1 Naive Bayes

Lab Objective: *Create a Naïve Bayes Classifier. Use this classifier, and Sklearn's premade classifier to make an SMS spam filter.*

About Naïve Bayes

Naïve Bayes classifiers are a family of machine learning classification methods that use Bayes' theorem to probabilistically categorize data. They are called naïve because they assume independence between the features. The main idea is to use Bayes' theorem to determine the probability that a certain data point belongs in a certain class, given the features of that data. Despite what the name may suggest, the naïve Bayes classifier is not Bayesian method. This is because naïve Bayes is based on likelihood rather than Bayesian inference.

While naïve Bayes classifiers are most easily seen as applicable in cases where the features have, ostensibly, well defined probability distributions (such as classifying sex given physical characteristics), they are applicable in many other cases. While it is generally a bad idea to assume independence naïve Bayes classifiers are still very effective, even when we can be confident there is nonzero covariance between features.

The Classifier

You are likely already familiar with Bayes' Theorem, but we will review how we can use Bayes' Theorem to construct a robust machine learning model.

Given the feature vector of a piece of data we want to classify, we want to know which of all the classes is most likely. Essentially, we want to answer the following question

$$\text{argmax}_{k\in K} P(C = k|\mathbf{x}), \tag{1.1}$$

where C is the random variable representing the class of the data. Using Bayes' Theorem, we can reformulate this problem into something that is actually computable. We find that for any $k \in K$ we have

$$P(C = k|\mathbf{x}) = \frac{P(C = k)P(\mathbf{x}|C = k)}{P(\mathbf{x})}.$$

Now we will examine each feature individually and use the chain rule to expand the following expression

$$\begin{aligned} P(C=k)P(\mathbf{x}|C=k) &= P(x_1,\ldots,x_n,C=k) \\ &= P(x_1|x_2,\ldots,x_n,C=k)P(x_2,\ldots,x_n,C=k) \\ &= \ldots \\ &= P(x_1|x_2,\ldots,x_n,C=k)P(x_2|x_3,\ldots,x_n,C=k)\cdots P(x_n|C=k)P(C=k), \end{aligned}$$

and applying the assumption that each feature is independent we can drastically simplify this expression to the following

$$P(x_1|x_2,\ldots,x_n,C=k)\cdots P(x_n|C=k) = \prod_{i=1}^{n} P(x_i|C=k).$$

Therefore we have that

$$P(C=k|\mathbf{x}) = \frac{P(C=k)}{P(\mathbf{x})}\prod_{i=1}^{n} P(x_i|C=k),$$

which reforms Equation 1.1 as

$$\text{argmax}_{k\in K} P(C=k)\prod_{i=1}^{n} P(x_i|C=k). \tag{1.2}$$

We drop the $P(\mathbf{x})$ in the denominator since it is not dependent on k.

This problem is approximately computable, since we can use training data to attempt to find the parameters which describe $P(x_i|C=k)$ for each i,k combination, and $P(C=k)$ for each k. In reality, a naïve Bayes classifier won't often find the actual correct parameters for each distribution, but in practice the model does well enough to be robust. Something to note here is that we are actually computing $P(C=k|\mathbf{x})$ by finding $P(C=k,\mathbf{x})$. This means that naïve Bayes is a generative classifier, and not a discriminative classifier.

Spam Filters

A spam filter is just a special case of a classifier with two classes: spam and not spam (or ham). We can now describe in more detail how we are to calculate Equation 1.2 given that we know what the features are. We can use a labeled training set to determine $P(C=\text{spam})$ the probability of spam and $P(C=\text{ham})$ the probability of ham. To do this we will assume that the training set is a representative sample and define

$$P(C=\text{spam}) = \frac{N_{\text{spam}}}{N_{\text{samples}}}, \tag{1.3}$$

and

$$P(C=\text{ham}) = \frac{N_{\text{ham}}}{N_{\text{samples}}}. \tag{1.4}$$

Using a bag of words model, we can create a simple representation of $P(x_i|C=k)$ where x_i is the i^{th} word in a message, and therefore $\mathbf{x}$ is the entire message. This results in the simple definition of

$$P(x_i|C=k) = \frac{N_{\text{occurrences of } x_i \text{ in class } k}}{N_{\text{words in class } k}}. \tag{1.5}$$

Note that the denominator in Equation 1.5 is not the number of unique words in class k, but the total number of occurrences of any word in class k. In the case we have some word x_u that is not found in the training set, we can may choose $P(x_u|C=k)$ so that the computation is not effected, i.e. letting $P(x_u|C=k)=1$ for unique words.

A First Model

When building a naïve Bayes classifier we need to choose what probability distribution we believe our features to have. For this first model, we will assume that the words are a categorically distributed random variable. This means the random variable may take on say N different values, each value has a certain probability of occurring. This distribution can be thought of as a Bernoulli trial with N outcomes instead of 2.

In our situation we may have N different words which we expect may occur in a spam or ham message, so we need to use the training data to find each word and its associated probability. In order to do this we will make a DataFrame that will allow us to calculate the probability of the occurrence of a certain word x_i based on what percentage of words in the training set were that word x_i. This DataFrame that will allow us to more easily compute Equation 1.5, assuming the words are categorically distributed. While we are creating this DataFrame, it will also be a good opportunity to compute Equations 1.3 and 1.4.

Throughout the lab we will use an SMS spam dataset contained in `sms_spam_collection.csv`. We will use portions of data to check our progress. This codes example makes full test and train sets, but we will provide you opportunity to check against certain subsets.

```
>>> import pandas as pd
>>> from sklearn.model_selection import train_test_split

>>> # load in the sms dataset
>>> df = pd.read_csv('sms_spam_collection.csv')

>>> # separate the data into the messages and labels
>>> X = df.Message
>>> y = df.Label

>>> # split the data into test and train sets
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=.7)
```

Training The Model

Problem 1. Create a class `NaiveBayesFilter`, with an `__init__()` method that be may empty. Add a `fit()` method which takes as arguments `X`, the training data, and `y` the training labels. In this case `X` is a `pandas.Series` containing strings that are SMS messages. For each message in `X` count the number of occurrences of each word and record this information in a DataFrame.

The final form of the DataFrame should have a column for each unique word that appears in any message of `X` as well as a `Label` column, you may also include any other columns you think you'll need. Each row of the DataFrame corresponds to a row of `X`, and records the number of occurrences of each word in a given message. The `Label` column records the label of the message. e.g., `df.loc[5,'red']` gives the number of times the word 'red' appears in message 5 (assuming that 'red' appears in any of the messages).

Save this DataFrame as `self.data`.

HINT: Ensure that the index of the DataFrame matches the index of `X` and `y`, by setting `index=X.index` (or `index=y.index`) when creating the DataFrame.

HINT: Be sure that you are counting the number of occurrences of a word and not a string, for example: when searching the string `'find it in there'` for the word `'in'`, make sure your get 1 and not 2 (becuause of the `'in'` in `'find'`). `pd.Series.str.split()`, and the `count()` methods may be helpful.

Predictions

Now that we have implemented the `fit()` method, we can begin to classify new data. We will do this with two methods, the first will be a method that calculates $P(S|\mathbf{x})$ and $P(H|\mathbf{x})$, and the other will determine the more likely of the two and assign a label. While it may seem like we should have $P(C = S|\mathbf{x}) = 1 - P(C = H|\mathbf{x})$, we do not. This would only be true if we assume the S and H are independent of $\mathbf{x}$. It is clear that we shouldn't make this assumption, because we are trying to determine the likelihood of S and H based on what $\mathbf{x}$ tells us. Therefore we must compute both $P(C = S|\mathbf{x})$ and $P(C = H|\mathbf{x})$.

Problem 2. Implement the `predict_proba()` method in your naïve Bayes classifier. This should take as an argument `X`, the data that needs to be classified. For each message `x` in `X` compute $P(S|\mathbf{x})$ and $P(H|\mathbf{x})$ using Equations 1.3, 1.4, and 1.5.

The method should return an (`Nx2`) array, where `N` is the length of `X`. The first column corresponds to $P(C = H|\mathbf{x})$, and the second to $P(C = S|\mathbf{x})$.

Problem 3. Implement the `predict()` method in your naïve Bayes classifier. This should take as an argument `X`, the data that needs to be classified. Implement equation 1.2 and return an array that classifies each message in `X`.

```
>>> # create the filter
>>> NB = NaiveBayesFilter()

>>> # fit the filter to the first 300 data points
>>> NB.fit(X[:300], y[:300])

>>> # test the predict function
>>> NB.predict(X[530:535])
array(['ham', 'spam', 'ham', 'ham', 'ham'], dtype=object)

>>> # score the filter on the last 300 data points
>>> # score will use your predict() method
>>> NB.score(X[-300:], y[-300:])
0.9233333333333333
```

Underflow

There are some issues that we encounter given this implementation. Notice that in the following example, the likelihoods for both spam and ham are 0 for each message.

```
>>> # find the likelihoods for messages 1085 and 2010
>>> NB.predict_proba(X[[1085,2010]])
array([[0., 0.],
       [0., 0.]])
```

This is because the messages are long, and thus involve the product of many numbers that are between 0 and 1. Because of this, we have encountered what is called underflow, where a number becomes so small it is not machine representable. Therefore, we should work in logspace, as to avoid inevitable underflow caused by long messages. If we take the log of Equation 1.2 have

$$\text{argmax}_{k\in K} \ln\left(P(C=k)\right) + \sum_{i=1}^{n} \ln\left(P(x_i|C=k)\right), \tag{1.6}$$

and this problem is still valid because log is monotone increasing.

Problem 4. Implement `predict_log_proba()` and `predict_log()` using equation 1.6.
Notice how `X[[1085,2010]]` is now classifiable.

Optimizing the Model

As you may have noticed that while the DataFrame model is rather quick to train, it is very slow to classify. This is because of the in depth lookup that must be done for every word in every message of the testing data. While there are some ways to speed up this lookup process, like finding repeated words in an unclassified message, ultimately looking up the word frequencies and summing them is expensive. What we can do instead is do these lookups ahead of time. This will result in a DataFrame that is significantly smaller (of size $2 \times N_{\text{vocabulary}}$) and computation time that is around 50 times faster.

Problem 5. Implement the two following optimizations to the DataFrame filter

- Reduce the lookup time by altering the DataFrame created in the `fit()` method. The new DataFrame will have two rows, and $N_{\text{vocabulary}}$ columns, with `'spam'` and `'ham'` being the index. Each entry will be the number of times a word appears in spam or ham messages. E.g. `self.data.loc['ham','red']` is the number of times the word "red" appeard in ham messages. Alter the `predict_proba()` and `predict_log_proba()` to appropriately utilize this improvement.
- If not already implemented during Problem 2, instead of computing $\prod_{i=1}^{n} P(x_i|C)$ for a message with n words, find $\prod_{i=1}^{l} P(x_i|C)^{n_i}$ where l is the number of unique words in the message and n_i is number of times the i^{th} word occurs. (Since $P(x_i|C)$ is the same for any word that is repeated in a message, the lookup should only need to be done once.)

```
>>> # checkout what the new DataFrame looks like after the changes
>>> NB = NaiveBayesFilter()
>>> NB.fit(X[:300], y[:300])
>>> NB.data.loc['ham','i']
184
>>> NB.data.loc['spam','i']
4
```

The Poisson Model

Now that we've examine one way to constructing a naïve Bayes classifier, let us look at one more method. In the Poisson model we assume that each word is Poisson random variable, occurring with potentially different frequencies among spam and ham messages. Because each of the messages is a different length, we can reparameterize the Poisson PMF to the following

$$P(n_i = x) = \frac{(rn)^x e^{-rn}}{x!} \tag{1.7}$$

where n_i is the number of times word i occurs in a message, n is the length of the message, and $\lambda = rn$ is the classical Poisson rate. In this case r represents the number of events per unit time/space/etc.

While we will use maximum likelihood estimation to determine r, we could easily refactor this model to use Bayesian inference, allowing greater control over the model. This would create a condition where the training data doesn't completely determine the model's viability. I encourage you to do this refactor once you cover Bayesian inference and see how much better your results can be. (Try a beta prior with $a = 2, b = 5$)

Training the Model

Similar to the other classifier that we made, training the model amounts to using the training data to determine how $P(x_i|C = k)$ is computed, as well as compting $P(C = k)$. As stated earlier, we will attempt to find the most likely value of r for each word that appears in the training set. To do this we will use maximum likelihood estimation. The parameter we choose is the one that maximizes the likelihood function

$$\hat{r} = \text{argmax}_r \mathcal{L}(r|\mathbf{x}) = \text{argmax}_r P(\mathbf{x}|r).$$

In this case, since we are using a Poisson distribution (1.7) for each word, we will solve the following problem for both the spam class and the ham classes

$$r_{i,k} = \text{argmax}_{r\in[0,1]} \frac{(rN_k)^{n_i} e^{-rN_k}}{n_i!}, \tag{1.8}$$

where $r_{i,k}$ is the Poisson rate for word i in class k, N_k is the total number of words in class k (either spam or ham), and n_i is the number of times word i occurs in that class. We have $r \in [0, 1]$ because a word cannot occur more than once per word in the message.

Problem 6. Create a new class called `PoissonBayesFilter` with an `__init__()` method that may be empty. Add a `fit()` method which takes as arguments `X`, the training data, and `y` the training labels. Implement `fit()` by implementing the MLE algorithm found in 1.8 to predict r for each word in both the spam and ham classes, and therefore train the model. Store these computed rates in dictionaries called `self.spam_rates` and `self.ham_rates`, where the key is the word and the value is the associated r. (E.g. `self.ham_rates['i']` will give the computed r value for the word "i" in ham messages)

```
>>> #create a poisson bayes object to examine it
>>> PB = PoissonBayesFilter()
>>> PB.fit(X[:300], y[:300])

>>> # check spam and ham rate of 'i'
>>> PB.ham_rates['i']
0.04830483048304831
>>> PB.spam_rates['i']
0.0033003300330033004
```

Predictions

Making predictions with this model is exactly the same as we did earlier. To clarify the calculation, lets reformulate 1.6 to fit the Poisson case better. This gives

$$\text{argmax}_{k\in K} \ln\left(P(C=k)\right) + \sum_{i=1}^{l} \ln\left(\frac{(r_{i,k}n)^{n_i}e^{-r_i n}}{n_i!}\right), \tag{1.9}$$

with l being the number of unique words in a message, n_i the number of times the i^{th} word occurs, n the number of words in the message, and $r_{i,k}$ the Poisson rate of the i^{th} word in class k.

Problem 7. Implement the `predict_proba()` and `predict()` methods using Equation 1.9. These methods will expect the same arguments and return the same types as the previous problems. You may use `scipy.stats.poisson.pmf` if you wish.

Naive Bayes with Sklearn

Now that we have explored a few ways to implement out own naïve Bayes classifier, we can examine the tools from the sklearn library. Sklearn provides robust tools that will accomplish all the things that we've coded up so far.

First we will want to use `CountVectorizer` from `sklearn.feature_extraction.text`. This tool will essentially do the work of the first `fit()` method we wrote. The vectorizer must learn a dictionary as well as transform the training set, this can be done with the `fit_transform()` method. This will fit the vectorizer, i.e. create the dictionary, and transform the data.

Now we can use the transformed training data to fit a `MultinomialNB` model from `sklearn.naive_bayes`. Any data that we want to classify, we must first transform with the vectorizer (using the `transform()` method, not the `fit_transform()` method), then we can classify it using the `predict()` method of the `MultinomialNB` model. This naïve Bayes model uses the multinomial distribution where we have used the categorical and poisson distributions. Multinomial is very similar to the categorical implementation, as the multinomial distribution models the outcome of n categorical trials (in the same way that the binomial distribution models n Bernoulli trials).

Problem 8. Write a function that will classify messages. It will take as arguments training data `X_train` and `y_train`, and test data `y_test`. In this function use the `CountVectorizer` and `MultinomialNB` from sklearn and return the predicted classification of the model.

References

Rish, Irina. (2001). An Empirical Study of the Naïve Bayes Classifier. IJCAI 2001 Work Empir Methods Artif Intell. 3.

Data from: http://www.dt.fee.unicamp.br/∼tiago/smsspamcollection/