \ldots F_N = f_n) \\ &= \operatorname*{argmax}_y P(Y = y, F_1 = f_1, \ldots F_N = f_n) \\ &= \operatorname*{argmax}_y P(Y = y) \prod_{i=1}^{n} P(F_i = f_i|Y = y) \end{aligned}$$

where the first step is because the highest probability class will be the same in the normalized or unnormalized distribution, and the second comes directly from the Naive Bayes' independence assumption that features are independent given the class label (as seen in the graphical model structure).

Generalizing away from a spam filter, assume now that there are k class labels (possible values for Y). Additionally, after noting that our desired probabilities - the probability of each label y_i given our features,

$P(Y = y_i | F_1 = f_1, \ldots, F_n = f_n)$ - is proportional to the joint $P(Y = y_i, F_1 = f_1, \ldots, F_n = f_n)$, we can compute:

$$P(Y, F_1 = f_1, \ldots, F_n = f_n) = \begin{bmatrix} P(Y = y_1, F_1 = f_1, \ldots, F_n = f_n) \\ P(Y = y_2, F_1 = f_1, \ldots, F_n = f_n) \\ \vdots \\ P(Y = y_k, F_1 = f_1, \ldots, F_n = f_n) \end{bmatrix} = \begin{bmatrix} P(Y = y_1) \prod_i P(F_i = f_i | Y = y_1) \\ P(Y = y_2) \prod_i P(F_i = f_i | Y = y_2) \\ \vdots \\ P(Y = y_k) \prod_i P(F_i = f_i | Y = y_k) \end{bmatrix}$$

Our prediction for class label corresponding to the feature vector F is simply the label corresponding to the maximum value in the above computed vector:

$$prediction(F) = \underset{y_i}{\text{argmax}} \; P(Y = y_i) \prod_j P(F_j = f_j | Y = y_i)$$

We've now learned the basic theory behind the modeling assumptions of the Naive Bayes' classifier and how to make predictions with one, but have yet to touch on how exactly we learn the conditional probability tables used in our Bayes' net from the input data. This will have to wait for our next topic of discussion, **parameter estimation**.

Parameter Estimation

Assume you have a set of N **sample points** or **observations**, $x_1, \ldots, x_N$, and you believe that this data was drawn from a distribution **parametrized** by an unknown value θ. In other words, you believe that the probability $P_\theta(x_i)$ of each of your observations is a function of θ. For example, we could be flipping a coin which has probability θ of coming up heads.

How can you "learn" the most likely value of θ given your sample? For example, if we have 10 coin flips, and saw that 7 of them were heads, what value should we choose for θ? One answer to this question is to infer that θ is equal to the value that maximizes the probability of having selected your sample $x_1, \ldots, x_N$ from your assumed probability distribution. A frequently used and fundamental method in machine learning known as **maximum likelihood estimation** (MLE) does exactly this.

Maximum likelihood estimation typically makes the following simplifying assumptions:

- Each sample is drawn from the same distribution. In other words, each x_i is **identically distributed**. In our coin flipping example, each coin flip has the same chance, θ, of coming up heads.
- Each sample x_i is conditionally **independent** of the others, given the parameters for our distribution. This is a strong assumption, but as we'll see greatly helps simplify the problem of maximum likelihood estimation and generally works well in practice. In the coin flipping example, the outcome of one flip doesn't affect any of the others.
- All possible values of θ are equally likely before we've seen any data (this is known as a **uniform prior**).

The first two assumptions above are often referred to as **independent, identically distributed** (i.i.d.). The third assumption above makes the MLE method a special case of the maximum a priori (MAP) method, which allows for non-uniform priors.

Let's now define the **likelihood** $\mathscr{L}(\theta)$ of our sample, a function which represents the probability of having drawn our sample from our distribution. For a fixed sample x_1, x_N, the likelihood is just a function of θ:

$$\mathscr{L}(\theta) = P_\theta(x_1, \ldots, x_N)$$

Using our simplifying assumption that the samples x_i are i.i.d., the likelihood function can be re-expressed as follows:

$$\mathscr{L}(\theta) = \prod_{i=1}^{N} P_\theta(x_i)$$

How can we find the value of θ that maximizes this function? This will be the value of θ that best explains the data we saw. Recall from calculus that at points where a function's maxima and minima are realized, its first derivative with respect to each of its inputs (also known as the function's **gradient**) must be equal to zero. Hence, the maximum likelihood estimate for θ is a value that satisfies the following equation:

$$\frac{\partial}{\partial \theta}\mathscr{L}(\theta) = 0$$

Let's go through an example to make this concept more concrete. Say you have a bag filled with red and blue balls and don't know how many of each there are. You draw samples by taking a ball out of the bag, noting the color, then putting the ball back in (sampling with replacement). Drawing a sample of three balls from this bag yields *red*, *red*, *blue*. This seems to imply that we should infer that $\frac{2}{3}$ of the balls in the bag are red and $\frac{1}{3}$ of the balls are blue. We'll assume that each ball being taken out of the bag will be red with probability θ and blue with probability $1-\theta$, for some value θ that we want to estimate (this is known as a Bernoulli distribution):

$$P_\theta(x_i) = \begin{cases} \theta & x_i = red \\ (1-\theta) & x_i = blue \end{cases}$$

The likelihood of our sample is then:

$$\mathscr{L}(\theta) = \prod_{i=1}^{3} P_\theta(x_i) = P_\theta(x_1 = red)P_\theta(x_2 = red)P_\theta(x_2 = blue) = \theta^2 \cdot (1-\theta)$$

The final step is to set the derivative of the likelihood to 0 and solve for θ:

$$\frac{\partial}{\partial \theta}\mathscr{L}(\theta) = \frac{\partial}{\partial \theta}\theta^2 \cdot (1-\theta) = \theta(2-3\theta) = 0$$

Solving this equation for θ yields $\theta = \frac{2}{3}$, which intuitively makes sense! (There's a second solution, too, $\theta = 0$ – but this corresponds to a minimum of the likelihood function, as $\mathscr{L}(0) = 0 < \mathscr{L}(\frac{2}{3}) = \frac{4}{27}$.)

Maximum Likelihood for Naive Bayes

Let's now return to the problem of inferring conditional probability tables for our spam classifier, beginning with a recap of variables we know:

- n - the number of words in our dictionary.
- N - the number of observations (emails) you have for training. For our upcoming discussion, let's also define N_h as the number of training samples labeled as ham and N_s as the number of training samples labeled as spam. Note $N_h + N_s = N$.
- F_i - a random variable which is 1 if the i^{th} dictionary word is present in an email under consideration, and 0 otherwise.
- Y - a random variable that's either *spam* or *ham* depending on the label of the corresponding email.

- $f_i^{(j)}$ - this references the resolved value of the random variable F_i in the j^{th} item in the training set. In other words, each $f_i^{(j)}$ is a 1 if word i appeared in j^{th} email under consideration and 0 otherwise. This is the first time we're seeing this notation, but it'll come in handy in the upcoming derivation.

Now within each conditional probability table $P(F_i|Y)$, note that we have two distinct Bernoulli distributions: $P(F_i|Y = ham)$ and $P(F_i|Y = spam)$. For simplicity, let's specifically consider $P(F_i|Y = ham)$ and try to find the maximum likelihood estimate for a parameter $\theta = P(F_i = 1|Y = ham)$ i.e. the probability that the i^{th} word in our dictionary appears in a ham email. Since we have N_h ham emails in our training set, we have N_h observations of whether or not word i appeared in a ham email. Because our model assumes a Bernoulli distribution for the appearance of each word given its label, we can formulate our likelihood function as follows:

$$\mathscr{L}(\theta) = \prod_{j=1}^{N_h} P(F_i = f_i^{(j)}|Y = ham) = \prod_{j=1}^{N_h} \theta^{f_i^{(j)}}(1-\theta)^{1-f_i^{(j)}}$$

The second step comes from a small mathematical trick: if $f_i^{(j)} = 1$ then

$$P(F_i = f_i^{(j)}|Y = ham) = \theta^1(1-\theta)^0 = \theta$$

and similarly if $f_i^{(j)} = 0$ then

$$P(F_i = f_i^{(j)}|Y = ham) = \theta^0(1-\theta)^1 = (1-\theta)$$

In order to compute the maximum likelihood estimate for θ, recall that the next step is to compute the derivative of $\mathscr{L}(\theta)$ and set it equal to 0. Attempting this proves quite difficult, as it's no simple task to isolate and solve for θ. Instead, we'll employ a trick that's very common in maximum likelihood derivations, and that's to instead find the value of θ the maximizes the log of the likelihood function. Because $\log(x)$ is a strictly increasing function (sometimes referred to as a **monotonic transformation**), finding a value that maximizes $\log \mathscr{L}(\theta)$ will also maximize $\mathscr{L}(\theta)$. The expansion of $\log \mathscr{L}(\theta)$ is below:

$$\begin{aligned}
\log \mathscr{L}(\theta) &= \log\left(\prod_{j=1}^{N_h} \theta^{f_i^{(j)}}(1-\theta)^{1-f_i^{(j)}}\right) \\
&= \sum_{j=1}^{N_h} \log\left(\theta^{f_i^{(j)}}(1-\theta)^{1-f_i^{(j)}}\right) \\
&= \sum_{j=1}^{N_h} \log\left(\theta^{f_i^{(j)}}\right) + \sum_{j=1}^{N_h} \log\left((1-\theta)^{1-f_i^{(j)}}\right) \\
&= \log(\theta)\sum_{j=1}^{N_h} f_i^{(j)} + \log(1-\theta)\sum_{j=1}^{N_h}(1-f_i^{(j)})
\end{aligned}$$

Note that in the above derivation, we've used the properties of the log function that $\log(a^c) = c \cdot \log(a)$ and $\log(ab) = \log(a) + \log(b)$. Now we set the derivative of the log of the likelihood function to 0 and solve for θ:

$$
\begin{aligned}
\frac{\partial}{\partial\theta}\left(\log(\theta)\sum_{j=1}^{N_h} f_i^{(j)} + \log(1-\theta)\sum_{j=1}^{N_h}(1-f_i^{(j)})\right) &= 0 \\
\frac{1}{\theta}\sum_{j=1}^{N_h} f_i^{(j)} - \frac{1}{(1-\theta)}\sum_{j=1}^{N_h}(1-f_i^{(j)}) &= 0 \\
\frac{1}{\theta}\sum_{j=1}^{N_h} f_i^{(j)} &= \frac{1}{(1-\theta)}\sum_{j=1}^{N_h}(1-f_i^{(j)}) \\
(1-\theta)\sum_{j=1}^{N_h} f_i^{(j)} &= \theta\sum_{j=1}^{N_h}(1-f_i^{(j)}) \\
\sum_{j=1}^{N_h} f_i^{(j)} - \theta\sum_{j=1}^{N_h} f_i^{(j)} &= \theta\sum_{j=1}^{N_h} 1 - \theta\sum_{j=1}^{N_h} f_i^{(j)} \\
\sum_{j=1}^{N_h} f_i^{(j)} &= \theta \cdot N_h \\
\theta &= \frac{1}{N_h}\sum_{j=1}^{N_h} f_i^{(j)}
\end{aligned}
$$

We've arrived at a remarkably simple final result! According to our formula above, the maximum likelihood estimate for θ (which, remember, is the probability that $P(F_i = 1|Y = ham)$) corresponds to counting the number of ham emails in which word i appears and dividing it by the total number of ham emails. You may think this was a lot of work for an intuitive result (and it was), but the derivation and techniques will be useful for more complex distributions than the simple Bernoulli distribution we are using for each feature here. To summarize, in this Naive Bayes model with Bernoulli feature distributions, within any given class the maximum likelihood estimate for the probability of any outcome corresponds to the count for the outcome divided by the total number of samples for the given class. The above derivation can be generalized to cases where we have more than two classes and more than two outcomes for each feature, though this derivation is not provided here.

Smoothing

Though maximum likelihood estimation is a very powerful method for parameter estimation, bad training data can often lead to unfortunate consequences. For example, if every time the word "minute" appears in an email in our training set, that email is classified as spam, our trained model will learn that

$$P(F_{minute} = 1|Y = ham) = 0$$

Hence in an unseen email, if the word *minute* ever shows up, $P(Y = ham)\prod_i P(F_i|Y = ham) = 0$, and so your model will never classify any email containing the word *minute* as ham. This is a classic example of **overfitting**, or building a model that doesn't generalize well to previously unseen data. Just because a specific word didn't appear in an email in your training data, that doesn't mean that it won't appear in an email in your test data or in the real world. Overfitting with Naive Bayes' classifiers can be mitigated by **Laplace smoothing**. Conceptually, Laplace smoothing with strength k assumes having seen k extra of each outcome. Hence if for a given sample your maximum likelihood estimate for an outcome x that can take on

$|X|$ different values from a sample of size N is

$$P_{MLE}(x) = \frac{count(x)}{N}$$

then the Laplace estimate with strength k is

$$P_{LAP,k}(x) = \frac{count(x)+k}{N+k|X|}$$

What does this equation say? We've made the assumption of seeing k additional instances of each outcome, and so act as if we've seen $count(x)+k$ rather than $count(x)$ instances of x. Similarly, if we see k additional instances of each of $|X|$ classes, then we must add $k|X|$ to our original number of samples N. Together, these two statements yield the above formula. A similar result holds for computing Laplace estimates for conditionals (which is useful for computing Laplace estimates for outcomes across different classes):

$$P_{LAP,k}(x|y) = \frac{count(x,y)+k}{count(y)+k|X|}$$

There are two particularly notable cases for Laplace smoothing. The first is when $k = 0$, then $P_{LAP,0}(x) = P_{MLE}(x)$. The second is the case where $k = \infty$. Observing a very large, infinite number of each outcome makes the results of your actual sample inconsequential and so your Laplace estimates imply that each outcome is equally likely. Indeed:

$$P_{LAP,\infty}(x) = \frac{1}{|X|}$$

The specific value of k that's appropriate to use in your model is typically determined by trial-and-error. k is a hyperparameter in your model, which means that you can set it to whatever you want and see which value yields the best prediction accuracy/performance on your validation data.

Linear Regression

Now we'll move on from our previous discussion of Naive Bayes to **Linear Regression**. This method, also called **least squares**, dates all the way back to Carl Friedrich Gauss and is one of the most studied tools in machine learning and econometrics.

Regression problems are a form of machine learning problem in which the output is a continuous variable (denoted with y). The features can be either continuous or categorical. We will denote a set of features with $\mathbf{x} \in \mathbb{R}^n$ for n features, i.e. $\mathbf{x} = (x_1, \ldots, x_n)$.

We use the following linear model to predict the output:

$$h_{\mathbf{w}}(\mathbf{x}) = w_0 + w_1 x_1 + \cdots w_n x_n$$

where the weights w_i of the linear model are what we want to estimate. The weight w_0 corresponds to the intercept of the model. Sometimes in literature we add a 1 on the feature vector $\mathbf{x}$ so that we can write the linear model as $\mathbf{w}^T\mathbf{x}$ where now $\mathbf{x} \in \mathbb{R}^{n+1}$. To train the model, we need a metric of how well our model predicts the output. For that we will use the $L2$ loss function which penalizes the difference of the predicted from the actual output using the $L2$ norm. If our training dataset has N data points then the loss function is defined as follows:

$$Loss(h_{\mathbf{w}}) = \frac{1}{2}\sum_{j=1}^{N} L2(y_j, h_{\mathbf{w}}(\mathbf{x}_j)) = \frac{1}{2}\sum_{j=1}^{N}(y_j - h_{\mathbf{w}}(\mathbf{x}_j)^2 = \frac{1}{2}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2$$

Note that $\mathbf{x}_j$ corresponds to the jth data point $\mathbf{x}_j \in \mathbb{R}^n$. The term $\frac{1}{2}$ is just added to simplify the expressions of the closed form solution. The last expression is an equivalent formulation of the loss function which makes the least square derivation easier. The quantities $\mathbf{y}, \mathbf{X}$ and $\mathbf{w}$ are defined as follows:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} 1 & x_1^1 & \cdots & x_n^1 \\ 1 & x_1^2 & \cdots & x_n^2 \\ \vdots & \vdots & \cdots & \vdots \\ 1 & x_1^N & \cdots & x_n^N \end{bmatrix}, \quad \mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix},$$

where $\mathbf{y}$ is the vector of the stacked outputs, $\mathbf{X}$ is the matrix of the feature vectors where x_i^j denotes the ith component of the jth data point. The least squares solution denoted with $\hat{\mathbf{w}}$ can now be derived using basic linear algebra rules [1]. More specifically, we will find the $\hat{\mathbf{w}}$ that minimizes the loss function by differentiating the loss function and setting the derivative equal to zero.

$$\begin{aligned}\nabla_{\mathbf{w}} \frac{1}{2}\|\mathbf{y} - \mathbf{X}\mathbf{w}\|_2^2 &= \nabla_{\mathbf{w}} \frac{1}{2}(\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w}) = \nabla_{\mathbf{w}} \frac{1}{2}\left(\mathbf{y}^T\mathbf{y} - \mathbf{y}^T\mathbf{X}\mathbf{w} - \mathbf{w}^T\mathbf{X}^T\mathbf{y} + \mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w}\right) \\ &= \nabla_{\mathbf{w}} \frac{1}{2}\left(\mathbf{y}^T\mathbf{y} - 2\mathbf{w}^T\mathbf{X}^T\mathbf{y} + \mathbf{w}^T\mathbf{X}^T\mathbf{X}\mathbf{w}\right) = -\mathbf{X}^T\mathbf{y} + \mathbf{X}^T\mathbf{X}\mathbf{w}.\end{aligned}$$

Setting the gradient equal to zero we obtain:

$$-\mathbf{X}^T\mathbf{y} + \mathbf{X}^T\mathbf{X}\mathbf{w} = 0 \Rightarrow \hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}.$$

Having obtained the estimated vector of weights we can now make a prediction on new unseen test data points as follows:

$$h_{\hat{\mathbf{w}}}(\mathbf{x}) = \hat{\mathbf{w}}^T\mathbf{x}.$$

Perceptron

Linear Classifiers

The core idea behind Naive Bayes is to extract certain attributes of the training data called features and then estimate the probability of a label given the features: $P(y|f_1, f_2, ...f_n)$. Thus, given a new data point, we can then extract the corresponding features, and classify the new data point with the label with the highest probability given the features. This all, however, this requires us to estimate distributions, which we did with MLE. What if instead we decided not to estimate the probability distribution? Lets start by looking at a simple linear classifier, which we can use for **binary classification**, which is when the label has two possibilities, positive or negative.

[1] For matrix algebra rules you can refer to The Matrix Cookbook.

The basic idea of a **linear classifier** is to do classification using a linear combination of the features– a value which we call the **activation**. Concretely, the activation function takes in a data point, multiplies each feature of our data point, $f_i(\mathbf{x})$, by a corresponding weight, w_i, and outputs the sum of all the resulting values. In vector form, we can also write this as a dot product of our weights as a vector, $\mathbf{w}$, and our featurized data point as a vector $\mathbf{f}(\mathbf{x})$:

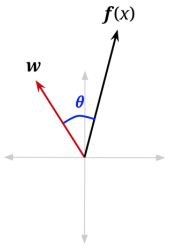
$$\text{activation}_w(\mathbf{x}) = h_{\mathbf{w}}(\mathbf{x}) = \sum_i w_i f_i(\mathbf{x}) = \mathbf{w}^T \mathbf{f}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{f}(\mathbf{x})$$

How does one do classification using the activation? For binary classification, when the activation of a data point is positive, we classify that data point with the positive label, and if it is negative, we classify with the negative label.

$$\text{classify}(\mathbf{x}) = \begin{cases} + & \text{if } h_{\mathbf{w}}(\mathbf{x}) > 0 \\ - & \text{if } h_{\mathbf{w}}(\mathbf{x}) < 0 \end{cases}$$

To understand this geometrically, let us reexamine the vectorized activation function. We can rewrite the dot product as follows, where $\|\cdot\|$ is the magnitude operator and θ is the angle between $\mathbf{w}$ and $\mathbf{f}(\mathbf{x})$:

$$h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w} \cdot \mathbf{f}(\mathbf{x}) = \|\mathbf{w}\| \|\mathbf{f}(\mathbf{x})\| \cos(\theta)$$



Since magnitudes are always non-negative, and our classification rule looks at the sign of the activation, the only term that matters for determining the class is $\cos(\theta)$.

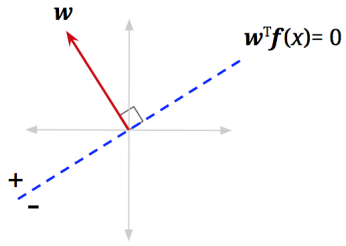
$$\text{classify}(\mathbf{x}) = \begin{cases} + & \text{if } cos(\theta) > 0 \\ - & \text{if } cos(\theta) < 0 \end{cases}$$

We, therefore, are interested in when $cos(\theta)$ is negative or positive. It is easily seen that for $\theta < \frac{\pi}{2}$, $cos(\theta)$ will be somewhere in the interval $(0,1]$, which is positive. For $\theta > \frac{\pi}{2}$, $cos(\theta)$ will be somewhere in the interval $[-1,0)$, which is negative. You can confirm this by looking at a unit circle. Essentially, our simple linear classifier is checking to see if the feature vector of a new data point roughly "points" in the same direction as a predefined weight vector and applies a positive label if it does.

$$\text{classify}(\mathbf{x}) = \begin{cases} + & \text{if } \theta < \frac{\pi}{2} \quad \text{(i.e. when } \theta \text{ is less than } 90°\text{, or acute)} \\ - & \text{if } \theta > \frac{\pi}{2} \quad \text{(i.e. when } \theta \text{ is greater than } 90°\text{, or obtuse)} \end{cases}$$

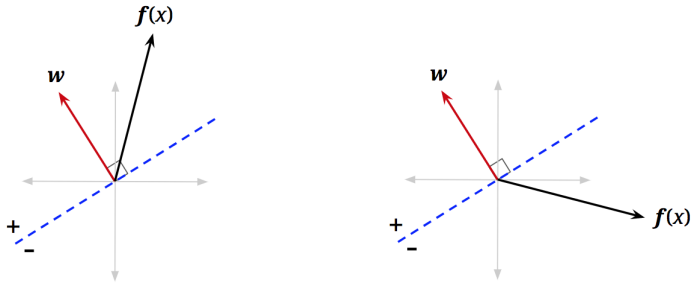
Up to this point, we haven't considered the points where $\text{activation}_w(\mathbf{x}) = \mathbf{w}^T \mathbf{f}(\mathbf{x}) = 0$. Following all the same logic, we will see that $cos(\theta) = 0$ for those points. Furthermore, $\theta = \frac{\pi}{2}$ (i.e θ is $90°$) for those points. In otherwords, these are the data points with feature vectors that are orthogonal to $\mathbf{w}$. We can add a dotted blue line, orthogonal to $\mathbf{w}$, where any feature vector that lies on this line will have activation equaling 0.

We call this blue line the **decision boundary** because it is the boundary that separates the region where we classify data points as positive from the region of negatives. In higher dimensions, a linear decision



Decision Boundary

boundary is generically called a **hyperplane**. A hyperplane is a linear surface that is one dimension lower than the latent space, thus dividing the surface in two. For general classifiers (non-linear ones), the decision boundary may not be linear, but is simply defined as surface in the space of feature vectors that separates the classes. To classify points that end up on the decision boundary, we can apply either label since both classes are equally valid (in the algorithms below, we'll classify points on the line as positive).



x **classified into positive class** *x* **classified into negative class**

Binary Perceptron

Great, now you know how linear classifiers work, but how do we build a good one? When building a classifier, you start with data, which are labeled with the correct class, we call this the **training set**. You build a classifier by evaluating it on the training data, comparing that to you training labels, and adjusting the parameters of your classifier until you reach your goal.

Let's explore one specific implementation of a simple linear classifier: the binary perceptron. The perceptron is a binary classifier–though it can be extended to work on more than two classes. The goal of the binary perceptron is to find a decision boundary that perfectly separates the training data. In other words, we're seeking the best possible weights– the best **w**– such that any featured training point that is multiplied by the weights, can be perfectly classified.

The Algorithm

The perceptron algorithm works as follows:

1. Initialize all weights to 0: $\mathbf{w} = \mathbf{0}$
2. For each training sample, with features $\mathbf{f}(\mathbf{x})$ and true class label $y^* \in \{-1, +1\}$, do:
   (a) Classify the sample using the current weights, let y be the class predicted by your current $\mathbf{w}$:

$$y = \text{classify}(x) = \begin{cases} +1 & \text{if } h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T\mathbf{f}(\mathbf{x}) > 0 \\ -1 & \text{if } h_{\mathbf{w}}(\mathbf{x}) = \mathbf{w}^T\mathbf{f}(\mathbf{x}) < 0 \end{cases}$$

(b) Compare the predicted label y to the true label y^*:

- If $y = y^*$, do nothing
- Otherwise, if $y \neq y^*$, then update your weights: $\mathbf{w} \leftarrow \mathbf{w} + y^*\mathbf{f}(\mathbf{x})$

3. If you went through **every** training sample without having to update your weights (all samples predicted correctly), then terminate. Else, repeat step 2

Updating weights

Let's examine and justify the procedure for updating our weights. Recall that in step 2b above, when our classifier is right, nothing changes. But when our classifier is wrong, the weight vector is updated as follows:

$$\mathbf{w} \leftarrow \mathbf{w} + y^*\mathbf{f}(\mathbf{x})$$

where y^* is the true label, which is either 1 or -1, and $\mathbf{x}$ is the training sample which we mis-classified. You can interpret this update rule to be:

Case 1 : mis-classified positive as negative	$\mathbf{w} \leftarrow \mathbf{w} + \mathbf{f}(\mathbf{x})$
Case 2 : mis-classified negative as positve	$\mathbf{w} \leftarrow \mathbf{w} - \mathbf{f}(\mathbf{x})$

Why does this work? One way to look at this is to see it as a balancing act. Mis-classification happens either when the activation for a training sample is much smaller than it should be (causes a Case 1 misclassification) or much larger than it should be (causes a Case 2 misclassification).

Consider Case 1, where activation is negative when it should be positive. In other words, the activation is too small. How we adjust $\mathbf{w}$ should strive to fix that and make the activation larger for that training sample. To convince yourself that our update rule $\mathbf{w} \leftarrow \mathbf{w} + \mathbf{f}(\mathbf{x})$ does that, let us update $\mathbf{w}$ and see how the activation changes.

$$h_{\mathbf{w}+\mathbf{f}(\mathbf{x})}(\mathbf{x}) = (\mathbf{w} + \mathbf{f}(x))^T\mathbf{f}(\mathbf{x}) = \mathbf{w}^T\mathbf{f}(\mathbf{x}) + \mathbf{f}(\mathbf{x})^T\mathbf{f}(\mathbf{x}) = h_{\mathbf{w}}(\mathbf{x}) + \mathbf{f}(\mathbf{x})^T\mathbf{f}(\mathbf{x})$$

Using our update rule, we see that the new activation increases by $\mathbf{f}(\mathbf{x})^T\mathbf{f}(\mathbf{x})$, which is a postive number, therefore showing that our update makes sense. Activation is getting larger– closer to becoming positive. You can repeat the same logic for when the classifier is mis-classifying because the activation is too large (activation is positive when it should be negative). You'll see that the update will cause the new activation to decrease by $\mathbf{f}(\mathbf{x})^T\mathbf{f}(\mathbf{x})$, thus getting smaller and closer to classifying correctly.

While this makes it clear why we are adding and subtracting *something*, why would we want to add and subtract our sample point's features? A good way to think about it, is that the weights aren't the only thing that determines this score. The score is determined by multiplying the weights by the relevant sample. This means that certain parts of a sample contribute more than others. Consider the following situation where x is a training sample we are given with true label $y^* = -1$:

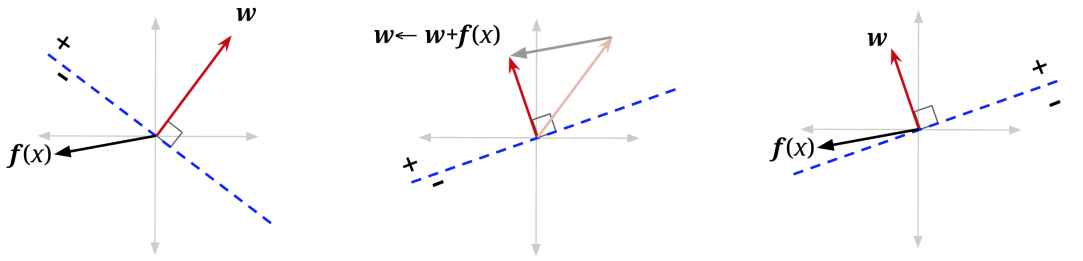
$$\mathbf{w}^T = \begin{bmatrix}2 & 2 & 2\end{bmatrix}, \mathbf{f}(\mathbf{x}) = \begin{bmatrix}4\\0\\1\end{bmatrix} \qquad h_{\mathbf{w}}(\mathbf{x}) = (2*4) + (2*0) + (2*1) = 10$$

We know that our weights need to be smaller because activation needs to be negative to classify correctly. We don't want to change them all the same amount though. You'll notice that the first element of our sample, the 4, contributed much more to our score of 10 than the third element, and that the second element did not

contribute at all. An appropriate weight update, then, would change the first weight alot, the third weight a little, and the second weight should not be changed at all. After all, the second and third weights might not even be that broken, and we don't fix want to fix what isn't broken!

When thinking about a good way to change our weight vector in order to fulfill the above desires, it turns out just using the sample itself does in fact do what we want; it changes the first weight by a lot, the third weight by a little, and doesn't change the second weight at all!

A visualization may also help. In the figure below, $\mathbf{f}(\mathbf{x})$ is the feature vector for a data point with positive class ($y^* = +1$) that is currently misclassified – it lies on the wrong side of the decision boundary defined by "old $\mathbf{w}$". Adding it to the weight vector produces a new weight vector which has a smaller angle to $\mathbf{f}(\mathbf{x})$. It also shifts the decision boundary. In this example, it has shifted the decision boundary enough so that x will now be correctly classified (note that the mistake won't always be fixed – it depends on the size of the weight vector, and how far over the boundary $\mathbf{f}(\mathbf{x})$ currently is).



Mis-classifying x with old w **Updating w** **Updated classification of x**

Bias

If you tried to implement a perceptron based on what has been mentioned thus far, you will notice one particularly unfriendly quirk. Any decision boundary that you end up drawing will be crossing the origin. Basically, your perceptron can only produce a decision boundary that could be represented by the function $\mathbf{w}^\top \mathbf{f}(\mathbf{x}) = 0$, $\mathbf{w}, \mathbf{f}(\mathbf{x}) \in \mathbb{R}^n$. The problem is, even among problems where there is a linear decision boundary that separates the positive and negative classes in the data, that boundary may not go through the origin, and we want to be able to draw those lines.

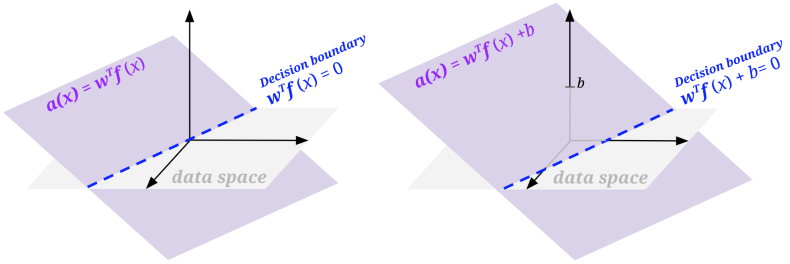
To do so, we will modify our feature and weights to add a bias term: add a feature to your sample feature vectors that is always 1, and add an extra weight for this feature to your weight vector. Doing so essentially allows us to produce a decision boundary representable by $\mathbf{w}^\top \mathbf{f}(\mathbf{x}) + b = 0$, where b is the weighted bias term (i.e. 1 * the last weight in the weight vector).

Geometrically, we can visualize this by thinking about what the activation function looks like when it is $\mathbf{w}^\top \mathbf{f}(\mathbf{x})$ and when there is a bias $\mathbf{w}^\top \mathbf{f}(\mathbf{x}) + b$. To do so, we need to be one dimension higher than the space of our featurized data (labeled data space in the figures below). In all the above sections, we had only been looking at a flat view of the data space.

Example

Let's see an example of running the perceptron algorithm step by step.

Let's run one pass through the data with the perceptron algorithm, taking each data point in order. We'll start with the weight vector $[w_0, w_1, w_2] = [-1, 0, 0]$ (where w_0 is the weight for our bias feature, which remember is always 1).



Without bias **With bias**

Training Set

#	f_1	f_2	y^*
1	1	1	-
2	3	2	+
3	2	4	+
4	3	4	+
5	2	3	-

Single Perceptron Update Pass

step	Weights	Score	Correct?	Update
1	[-1, 0, 0]	$-1\cdot 1+0\cdot 1+0\cdot 1=-1$	yes	none
2	[-1, 0, 0]	$-1\cdot 1+0\cdot 3+0\cdot 2=-1$	no	+[1, 3, 2]
3	[0, 3, 2]	$0\cdot 1+3\cdot 2+2\cdot 4=14$	yes	none
4	[0, 3, 2]	$0\cdot 1+3\cdot 3+2\cdot 4=17$	yes	none
5	[0, 3, 2]	$0\cdot 1+3\cdot 2+2\cdot 3=12$	no	-[1, 2, 3]
6	[-1, 1, -1]			

We'll stop here, but in actuality this algorithm would run for many more passes through the data before all the data points are classified correctly in a single pass.

Multiclass Perceptron

The perceptron presented above is a binary classifier, but we can extend it to account for multiple classes rather easily. The main difference is in how we set up weights and how we update said weights. For the binary case, we had one weight vector, which had a dimension equal to the number of features (plus the bias feature). For the multi-class case, we will have one weight vector for each class, so in the 3 class case, we have 3 weight vectors. In order to classify a sample, we compute a score for each class by taking the dot product of the feature vector with each of the weight vectors. Whichever class yields the highest score is the one we choose as our prediction.

For example, consider the 3-class case. Let our sample have features $\mathbf{f}(\mathbf{x}) = \begin{bmatrix}-2 & 3 & 1\end{bmatrix}$ and our weights for classes 0, 1, and 2 be:

$$\mathbf{w}_0 = \begin{bmatrix}-2 & 2 & 1\end{bmatrix}$$
$$\mathbf{w}_1 = \begin{bmatrix}0 & 3 & 4\end{bmatrix}$$
$$\mathbf{w}_2 = \begin{bmatrix}1 & 4 & -2\end{bmatrix}$$

Taking dot products for each class gives us scores $s_0 = 11, s_1 = 13, s_2 = 8$. We would thus predict that $\mathbf{x}$ belongs to class 1.

An important thing to note is that in actual implementation, we do not keep track of the weights as separate structures, we usually stack them on top of each other to create a weight matrix. This way, instead of doing as many dot products as there are classes, we can instead do a single matrix-vector multiplication. This tends to be much more efficient in practice (because matrix-vector multiplication usually has a highly optimized implementation).

In our above case, that would be:

$$\mathbf{W} = \begin{bmatrix} -2 & 2 & 1 \\ 0 & 3 & 4 \\ 1 & 4 & -2 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} -2 \\ 3 \\ 1 \end{bmatrix}$$

And our label would be:

$$\arg\max(\mathbf{W}\mathbf{x}) = \arg\max(\begin{bmatrix} 11 \\ 13 \\ 8 \end{bmatrix}) = 1$$

Along with the structure of our weights, our weight update also changes when we move to a multi-class case. If we correctly classify our data point, then do nothing just like in the binary case. If we chose incorrectly, say we chose class $y \neq y^*$, then we add the feature vector to the weight vector for the true class to y^*, and subtract the feature vector from the weight vector corresponding to the predicted class y. In our above example, let's say that the correct class was class 2, but we predicted class 1. We would now take the weight vector corresponding to class 1 and subtract x from it,

$$\mathbf{w}_1 = \begin{bmatrix} 0 & 3 & 4 \end{bmatrix} - \begin{bmatrix} -2 & 3 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 0 & 3 \end{bmatrix}$$

Next we take the weight vector corresponding to the correct class, class 2 in our case, and add x to it:

$$\mathbf{w}_2 = \begin{bmatrix} 1 & 4 & -2 \end{bmatrix} + \begin{bmatrix} -2 & 3 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 7 & -1 \end{bmatrix}$$

What this amounts to is 'rewarding' the correct weight vector, 'punishing' the misleading, incorrect weight vector, and leaving alone an other weight vectors. With the difference in the weights and weight updates taken into account, the rest of the algorithm is essentially the same; cycle through every sample point, updating weights when a mistake is make, until you stop making mistakes.

In order to incorporate a bias term, do the same as we did for binary perceptron – add an extra feature of 1 to every feature vector, and an extra weight for this feature to every class's weight vector (this amounts to adding an extra column to the matrix form).

Summary

In this note, we introduced several fundamental principles of machine learning, including:

- Splitting our data into training data, validation data, and test data.
- The difference between supervised learning, which learns from labeled data, and unsupervised learning, which doesn't have labeled data and so attempts to infer inherent structure from it.

We then proceeded to discuss an number of supervised learning algorithms such as Naive Bayes, Linear Regression, and the Perceptron Algorithm.

- We covered the Naive Bayes algorithm and derived the maximum likelihood estimates of the unknown model parameters. We extended this idea to discuss the problem of overfitting in the context of Naive Bayes' and how this issue can be mitigated with Laplace smoothing.
- We talked about Linear Regression, a simple model where we predict real-valued outputs as linear combinations of our input features. We also derived the linear regression closed form solution using vector calculus.
- Finally, we talked about linear decision boundaries and the perceptron algorithm - a method for classification that repeatedly iterates over all our data and updates weight vectors when it classifies points incorrectly.