CS 475 Machine Learning (Spring 2017): Assignment 4

Due on April 24th, 2017 at 3:00PM

Raman Arora

Instructions: Please read these instructions carefully and follow them precisely. Feel free to ask the instructor if anything is unclear!

1. Please submit your solutions electronically via Gradescope.
2. Please submit a PDF file for the written component of your solution including derivations, explanations, etc. You can create this PDF in any way you want: typeset the solution in LATEX (recommended), type it in Word or a similar program and convert/export to PDF, or even hand write the solution (legibly!) and scan it to PDF. We recommend that you restrict your solutions to the space allocated for each problem; you may need to adjust the white space by tweaking the argument to `\vspace{xpt}` command. Please name this document <firstname-lastname>-sol4.pdf.
3. Submit the empirical component of the solution (Python code and the documentation of the experiments you are asked to run, including figures) in a Jupyter notebook file. In addition, we require that you save the Python notebook as a pdf file and append it to the rest of the solutions.
4. In addition, you will need to submit your predictions on digit recognition and sentiment classification tasks to `Kaggle`, as described below, according to the competition rules.
5. **Late submissions:** You have a total of 72 late hours for the entire semester that you may use as you deem fit. After you have used up your quota, there will be a penalty of 50% of your grade on a late homework if submitted within 48 hours of the deadline and a penalty of 100% of your grade on the homework for submissions that are later than 48 hours past the deadline.
6. **What is the required level of detail?** When asked to derive something, please clearly state the assumptions, if any, and strive for balance: justify any non-obvious steps, but try to avoid superfluous explanations. When asked to plot something, please include the figure as well as the code used to plot it. If multiple entities appear on a plot, make sure that they are clearly distinguishable (by color or style of lines and markers). When asked to provide a brief explanation or description, try to make your answers concise, but do not omit anything you believe is important. When submitting code, please make sure it's reasonably documented, and describe succinctly in the written component of the solution what is done in each `py`-file.

Name: Tony Melo

I. Generative Models

In this section, we will work with two generative models: Naïve Bayes and Mixture Models. We will explore their performance on two datasets you're already familiar with: MNIST (the large train portion from HW2 now with binarized features) for digit classification and Consumer Reviews (from HW3) for sentiment analysis.

There will be Kaggle competitions to compare the performance of Mixture Models applied separately to MNIST and Consumer Reviews, while there will be no Kaggle component for the implemented Naïve Bayes model. The same data, however, is used for both models and must be accessed through Kaggle.

Naïve Bayes

In this problem you will implement a Naïve Bayes (NB) classifier to be applied for binary (sentiment) and multi-class (digit) classification. We have not discussed Naïve Bayes classifiers in class; below is the brief overview of this approach.

A generative model, as we have seen in class, assigns the label according to a discriminant rule

$$\widehat{y}(\mathbf{x}) = \underset{c}{\operatorname{argmax}} \left\{ \log p\left(\mathbf{x} \,|\, y = c\right) + \log p(y = c) \right\}. \tag{1}$$

The model of class-conditional densities $p\left(\mathbf{x} \,|\, y = c\right)$ for each class c is typicaly parametric and fit separately to examples from each class. The model for the priors $p(c)$ is usually simply based on counting relative frequency of the classes in the training data. In the common scenario of balanced data where all C classes are represented with the same number of examples, the priors boil down to the uniform $p(c) = 1/C$ and so can be ignored since they don't contribute to the decision in (1).

A commonly used approach to regularization of such models is to limit, or penalize, the complexity of the class-conditionals. In the case of Gaussian $p\left(\mathbf{x} \,|\, y\right)$, we can consider restricting the covariance matrices for the class-specific Gaussian models. One such restriction is to limit the covariance matrix to be diagonal for each class. If $\mathbf{x} \in \mathbb{R}^d$, then this means

$$\boldsymbol{\Sigma}_c = \begin{bmatrix} \sigma_{c,1}^2 & 0 & \dots & 0 \\ 0 & \sigma_{c,2}^2 & 0 & 0 \\ \dots & \dots & \dots & \dots \\ 0 & \dots & \dots & \sigma_{c,d}^2 \end{bmatrix} \tag{2}$$

Under this restriction, the form of the Gaussian distribution for class c can be simplified:

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}_c, \boldsymbol{\Sigma}_c) = \frac{1}{(2\pi)^{d/2} |\boldsymbol{\Sigma}_c|^{1/2}} \exp\left(-\frac{1}{2} (\mathbf{x} - \boldsymbol{\mu}_c)^T \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu}) \right) \tag{3}$$

$$= \frac{1}{(2\pi)^{d/2} \prod_{j-1}^{d} \sigma_{c,j}} \exp\left(-\frac{1}{2} \sum_{j-1}^{d} (x_j - \mu_{c,j})^2 / \sigma_{c,j}^2 \right) \tag{4}$$

$$= \prod_{j-1}^{d} \frac{1}{(2\pi)^{1/2} \sigma_{c,j}} \exp\left(-\frac{1}{2} (x_j - \mu_{c,j})^2 / \sigma_{c,j}^2 \right) \tag{5}$$

where $\mu_{c,j}$ is the j-th coordinate of μ_c. So, under the diagonal covariance assumption, the joint distribution of features (dimensions) of $\mathbf{x}$ given class c is simply the product of distributions of individual features given that class. By definition, this means that the features of $\mathbf{x}$ are independent conditioned on the class.

This is the Naïve Bayes model. In its general form, the NB model for d-dimensional input examples $\mathbf{x}$ can be written as

$$\log p(\mathbf{x} \mid y = c) = \sum_{j=1}^{d} \log p(x_j \mid y = c), \tag{6}$$

where $p(x_j \mid y)$ specifies the class-conditional model for a one-dimensional (scalar) value of the j-th feature of $\mathbf{x}$ given the class.

It's called "naïve" since this conditional independence assumption is somewhat simplistic. However, it often works well in practice, both because of the regularizing effect of the independence assumption (think how many parameters does the general Gaussian model have vs. the NB Gaussian model) and because estimating class-conditional density is much easier computationally if it needs to be done per feature rather than jointly for many features.

In the following problem, we will use a Bernoulli distribution for each binary feature $p(x_j \mid y)$, where j ranges over the indices of the pixels in the input images for MNIST, or over the indices of word/bigram occurence features for the text analysis. You will need to write the code estimating the parameters of the Bernoulli distributions, and of course the NB classifier. Assume for the purpose of this assignment that the priors are equal for both/all classes, and can be ignored.

Accessible through the below Kaggle competition link is the dataset labeled **`datasets-hw4.h5`** which is an hdf5 file similar to the one used in HW2. Within the datafile the following are accessible, note, however, that functions are provided within the **`utils.py`** file and notebook to load the data once the hdf5 file is in your working directory:

```
sentiment_analysis/train/data
sentiment_analysis/train/labels
sentiment_analysis/val/data
sentiment_analysis/val/labels
sentiment_analysis/kaggle/data

mnist/train/data
mnist/train/labels
mnist/val/data
mnist/val/labels
mnist/kaggle/data
```

Here are the kaggle competition links:

- `https://inclass.kaggle.com/c/cs475-mixturemodel-consumerreviews`
- `https://inclass.kaggle.com/c/cs475-mixturemodel-mnist`

With regards to the Consumer Reviews data, your NB classifier will need to use binary features based on the occurence of bigram features (as used in HW3). With regards to the MNIST dataset, the features are (differently from in HW2) **binarized** pixel values 0 or 1. Code is provided to load the train/val data using the `load_experiment()` function in `utils.py`, and the kaggle data is loaded using the `load_data()` function.

Problem 1 [25 points] Use the sentiment and digit classification data described above for training (i.e. for estimating the parameters). Construct an NB classifier using ML estimate for Bernoulli parameters.

Please evaluate your trained models on the corresponding validation sets (for MNIST and Consumer Reviews), then report the accuracy and confusion matrix in the ipython notebook. How might you change the parameter estimation to improve this fairly "naïve" model?

To improve parameter estimation, we could use a different distribution, try and remove redundant features, or even use less data since naïve bayes does not need much data.

III. Mixture Models

In this part of the problem set we will derive and implement the EM algorithm for a discrete space, in which the observations are d-dimensional binary vectors (containing either 0 or 1). We will model distributions in this space as a mixture of multivariate Bernoulli distributions. That is,

$$p(\mathbf{x} \mid \theta, \pi) = \sum_{l=1}^{k} \pi_l p(\mathbf{x} \mid \theta_l), \tag{7}$$

where $\theta = [\theta_1, \ldots, \theta_l]$ are the parameters of the k components, and the mixing probabilities $\pi = [\pi_1, \ldots, \pi_k]^T$ are subject to $\sum_l \pi_l = 1$.

The l-th component of the mixture is parametrized by a d-dimensional vector $\theta_l = [\theta_{l1}, \ldots, \theta_{ld}]^T$. The value of θ_{lj} is the probability of 1 in the j-th coordinate in a vector drawn from this distribution. Since the dimensions are assumed to be independent, given θ_l, the conditional distribution of $\mathbf{x}$ under this component is

$$p(\mathbf{x} \mid \theta_l) = \prod_{j=1}^{d} p(x_j \mid \theta_{lj}) = \prod_{j=1}^{d} \theta_{lj}^{x_j} (1 - \theta_{lj})^{1 - x_j}. \tag{8}$$

The hidden variables here are, just in the Gaussian mixture case, the identities of the component that generated each observation. We will denote the hidden variable associated with $\mathbf{x}_i$ by z_i. The EM with the model proceeds as follows. The E-step consists of computing the posterior of the hidden variables, $\gamma_{i,c} = p(z_i = c \mid \mathbf{x}_i; \theta, \pi)$. In the M-step, we need to update the estimates of θ and π. These updates can be derived in a way similar to the derivation for the Gaussian mixture.

Problem 2 [25 points] Write the exact expression for the posterior probability

$$\gamma_{i,c} = p(z_i = c \mid \mathbf{x}_i; \theta, \pi)$$

in terms of $\mathbf{x}_1, \ldots, \mathbf{x}_N$ and the elements of θ and π.

$$\gamma_{i,c} = p(z_i = c \mid x_i, \theta, \pi) \Rightarrow E_{z \mid x, \theta, \pi}[z_{i,c}]$$

$$= \sum_{z_{i,c}} p(z_{i,c} = c \mid x_i, \theta, \pi) z_{i,c}$$

$$= \frac{\pi_k p(x_n \mid \theta_k)}{\sum_{c=1}^{K} \pi_c p(x_n \mid \theta_c)} \Rightarrow \frac{\pi_k \prod_{j=1}^{D} \theta_{k,j}^{x_{n,i}} (1 - \theta_{k,j})^{1 - x_{n,j}}}{\sum_{c=1}^{K} \pi_c \prod_{j=1}^{D} \theta_{c,j}^{x_{n,i}} (1 - \theta_{c,i})^{1 - x_{n,i}}}$$

Problem 3 [25 points] Derive the M-step updates for the multivariate Bernoulli mixture, using given $\gamma_{i,c}$ values from the preceding E-step.

$$\theta^{t+1} \leftarrow \underset{\theta}{\operatorname{argmax}}\, E_{z|x,\theta^t}[L(\theta^t)]$$

$$L(\theta^t) = P(x,z|\theta,\pi) = P(x|z,\theta,\pi)p(z|\theta,\pi)$$

$$P(x|z,\theta,\pi) = \prod_{i=1}^{N} p(x_i|z_i,\theta,\pi)$$

$$= \prod_{i=1}^{N}\prod_{c=1}^{K} p(x_i|\theta_c)^{z_{i,c}}$$

$$= \prod_{i=1}^{N}\prod_{c=1}^{K}\left(\prod_{j=1}^{D}\theta_{i,c}^{z_{i,j}}\left(1-\theta_{c,i}\right)^{1-z_{i,j}}\right)^{z_{i,c}}$$

$$P(x,z|\theta,\pi) = \prod_{i=1}^{N}\prod_{c=1}^{K}\left(\pi_c\prod_{j=1}^{D}\theta_{c,j}^{z_{i,j}}\left(1-\theta_{c,j}\right)^{1-z_{i,j}}\right)^{z_{i,c}}$$

$$L(\theta^t) = \log p(x,z|\theta,\pi)$$

$$= \sum_{i=1}^{N}\sum_{c=1}^{K} z_{i,c}\left(\log\pi_c + \sum_{j=1}^{D} x_{i,j}\log\theta_{c,i} + (1-x_{i,j})\log(1-\theta_{c,j})\right)$$

$$E_{z|x,\theta^t}[L(\theta^t)] = \sum_{i=1}^{N}\sum_{c=1}^{K} E_{z|x,\theta^t,\pi^t}[z_{i,c}]\left(\log\pi_c + \sum_{j=1}^{D} x_{i,j}\log\theta_{c,j} + (1-x_{i,j})\right.$$
$$\left.(\log(1-\theta_{c,i}))\right)$$

Maximize w.r.t θ

$$0 = \frac{\partial}{\partial\theta_{m,j}} E_{z|x,\theta^t}[L(\theta)] = \sum_{i=1}^{N} E_{z|x,\theta^t,\pi^t}[z_{i,c}]\left(\frac{x_{i,j}}{\theta_{m,j}} - \frac{1-x_{i,j}}{1-\theta_{m,j}}\right)$$

$$\Rightarrow \boxed{\theta_{m,j}^{t+1} = \frac{\sum_{i=1}^{N} x_{i,j}\, z_{i,m}^{t+1}}{\sum_{i=1}^{N} z_{i,m}^{t+1}}}$$

Now maximize w.r.t π subject to $\sum_{c=1}^{K}\pi_c = 1$

$$\Lambda(\theta,\lambda) = E_{z|x,\theta^t}[L(\theta)] + \lambda\left(\sum_{c=1}^{K}\pi_c - 1\right)$$

$$\frac{\partial}{\partial\pi_m}\Lambda = \frac{1}{\pi_m}\sum_{i=1}^{N} z_{i,m}^{t+1} + \lambda = 0 \Rightarrow \pi_m = -\frac{\sum_{i=1}^{N} z_{i,m}^{t+1}}{\lambda} \Rightarrow \frac{\partial}{\partial\lambda}\Lambda = \sum_{c=1}^{K}\pi_c - 1 = 0$$

$$\boxed{\pi_m = \textstyle\sum z_{im}^{t+1}/N}$$

This model could be applicable, for instance, in modeling distributions of consumer reviews using binary features. In the following problem, we will apply it to the sentiment analysis task we have encountered in an earlier problem set, as well as a multi-class classification objective in MNIST digit classification.

One method of using EM in a discriminative setting is to train a separate mixture model for each class. That is, in the context of digit classification, one EM would only see images of the digit 0 and a different EM would see each of the other digits 1 9. It is common for generative models to be trained separately for each label like this

$$p(\mathbf{x} \mid y = c, \boldsymbol{\theta}_c, \boldsymbol{\pi}_c) = \sum_{l=1}^{k} \pi_{cl} p(\mathbf{x} \mid \boldsymbol{\theta}_{cl}), \quad \forall c \tag{9}$$

Once these are trained, a discriminative model emerges naturally by selecting the class with maximum likelihood for a given sample

$$\widehat{y} = \underset{c}{\operatorname{argmax}}\, p(\mathbf{x} \mid y = c, \boldsymbol{\theta}_c, \boldsymbol{\pi}_c). \tag{10}$$

A common simplification, which you may use if you want, is to assume that one component will dominate the likelihood and thus compare the likelihoods separately for each component

$$\widehat{y} = \underset{c}{\operatorname{argmax}} \max_{l} p(\mathbf{x} \mid \boldsymbol{\theta}_{cl}). \tag{11}$$

The mixing weights are in this case ignored. Note that this is equivalent to viewing all the mixtures models as a single hierarchical mixture model and then selecting the leaf component with the maximum likelihood.

Problem 4 [25 points] Implement a classifier for the aforementioned tasks based on multiple Bernoulli mixture models as described above. A new Kaggle competition has been set up, and the data as well as further descriptions are available at this link (note: this is the same link as above copied again for your convenience):

- `https://inclass.kaggle.com/c/cs475-mixturemodel-consumerreviews`
- `https://inclass.kaggle.com/c/cs475-mixturemodel-mnist`

A few practical considerations when implementing EM:

- The EM algorithm is sensitive to initialization. You may find that randomly setting your weights will not give good results. A simple method that works well in practice is to randomly assign samples to a component and then calculating the corresponding means of those clusters. This may seem strange, since all components will start very close to the population mean. However, it works well in practice.
- When your means ($\boldsymbol{\theta}$) get close to 0 or 1, it can cause overfitting and instability. One way of solving this problem is by deciding on an $\epsilon \in (0, \frac{1}{2})$ and each time you update your means make sure that $\theta \in [\epsilon, 1 - \epsilon]$. If they lie outside of this range, you simply snap them to the closest boundary value. You will have decide an appropriate value for ϵ.

- You will need calculations of the general form

$$f(\mathbf{x}) = \log \sum_i \exp(x_i)$$

If the values of $\mathbf{x}$ have large absolute values, you will run into numerical underflow or overflow as $\exp(x_i)$ approach zero or infinity. In our case, we are worried about underflow, since we are dealing with events of very low probability. A common trick is to find an appropriate constant C and realize that the following is mathematically (but not numerically) equivalent

$$f(\mathbf{x}) = C + \log \sum_i \exp(x_i - C),$$

where typically $C = \max(\mathbf{x})$. There is a function in Python that does this: `scipy.misc.logsumexp`

If you find it difficult to debug your EM, a suggestion is to run it on MNIST first. This way, you will be able to visually inspect your components. If it works correctly, your components should look like prototypical digits.

PS4-generative-models

April 26, 2017

In [1]:
```
from __future__ import division
from ps4_utils import load_data,load_experiment
from ps4_utils import AbstractGenerativeModel
from ps4_utils import save_submission
from scipy.misc import logsumexp
import numpy as np
data_fn = "datasets-hw4.h5"
MAX_OUTER_ITER = 15
```

In [2]:
```
class MixtureModel(AbstractGenerativeModel):
    def __init__(self, CLASSES, NUM_FEATURES, NUM_MIXTURE_COMPONENTS, MAX_I
        AbstractGenerativeModel.__init__(self, CLASSES, NUM_FEATURES)
        self.num_mixture_components = NUM_MIXTURE_COMPONENTS # list of num_
        self.max_iter = MAX_ITER # max iterations of EM
        self.epsilon = EPS # help with stability, to be used according to h
        self.params = { # lists of length CLASSES
            'pi': [np.repeat(1/k,k) for k in self.num_mixture_components],
            'theta': [np.zeros((self.num_features,k)) for k in self.num_mix
        }
    def pack_params(self, X, class_idx):
        pi,theta = self.fit(X[class_idx],class_idx) # fit parameters
        self.params['pi'][class_idx] = pi # update member variable pi
        self.params['theta'][class_idx] = theta #update member variable the

    #make classification based on which mixture model gives higher probabil
    def classify(self, X):
        P = list()
        pi = self.params['pi']
        theta = self.params['theta']
        for c in range(self.num_classes):
            _,Pc = self.findP(X, pi[c], theta[c])
            P.append(Pc)
        return np.vstack(P).T.argmax(-1) # np.array of class predictions fo

    # --- E-step
    def updateLatentPosterior(self, X, pi, theta, num_mixture_components):
        # YOUR CODE HERE
```

```
        # --- gamma: responsibilities (probabilities), np.array (matrix)
        # ---        shape: number of data points in X (where X consists of
        # note: can use output of findP here (with care taken to return gam
        t, sumlogt = self.findP(X, pi, theta)
        gamma = np.zeros((X.shape[0], num_mixture_components))
        for i in range(X.shape[0]):
            for c in range(num_mixture_components):
                log_gamma = t[i,c]-(sumlogt[i])
                gamma[i,c] = np.exp(log_gamma)

        return gamma
    # --- M-step (1)
    @staticmethod
    def updatePi(gamma): #update the pi component using the posteriors (gam
        # YOUR CODE HERE
        # --- pi_c: class specific pi, np.array (vector)
        # ---        shape: NUM_MIXTURE_COMPONENTS[c]
        pi_c = np.sum(gamma, axis = 0) / gamma.shape[0]
        return pi_c
    # -- M-step (2)
    @staticmethod
    def updateTheta(X, gamma): #update theta component using posteriors (ga
        # YOUR CODE HERE
        # --- theta_c: class specific theta, np.array matrix
        # ---        shape: NUM_FEATURES by NUM_MIXTURE_COMPONENTS[c]
        theta_c = np.dot(X.T, gamma) / np.sum(gamma, axis = 0)
        eps = 10**(-7)
        for i in range(theta_c.shape[0]):
            for j in range(theta_c.shape[1]):
                if theta_c[i][j] < eps:
                    theta_c[i][j] = eps
                elif theta_c[i][j] > 1-eps:
                    theta_c[i][j] = eps

        return theta_c

    @staticmethod
    def findP(X, pi, theta):
        # YOUR CODE HERE
        # NOTE: you can also use t as a probability, just change "logsumexp
        # --- t: logprobabilities of x given each component of mixture
        # ---        shape: number of data points in X (where X consists of
        # --- logsumexp(t,axis=1): (for convenience) once exponentiated, gi
        # ---        shape: number of data points in X (where X consists of
        eps = 10**(-7)
        t = np.dot(X, np.log(theta + eps)) + np.dot((1 - X), np.log(1 - the
        return t,logsumexp(t,axis=1)
```

```
        # --- execute EM procedure
        def fit(self, X, class_idx):
            max_iter = self.max_iter
            eps = self.epsilon
            N = X.shape[0]
            pi = self.params['pi'][class_idx]
            theta = self.params['theta'][class_idx]
            num_mixture_components = self.num_mixture_components[class_idx]
            # INITIALIZE theta, note theta is currently set to zeros but needs
            for i in range(num_mixture_components):
                theta[:,i] = np.sum(X[range(i,N,num_mixture_components),:], axi


            for i in range(theta.shape[0]):
                for j in range(theta.shape[1]):
                    if theta[i][j] < self.epsilon:
                        theta[i][j] = self.epsilon
                    elif theta[i][j] > 1 - self.epsilon:
                        theta[i][j] = 1 - self.epsilon

            for i in range(max_iter):
                gamma = self.updateLatentPosterior(X, pi, theta, num_mixture_co
                pi = self.updatePi(gamma)
                theta = self.updateTheta(X, gamma)


            return pi,theta #pi and theta, given class_idx

In [3]: class NaiveBayesModel(AbstractGenerativeModel):
        def __init__(self, CLASSES, NUM_FEATURES, EPS=10**(-12)):
            AbstractGenerativeModel.__init__(self, CLASSES, NUM_FEATURES)
            self.epsilon = EPS # help with stability
            self.params = {
                'p': [np.zeros((NUM_FEATURES))] * self.num_classes # estimated
            }
        def pack_params(self, X, class_idx):
            p = self.fit(X[class_idx])
            self.params['p'][class_idx] = p
        def classify(self, X): # naive bayes classifier
            # YOUR CODE HERE
            # --- predictions: predictions for data points in X (where X consis
            # ---         shape: number of data points
            pred = []
            for i in range(X.shape[0]):
                p_x_i = [0]*len(self.params['p'])
                x_i = X[i,:]
                for c, p_c in enumerate(self.params['p']):
                    for j in range(X.shape[1]):
```

```
                        p_x_i[c] += np.log((x_i[j] * p_c[j] + (1 - x_i[j]) * (1
                    pred.append(np.argmax(p_x_i))

                pred = np.array(pred)
                return pred
            def fit(self, X):
                # YOUR CODE HERE
                # --- estimated_p: estimated p's of features for input X (where X c
                # ---         shape: NUM_FEATURES
                estimated_p = np.sum(X, axis = 0) / X.shape[0]
                return np.array(estimated_p)
```

In [4]:
```
experiment_name = "sentiment_analysis"
# --- SENTIMENT ANALYSIS setup
Xtrain,Xval,num_classes,num_features = load_experiment(data_fn, experiment_

# -- build naive bayes model for sentiment analysis
print("SENTIMENT ANALYSIS -- NAIVE BAYES MODEL:")
nbm = NaiveBayesModel(num_classes, num_features)
nbm.train(Xtrain)
print("ACCURACY ON VALIDATION: " + str(nbm.val(Xval)))

# -- build mixture model for sentiment analysis
print("SENTIMENT ANALYSIS -- MIXTURE MODEL:")
for i in range(MAX_OUTER_ITER):
    num_mixture_components =  np.random.randint(2,15,num_classes)
    print("COMPONENTS: " + " ".join(str(i) for i in num_mixture_components)
    mm = MixtureModel(num_classes, num_features, num_mixture_components)
    mm.train(Xtrain)
    print("ACCURACY ON VALIDATION: " + str(mm.val(Xval)))

# submit to kaggle
Xkaggle = load_data(data_fn, experiment_name, "kaggle")
save_submission("mm-{}-submission.csv".format(experiment_name), mm.classify
```

```
SENTIMENT ANALYSIS -- NAIVE BAYES MODEL:
ACCURACY ON VALIDATION: 0.74
SENTIMENT ANALYSIS -- MIXTURE MODEL:
COMPONENTS: 3 2
ACCURACY ON VALIDATION: 0.718
COMPONENTS: 12 14
ACCURACY ON VALIDATION: 0.726
COMPONENTS: 2 10
ACCURACY ON VALIDATION: 0.676
COMPONENTS: 13 7
ACCURACY ON VALIDATION: 0.712
COMPONENTS: 5 12
ACCURACY ON VALIDATION: 0.712
```

```
COMPONENTS: 12 8
ACCURACY ON VALIDATION: 0.714
COMPONENTS: 14 14
ACCURACY ON VALIDATION: 0.732
COMPONENTS: 10 6
ACCURACY ON VALIDATION: 0.728
COMPONENTS: 2 2
ACCURACY ON VALIDATION: 0.712
COMPONENTS: 8 9
ACCURACY ON VALIDATION: 0.706
COMPONENTS: 7 13
ACCURACY ON VALIDATION: 0.714
COMPONENTS: 14 2
ACCURACY ON VALIDATION: 0.696
COMPONENTS: 10 12
ACCURACY ON VALIDATION: 0.738
COMPONENTS: 4 11
ACCURACY ON VALIDATION: 0.718
COMPONENTS: 3 10
ACCURACY ON VALIDATION: 0.696
Saved: mm-sentiment_analysis-submission.csv
```

```
In [5]: experiment_name = "mnist"
        # --- MNIST DIGIT CLASSIFICATION setup
        Xtrain,Xval,num_classes,num_features = load_experiment(data_fn, experiment_

        # -- build naive bayes model for mnist digit classification
        print("MNIST DIGIT CLASSIFICATION -- NAIVE BAYES MODEL:")
        nbm = NaiveBayesModel(num_classes, num_features)
        nbm.train(Xtrain)
        print("ACCURACY ON VALIDATION: " + str(nbm.val(Xval)))

        # -- build mixture model for mnist digit classification
        print("MNIST DIGIT CLASSIFICATION -- MIXTURE MODEL:")
        for i in range(MAX_OUTER_ITER):
            num_mixture_components =  np.random.randint(2,15,num_classes)
            print("COMPONENTS: " + " ".join(str(i) for i in num_mixture_components)
            mm = MixtureModel(num_classes, num_features, num_mixture_components)
            mm.train(Xtrain)
            print("ACCURACY ON VALIDATION: " + str(mm.val(Xval)))

        # submit to kaggle
        Xkaggle = load_data(data_fn, experiment_name, "kaggle")
        save_submission("mm-{}-submission.csv".format(experiment_name), mm.classify

MNIST DIGIT CLASSIFICATION -- NAIVE BAYES MODEL:
ACCURACY ON VALIDATION: 0.7355
```

```
MNIST DIGIT CLASSIFICATION -- MIXTURE MODEL:
COMPONENTS: 9 8 4 10 14 2 7 9 9 9
ACCURACY ON VALIDATION: 0.7715
COMPONENTS: 11 3 4 5 4 9 10 5 8 6
ACCURACY ON VALIDATION: 0.776
COMPONENTS: 13 3 4 9 2 8 4 7 5 4
ACCURACY ON VALIDATION: 0.789
COMPONENTS: 7 6 5 2 12 8 7 5 5 13
ACCURACY ON VALIDATION: 0.775
COMPONENTS: 5 13 12 3 8 9 6 6 10 10
ACCURACY ON VALIDATION: 0.7775
COMPONENTS: 8 13 14 3 2 4 8 8 14 10
ACCURACY ON VALIDATION: 0.7685
COMPONENTS: 3 6 13 7 3 10 9 10 8 10
ACCURACY ON VALIDATION: 0.7915
COMPONENTS: 2 7 9 4 4 4 3 9 9 8
ACCURACY ON VALIDATION: 0.773
COMPONENTS: 14 13 2 3 7 7 2 13 10 7
ACCURACY ON VALIDATION: 0.772
COMPONENTS: 3 13 13 9 14 5 3 11 2 10
ACCURACY ON VALIDATION: 0.771
COMPONENTS: 7 10 7 9 5 4 12 12 2 3
ACCURACY ON VALIDATION: 0.7675
COMPONENTS: 3 14 12 11 14 12 13 13 4 14
ACCURACY ON VALIDATION: 0.766
COMPONENTS: 13 11 3 3 13 10 6 12 14 6
ACCURACY ON VALIDATION: 0.767
COMPONENTS: 10 7 13 8 13 9 7 7 4 8
ACCURACY ON VALIDATION: 0.779
COMPONENTS: 10 14 11 6 5 3 5 8 2 6
ACCURACY ON VALIDATION: 0.7865
Saved: mm-mnist-submission.csv
```

In [ ]:

In [ ]: