Aniket Sanjay Gujarathi
1007629979

Assignment 1

CSC 2506
February 4, 2023

Q2. Naïve Bayes, A Generative Model

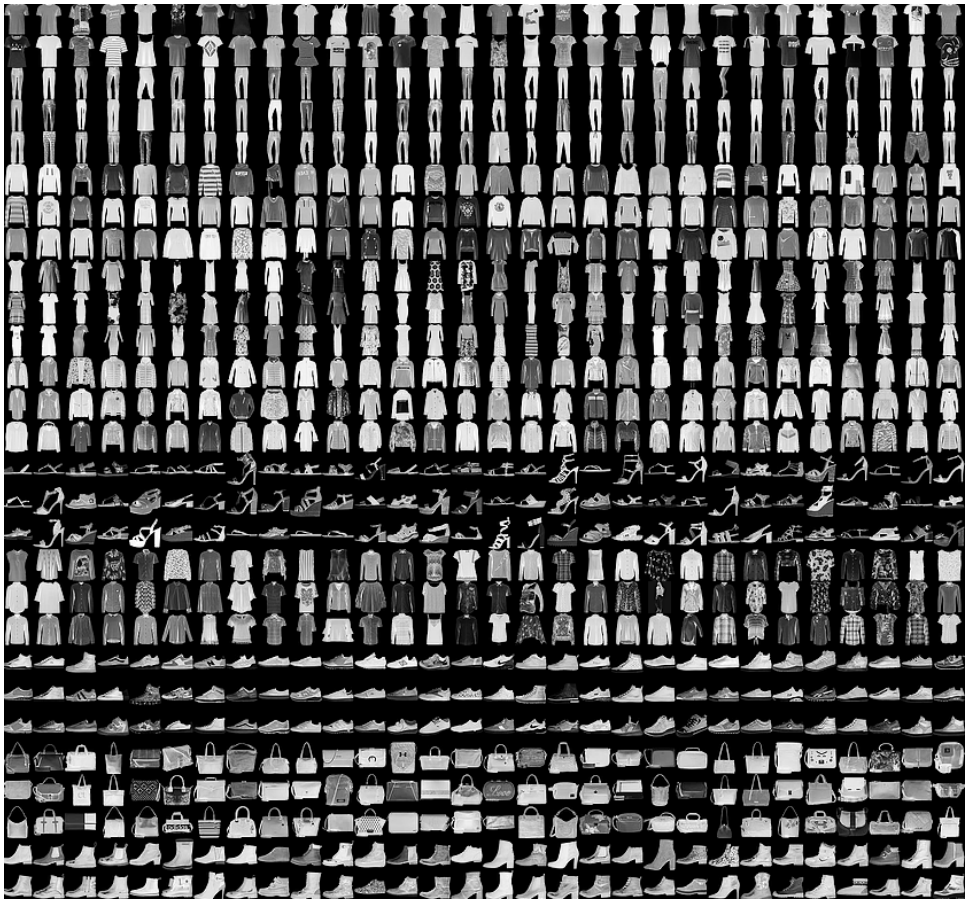

Figure 1: Naive Bayes Dataset

In this question, we'll fit a Naïve Bayes model to the fashion MNIST dataset, and use this model for making predictions and generating new images from the same distribution. Fashion MNIST is a dataset of 28x28 black-and-white images of items of clothing. We represent each image by a vector $x^{(i)} \in \{0,1\}^{784}$, where 0 and 1 represent white and black pixels respectively. Each class label $c^{(i)}$ is a different item of clothing, which in the code is represented by a 10-dimensional one-hot vector.

The Naïve Bayes model parameterized by θ and π defines the following joint probability of x and c,

$$p(x, c|\theta, \pi) = p(c|\pi)p(x|c, \theta) = p(c|\pi)\prod_{j=1}^{784} p(x_j|c, \theta),$$

where $x_j|c, \theta \sim \text{Bernoulli}(\theta_{jc})$ or in other words $p(x_j|c, \theta) = \theta_{jc}^{x_j}(1-\theta_{jc})^{1-x_j}$, and $c|\pi$ follows a simple categorical distribution, i.e. $p(c|\pi) = \pi_c$.

We begin by learning the parameters θ and π. The following code will download and prepare the training and test sets.

```
import numpy as np
import os
import gzip
import struct
import array
import matplotlib.pyplot as plt
import matplotlib.image
from urllib.request import urlretrieve

def download(url, filename):
    if not os.path.exists('data'):
        os.makedirs('data')
    out_file = os.path.join('data', filename)
    if not os.path.isfile(out_file):
        urlretrieve(url, out_file)


def fashion_mnist():
    base_url = 'http://fashion-mnist.s3-website.eu-central-1.amazonaws.com/'

    def parse_labels(filename):
        with gzip.open(filename, 'rb') as fh:
            magic, num_data = struct.unpack(">II", fh.read(8))
            return np.array(array.array("B", fh.read()), dtype=np.uint8)

    def parse_images(filename):
        with gzip.open(filename, 'rb') as fh:
            magic, num_data, rows, cols = struct.unpack(">IIII", fh.read(16))
            return np.array(array.array("B", fh.read()),
            dtype=np.uint8).reshape(num_data, rows, cols)

    for filename in ['train-images-idx3-ubyte.gz',
                     'train-labels-idx1-ubyte.gz',
                     't10k-images-idx3-ubyte.gz',
                     't10k-labels-idx1-ubyte.gz']:
        download(base_url + filename, filename)

    train_images = parse_images('data/train-images-idx3-ubyte.gz')
    train_labels = parse_labels('data/train-labels-idx1-ubyte.gz')
    test_images = parse_images('data/t10k-images-idx3-ubyte.gz')
    test_labels = parse_labels('data/t10k-labels-idx1-ubyte.gz')
    # Remove the data point that cause log(0)
    remove = (14926, 20348, 36487, 45128, 50945, 51163, 55023)
    train_images = np.delete(train_images,remove, axis=0)
    train_labels = np.delete(train_labels, remove, axis=0)
    return train_images, train_labels, test_images[:1000], test_labels[:1000]
```

```
def load_fashion_mnist():
    partial_flatten = lambda x: np.reshape(x, (x.shape[0], np.prod(x.shape[1:])))
    one_hot = lambda x, k: np.array(x[:, None] == np.arange(k)[None, :], dtype=int)
    train_images, train_labels, test_images, test_labels = fashion_mnist()
    train_images = (partial_flatten(train_images) / 255.0 > .5).astype(float)
    test_images = (partial_flatten(test_images) / 255.0 > .5).astype(float)
    train_labels = one_hot(train_labels, 10)
    test_labels = one_hot(test_labels, 10)
    N_data = train_images.shape[0]

    return N_data, train_images, train_labels, test_images, test_labels
```

Q2.1

Derive the expression for the Maximum Likelihood Estimator (MLE) of θ and π.

Answer:

We have the joint distribution of $p(x, c|\theta, \pi)$ due to the assumption of Naive Bayes Model. Taking the log-likelihood of the joint distribution, we get

$$L(\theta, \pi) = \sum_{i=1}^{N} log(p(c|\pi)) + \sum_{i=1}^{N} \sum_{j=1}^{D} (log(p(x_j|c, \theta))$$

$$L(\theta, \pi) = \sum_{i=1}^{N} log(\pi_c^i) + \sum_{j=1}^{D} \sum_{i=1}^{N} (x_j^i log(\theta_{jc} + (1 - x_j^i) log(1 - \theta_{jc})))$$

To find the Maximum Likelihood of parameters θ and π, we take the derivative of the log-likelihood and set it to zero. We know that the first term $p(c|\pi)$ follows a categorical distribution $p(c|\pi) = \pi_c$, and the second term $p(x_j|c, \theta)$ follows a Bernoulli distribution $p(x_j|c, \theta) = \theta_{jc}^{x_j}(1 - \theta_{jc})^{1-x_j}$.

We want the $argmax_{\theta,\pi}\ log(L(\theta, \pi))$ subject to $\sum_c \pi_c = 1$

Take the derivative wrt pi:

Using the lagrange multiplier to derive π

$$\frac{\partial L(\theta,\pi)}{\partial \pi_c} + \lambda \frac{\partial \sum_c \pi_c}{\partial \pi_c} = 0$$
$$\lambda = -\sum_{i=1}^{N} \frac{1(c^i = c)}{\pi_c}$$
$$\pi_c = -\frac{\sum_{n=1}^{N} 1(c^i = c)}{\lambda}$$

Applying constraint $\sum_c \pi_c = 1 \rightarrow \lambda = -N$

$$\hat{\pi}_c = \frac{\sum_{i=1}^{N} 1(c^i = c)}{N}$$

It can be expressed as the fraction of the number of class images in the dataset and the total number of images in the dataset.

Take the derivative wrt theta :

$$\sum_{i=1}^{N} 1(c^i = c)\left(\frac{x_j^i}{\theta_{jc}} - \frac{(1 - x_j^i)}{(1 - \theta_{jc})}\right) = 0$$

$$\hat{\theta}_{jc} = \frac{\sum_i 1(x_j^i = 1 \ \& \ c^i = c)}{\sum_i 1(c^i = c)}$$

It can be expressed as the fraction of the number of times the pixel values of the images in the particular class is 1 and the number of class images in the dataset.

Q2.2

Using the MLE for this data, many entries of will be estimated to be 0, which seems extreme. So we look for another estimation method.

Assume the prior distribution of is such that the entries are i.i.d. and drawn from Beta(2,2). Derive the Maximum A Posteriori (MAP) estimator for (it has a simple final form). You can return the MLE for in your implementation. From now on, we will work with this estimator.

Answer:

The MLE solution does not include any priors and depends solely on the dataset, however, the MAP solution allows us to introduce priors in our estimation thus giving better estimates.

We know that the prior distribution of θ is such that the entries are i.i.d and drawn from Beta(2, 2).

$$p(\theta) = \frac{1}{B(\alpha, \beta)}.\theta^{\alpha-1}(1-\theta)^{\beta-1}$$

The MAP estimate can be defined as:

$$\hat{\theta}_{MAP} = argmax_\theta(p(\theta|x))$$
$$\hat{\theta}_{MAP} = argmax_\theta(p(x|\theta)p(\theta))$$
$$p(\theta|x)p(x|\theta)p(\theta)$$
$$p(\theta|x)\prod_i Bernoulli(x_i|c,\theta).Beta(\theta|\alpha,\beta)$$

Taking the logarithm we get:

$$L = \sum_i (x_j^i log(\theta_{jc}) + (1-x_j^i)log(1-\theta_{jc})) + log(Beta(\theta|\alpha,\beta))$$

We find the argmax by taking the derivative of L and setting to zero. For the first term we get:

$$\frac{\partial}{\partial\theta} log(Bernoulli(x_j|\theta)) = \sum_i^N \left(\frac{x_j^i}{\theta_{jc}} - \frac{1-x_j^i}{1-\theta_{jc}}\right)$$

For the second term:

$$\frac{\partial}{\partial\theta} log(Beta(\theta|\alpha,\beta)) = \frac{\alpha-1}{\theta} - \frac{\beta-1}{1-\theta}$$

To find the argmax, we set the derivative of L to zero:

$$0 = \sum_i^N \left(\frac{x_j^i}{\theta_{jc}} - \frac{1-x_j^i}{1-\theta_{jc}}\right) + \frac{\alpha-1}{\theta_{jc}} - \frac{\beta-1}{1-\theta_{jc}}$$
$$\hat{\theta}_{MAP} = \frac{n_b + \alpha - 1}{n + \alpha + \beta - 2}$$

where $n_b = \sum_i 1[x_j^i = 1 \ \& \ c^i = c]$, $n = \sum_i 1[c^i = c]$

```
def train_map_estimator(train_images, train_labels):
    """
    Inputs: train_images (N_samples x N_features), train_labels
    (N_samples x N_classes)
    Returns the MAP estimator theta_est (N_features x N_classes) and the MLE
```

```
    estimator pi_est (N_classes)
    """

    # YOU NEED TO WRITE THIS PART
    num_images = len(train_images)
    num_features = len(train_images[0])
    num_classes = len(train_labels[0])

    # pi_est = N_c / N
    # find how many times the class has occurred in the train_images
    pi_est = []

    # keeps track of the indices of images belonging to which class
    class_count = {label:0 for label in range(num_classes)}

    # theta_est = (N_p + alpha - 1) / (N + alpha + beta - 2) = (N_p + 1) / (N + 2).
    # (number pixel j=1 appears in a class + 1) / (number of that class in dataset + 2)

    theta_est = np.zeros((num_features, num_classes))
    for label in range(num_classes):
      class_count[label] = np.where(np.where(train_labels == 1)[1] == label)[0]
      n_c = len(class_count[label])
      pi_est.append(n_c / num_images)

      n_b = np.count_nonzero(train_images[class_count[label]] == 1, axis=0)
      theta_est[:, label] = ((n_b + 1) / (len(class_count[label]) + 2))

    return theta_est, pi_est
```

2.3

Derive an expression for the class log-likelihood $\log p(c|x, \theta, \pi)$ for a single image. Then, complete the implementation of the following functions. Recall that our prediction rule is to choose the class that maximizes the above log-likelihood, and accuracy is defined as the fraction of samples that are correctly predicted.

Report the average log-likelihood $\frac{1}{N}\sum_{i=1}^{N} \log p(c^{(i)}|x^{(i)}, \hat{\theta}, \hat{\pi})$ (where N is the number of samples) on the training test, as well the training and test errors.

Answer:

Applying Bayes rule on $p(c|x, \theta, \pi)$

$$\begin{aligned} p(c|x, \theta, \pi) &= \frac{p(x, c|\theta, \pi)}{p(x|\theta, \pi)} \\ &= \frac{p(x, c|\theta, \pi)}{\sum_c p(x|c, \theta)p(c|\pi)} \end{aligned}$$

Class log likelihood for $log(p(c|x,\theta,\pi))$ for a single image is given by:

$$log(p(c|x,\theta,\pi)) = log(\pi_c) + \sum_{j=1}^{D}(x_j^i log(\theta_{jc}) + (1-x_j^i)log(1-\theta_{jc}))-$$
$$\sum_c (log(\pi_c) + \sum_{j=1}^{D}(x_j^i log(\theta_{jc}) + (1-x_j^i)log(1-\theta_{jc})))$$

```
def log_likelihood(images, theta, pi):
    """ Inputs: images (N_samples x N_features), theta, pi
        Returns the matrix 'log_like' of loglikehoods over the input images where
        log_like[i,c] = log p (c |x^(i), theta, pi) using the estimators theta and pi.
        log_like is a matrix of (N_samples x N_classes)
    Note that log likelihood is not only for c^(i), it is for all possible c's."""

    # YOU NEED TO WRITE THIS PART
    num_samples = len(images)
    num_features = len(images[0])
    num_classes = len(pi)

    # keeps track of the indices of images belonging to which class
    class_count = {label:0 for label in range(num_classes)}

    log_like = np.zeros((num_samples, num_classes))
    for image in range(num_samples):
      theta_sum = images[image] @ np.log(theta) + (1 - images[image]) @ np.log(1 - theta)
      log_like[image] = np.log(pi) + theta_sum

    return log_like

def accuracy(log_like, labels):
    """ Inputs: matrix of log likelihoods and 1-of-K labels (N_samples x N_classes)
    Returns the accuracy based on predictions from log likelihood values"""

    # YOU NEED TO WRITE THIS PART
    target_class = np.where(labels == 1)[1]
    predicted_class = np.argmax(log_like, axis=1)

    acc = np.sum(np.equal(predicted_class, target_class)) / len(target_class)

    return acc

N_data, train_images, train_labels, test_images, test_labels = load_fashion_mnist()
theta_est, pi_est = train_map_estimator(train_images, train_labels)

loglike_train = log_likelihood(train_images, theta_est, pi_est)
avg_loglike = np.sum(loglike_train * train_labels) / N_data
```

```
train_accuracy = accuracy(loglike_train, train_labels)
loglike_test = log_likelihood(test_images, theta_est, pi_est)
test_accuracy = accuracy(loglike_test, test_labels)

print(f"Average log-likelihood for MAP is {avg_loglike:.3f}")
print(f"Training accuracy for MAP is {train_accuracy:.3f}")
print(f"Test accuracy for MAP is {test_accuracy:.3f}")
```

Average log-likelihood for MAP is -285.334
Training accuracy for MAP is 0.651
Test accuracy for MAP is 0.638

2.4

Given this model's assumptions, is it always true that any two pixels xi and xj with i are independent given c? How about after marginalizing over c? Explain your answer.

Answer:

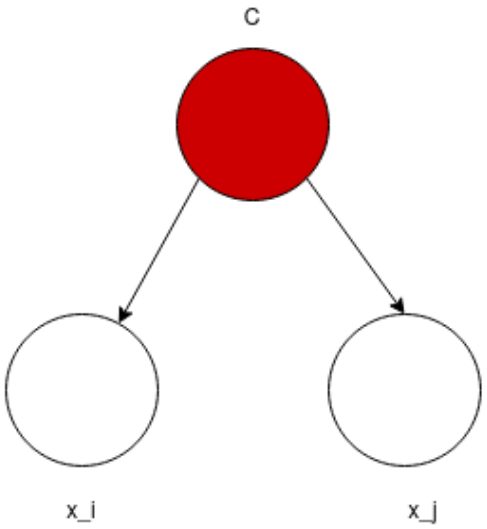

Figure 2: Graphical Model

Through Bayes ball we can prove that the two pixels x_i and x_j with $i \neq j$ are independent given c through common cause.

After marginalizing over c, the joint distribution of $p(x_i, x_j, c)$ is given by:

$$\sum_c p(x_i, x_j, c) = p(x_i, x_j)$$
$$= p(x_i|x_j)p(x_j)$$

As $p(x_i, x_j) \neq p(x_i)p(x_j)$, the two pixels are dependant when marginalized over c.

2.5

Since we have a generative model for our data, we can do more than just prediction. Randomly sample and plot 10 images from the learned distribution using the MAP estimates. (Hint: You first need to sample the class c, and then sample pixels conditioned on c.)

```
def image_sampler(theta, pi, num_images):
    """ Inputs: parameters theta and pi, and number of images to sample
    Returns the sampled images (N_images x N_features)"""

    # YOU NEED TO WRITE THIS PART

    # randomly sample the class c
    np.random.seed(5)
    random_class = []
    for i in range(num_images):
      one_hot = np.random.multinomial(1, pi)
      random_class.append(np.argmax(one_hot))

    # sample pixel values given c from p(x |c, theta, pi)
    X = [0, 1]
    image = np.zeros((num_images, len(theta[:, 0])))

    image = []
    for label in random_class:
      feature = []
      for i in range(len(theta[:, 0])):
        p_x = []
        for x in X:
          # find what the pixel value sampled from the log-likelihood
          # given the class and parameters
          p = x * np.log(theta[i, label]) + (1 - x) * np.log(1 - theta[i, label])
          p_x.append(p)
        feature.append(np.argmax(p_x))
      image.append(feature)
    return np.array(image)

def plot_images(images, ims_per_row=5, padding=5, image_dimensions=(28, 28),
                cmap=matplotlib.cm.binary, vmin=0., vmax=1.):
```

```
    """Images should be a (N_images x pixels) matrix."""
    fig = plt.figure(1)
    fig.clf()
    ax = fig.add_subplot(111)

    N_images = images.shape[0]
    N_rows = np.int32(np.ceil(float(N_images) / ims_per_row))
    pad_value = vmin
    concat_images = np.full(((image_dimensions[0] + padding) * N_rows + padding,
                             (image_dimensions[1] + padding) * ims_per_row +
                             padding), pad_value)
    for i in range(N_images):
        cur_image = np.reshape(images[i, :], image_dimensions)
        row_ix = i // ims_per_row
        col_ix = i % ims_per_row
        row_start = padding + (padding + image_dimensions[0]) * row_ix
        col_start = padding + (padding + image_dimensions[1]) * col_ix
        concat_images[row_start: row_start + image_dimensions[0],
                      col_start: col_start + image_dimensions[1]] = cur_image
        cax = ax.matshow(concat_images, cmap=cmap, vmin=vmin, vmax=vmax)
        plt.xticks(np.array([]))
        plt.yticks(np.array([]))

    plt.plot()


sampled_images = image_sampler(theta_est, pi_est, 10)
plot_images(sampled_images)
```

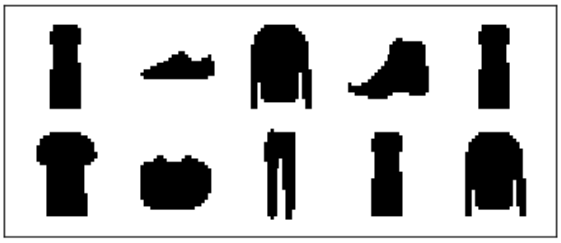

Figure 3: Generated Images

2.6

One of the advantages of generative models is that they can handle missing data, or be used to answer different sorts of questions about the model. Assume we have only observed some pixels of the image. Let $x_E = \{x_p : \text{pixel } p \text{ is observed}\}$. Derive an expression for $p(x_j|x_E, \theta, \pi)$, the conditional probability of an unobserved pixel j given the observed pixels and distribution parameters. (Hint: You have to marginalize over c.)

Answer

$$\begin{aligned} p(x_j|x_E,\theta,\pi) &= \frac{\sum_c p(x,c|\theta,\pi)}{p(x_E|\theta,\pi)} \\ &= \frac{\sum_c p(x,c|\theta,\pi)}{\sum_c p(x_E|c,\theta)p(c|\pi)} \\ &= \frac{\sum_c p(c|\pi)\prod_{j=1}^{D} p(x_j|c,\theta)p(x_E|c,\theta)}{\sum_c p(c|\pi)p(x_E|c,\theta)} \end{aligned}$$

where $p(x|c,\theta) = \theta_{jc}^{x}(1-\theta_{jc})^{1-x}$

2.7

We assume that only 30% of the pixels are observed. For the first 20 images in the training set, plot the images when the unobserved pixels are left as white, as well as the same images when the unobserved pixels are filled with the marginal probability of the pixel being 1 given the observed pixels, i.e. the value of the unobserved pixel j is $p(x_j = 1|x_E,\theta,\pi)$.

```
def probabilistic_imputer(theta, pi, original_images, is_observed):
    """Inputs: parameters theta and pi, original_images (N_images x N_features),
        and is_observed which has the same shape as original_images, with a value
        1. in every observed entry and 0. in every unobserved entry.
    Returns the new images where unobserved pixels are replaced by their
    conditional probability"""

    # YOU NEED TO WRITE THIS PART
    num_images = original_images.shape[0]
    num_features = original_images.shape[1]
    num_classes = theta.shape[1]

    # find the indices of x_E and x_j for the images
    for i in range(num_images):
      observed_inds = np.where(is_observed[i] == 1)[0]
      unobserved_inds = np.where(is_observed[i] == 0)[0]

      numerator = 0
      denominator = 0
      for label in range(num_classes):
        p_obs = 1
        # find the p(x_E | theta, pi)
        for obs_pixel in observed_inds:
          p_obs = p_obs * (theta[obs_pixel, label]**original_images[i, obs_pixel] + (1 - th

        #find the p(x_j | theta, pi)
        p_unobs = (theta[unobserved_inds, label])
```

```
            numerator += pi[label] * p_obs * p_unobs
            denominator += pi[label] * p_obs

            # replace the values of images with the prob(x_j=1 | x_E, theta, pi)
            original_images[i, unobserved_inds] = numerator / denominator

    return original_images

num_features = train_images.shape[1]
is_observed = np.random.binomial(1, p=0.3, size=(20, num_features))
plot_images(train_images[:20] * is_observed)
```

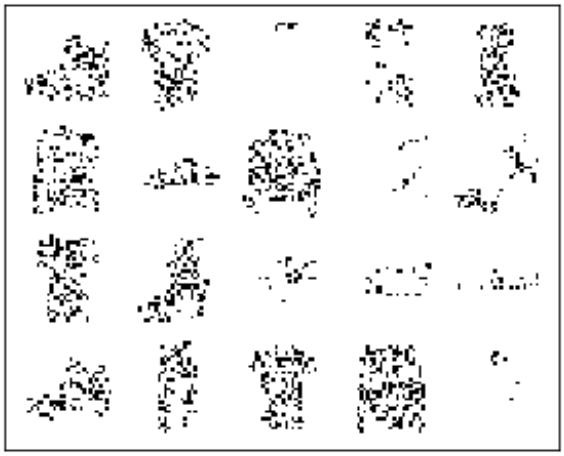

Figure 4: Unobserved pixels

```
imputed_images = probabilistic_imputer(theta_est, pi_est, train_images[:20],
is_observed)
plot_images(imputed_images)
```

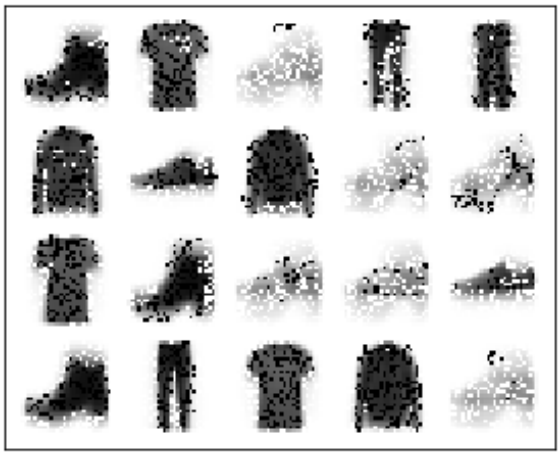

Figure 5: Filled pixels