# EECS553 HW2

Lingqi Huang

September 2024

## 1 Problem 1

*Proof.* Suppose that $B = [b_1, b_2, \ldots, b_q]$, where $b_i \in \mathbb{R}^p$ that is a $p \times 1$ dimension vector, where $1 \leq i \leq q$. Suppose the matrix $B$ is not full rank, then there exists $i, j, 1 \leq i < j \leq q$ such that $b_i = kb_j$, where $k \in \mathbb{R}$. we now have

$$B^T B = \begin{bmatrix} b_1^T \\ b_2^T \\ \vdots \\ b_i^T \\ \vdots \\ b_j^T \\ \vdots \\ b_q^T \end{bmatrix} [b_1, b_2 \cdots b_i \cdots b_j \cdots b_p]$$

Now notice that for row$i$ and row$j$ of $B^T B$, we observe that

$$\text{row}_i = [kb_j^T b_1, kb_j^T b_2, \cdots, k^2 b_j^T b_j, \cdots, kb_j^T b_j, \cdots] \qquad \text{row}_j = [b_j^T b_1, b_j^T b_2, \cdots, kb_j^T b_j, \cdots, b_j^T b_j, \cdots]$$

that row$i$ is k multiple of row$j$, which means there exists two columns that are linear dependent, thus $B^T B$ is not full rank, and so it is not invertible.
Now suppose that $B^T B$ is not invertible, then for some row$i$ and row$j$ that row$i$ can be written as $k$ multiple of row$j$, that is

$$\text{row}_i = [kb_j^T b_1, kb_j^T b_2, \cdots, k^2 b_j^T b_j, \cdots, kb_j^T b_j, \cdots] \qquad \text{row}_j = [b_j^T b_1, b_j^T b_2, \cdots, kb_j^T b_j, \cdots, b_j^T b_j, \cdots]$$

this means that for some $b_i, b_j$ of $B$, we must have $b_i = kb_j$, meaning there exists columns that are linearly dependent. This completes the proof.

□

## 2 Problem 2

*Solution*: Now set the loss function $L$ to be

$$L = \sum_{i=1}^{n} c_i (y_i - \mathbf{w}^T \mathbf{x_i} - b)^2$$

Now we set

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{n} -2c_i (y_i - \mathbf{w}^T \mathbf{x}_i - b) = 0$$

and by simplification, we will get that

$$\hat{b} = \frac{\sum_{i=1}^{n} c_i (y_i - \mathbf{w}^T \mathbf{x_i})}{\sum_{i=1}^{n} c_i}$$

Now to get $w's$, we first rewrite $\mathbf{X}\beta = \mathbf{w}^T\mathbf{x}_i + b$, where $\beta = [b, w_1, \ldots, w_p]^T$. Thus, the loss function could be rewrite as

$$L = (\mathbf{y} - \mathbf{X}\beta)^T C(\mathbf{y} - \mathbf{X}\beta)$$

where $C = diag(c_1, \ldots, c_n)$. Thus, we can easily show that

$$\nabla_\beta L = -2(\mathbf{y}^T C\mathbf{X})^T + 2\mathbf{X}^T C\mathbf{X}\beta = 0 \Rightarrow \hat{\beta} = (\mathbf{X}^T CX)^{-1}(\mathbf{X}^T C\mathbf{y})$$

# 3 Problem 3

*Proof.* In binary classification, we have that

$$P(Y = 1|X = x) = \frac{\pi_1 f_1(x)}{\pi_1 f_1(x) + \pi_0 f_0(x)}$$

Now suppose class-conditional densities are multivariate Gaussian,then we can write

$$f_1(x) = \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_1)^T\Sigma^{-1}(x - \mu_1)\right)$$

$$f_2(x) = \frac{1}{(2\pi)^{p/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(x - \mu_0)^T\Sigma^{-1}(x - \mu_0)\right)$$

Now plug $f_1(x)$ and $f_0(x)$ into $P(Y = 1|X = x)$, after some complicated algebra, we have that

$$P(Y = 1|X = x) = \frac{1}{1 + \exp\{(\mu_0^T\Sigma^{-1} - \mu_1^T\Sigma^{-1})x + \frac{1}{2}(\mu_1^T\Sigma^{-1}\mu_1 - \mu_0^T\Sigma^{-1}\mu_0) + \log\frac{\pi_0}{\pi_1}\}}$$

Thus, if we set $w^T = -(\mu_0^T\Sigma^{-1} - \mu_1^T\Sigma^{-1})$, and $b = -\frac{1}{2}(\mu_1^T\Sigma^{-1}\mu_1 - \mu_0^T\Sigma^{-1}\mu_0) + \log\frac{\pi_1}{\pi_0}$, we observed the format of logistic regression. So we proved that LDA assumption implies logistic regression assumption.

□

# 4 Problem 4

**(a)**: Denote

$$g(y) = y\log\left(\frac{1}{1 + \exp(-\theta^T\tilde{x}_i)}\right) + (1 - y)\log\left(\frac{\exp(-\theta^T\tilde{x}_i)}{1 + \exp(-\theta^T\tilde{x}_i)}\right)$$

$$f(z) = \log\left(\frac{1}{1 + \exp(-z\theta^T\tilde{x}_i)}\right)$$

We can easily observe that

$$g(0) = f(-1) = \log\left(\frac{1}{1 + \exp(\theta^T\tilde{x}_i)}\right)$$

$$g(1) = f(1) = \log\left(\frac{1}{1 + \exp(-\theta^T\tilde{x}_i)}\right)$$

Therefore, we showed that if we change label from $y \in \{0, 1\}$ to $y \in \{-1, 1\}$, we have that

$$-l(\theta) = \sum_{i=1}^{n} \log(1 + \exp(-y_i\theta^T\tilde{x}_i))$$

This completes the proof.

**(b) & (c)**: We can first write

$$J(\theta) = \sum_{i=1}^{n} \log(1 + \exp(-y_i\theta^T\tilde{x}_i)) + \lambda||w||^2$$

We denote the first part to be $J_1(\theta)$ and the second part to be $J_2(\theta)$, and we set $f(x) = \log(1 + e^x)$ and $f'(x) = \frac{e^x}{1+e^x}$.

Thus, by chain rule we have that

$$\nabla J_1(\theta) = \nabla \sum_{i=1}^{n} f(-y_i\theta^T\tilde{x_i}) = \sum_{i=1}^{n} -y_i \frac{\exp(-y_i\theta^T\tilde{x_i})}{1+\exp(-y_i\theta^T\tilde{x_i})}\tilde{x_i}$$

$$\nabla_{\mathbf{w}} J_2(\theta) = 2\lambda \begin{bmatrix} 0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix}$$

Thus, we conclude that the gradient of $J(\theta)$ is

$$\nabla J(\theta) = \sum_{i=1}^{n} -y_i \frac{\exp(-y_i\theta^T\tilde{x_i})}{1+\exp(-y_i\theta^T\tilde{x_i})}\tilde{x_i} + 2\lambda \begin{bmatrix} 0 \\ w_1 \\ \vdots \\ w_d \end{bmatrix}$$

Also, by using chain rule for taking derivative of function

$$\frac{\partial}{\partial\theta}\frac{1}{1+\exp(y_i\theta^T\tilde{x_i})} = -\frac{\exp(y_i\theta^T\tilde{x_i})}{(1+\exp(y_i\theta^T\tilde{x_i}))^2}y_i\tilde{x_i}$$

we conclude that

$$\nabla^2 J(\theta) = \sum_{i=1}^{n} \frac{\exp(y_i\theta^T\tilde{x_i})}{(1+\exp(y_i\theta^T\tilde{x_i}))^2}\tilde{x_i}\tilde{x_i}^T + 2\lambda I_{\mathbf{w}}$$

as we know $y_i^2 = 1$, and $I_{\mathbf{w}}$ represent the matrix the identity matrix with dimension of $(d+1)\times(d+1)$ but change the top left element to be 0.

**(d)**: Firstly, we use a conclusion from exercise 8 from lecture2, that summation of convex function is still convex. Now notice that for any vector $\mu$,

$$\mu^T\nabla^2 J(\theta)\mu = \sum_{i=1}^{n} \frac{\exp(-y_i\theta^T\tilde{x_i})}{(1+\exp(-y_i\theta^T\tilde{x_i}))^2}\mu^T\tilde{x_i}\tilde{x_i}^T\mu + 2\lambda\mu^T I_{\mathbf{w}}\mu$$

Notice that $\mu^T\tilde{x_i}\tilde{x_i}^T\mu \geq 0$, and the coefficient of it is always positive, and so when $\lambda \geq 0$, we must have that $2\lambda\mu^T I_w\mu \geq 0$, and so $\nabla^2 J(\theta)$ is semi-positive definite, and so $J(\theta)$ is convex. With same argument, when $\lambda > 0$, we have that $\nabla^2 J(\theta)$ is positive definite, and so $J(\theta)$ is strictly convex.

# 5 Problem 5

**(a)**: According to the code, I find that the test error is about 0.036, the number of iteration is 8, and the value of objective function is about 451.26.

```python
# Define the loss function
def loss_function(X, y, lamb, theta):
    loss = 0
    w = theta.copy()

    for i in range(X.shape[0]):
        # Compute the loss for each data point
        loss += np.log(1 + np.exp(-y[i] * np.dot(theta, X[i])))

    w[0] = 0  # Exclude the bias term from regularization

    # Add the regularization term to the loss
    loss += lamb * np.dot(w, w)

    return loss.item()

# Define the function of computing Gradient
def gradient(X, y, lamb, theta):
    grad = 0
    w = theta.copy()

    for i in range(X.shape[0]):
        grad += -y[i] * np.exp(-y[i] * np.dot(theta, X[i]))/(1 + np.exp(-y[i] * np.dot(theta, X[i]))) * X[i]

    w[0] = 0

    grad = grad + 2 * lamb * w

    return grad

# Define the function of computing Hessian
def hessian(X, y, lamb, theta):
    hess = np.zeros((785, 785))
    w = theta.copy()
# Compute the Hessian according to the formula
    for i in range(X.shape[0]):
        hess += np.exp(y[i] * np.dot(theta, X[i]))/((1 + np.exp(y[i] * np.dot(theta, X[i])))**2) * np.dot(X[i].reshape(785, 1), X[i].reshape(785, 1).T)

    # Define the matrix for w
    I_w = np.identity(785)
    I_w[0, 0] = 0

    hess = hess + 2 * lamb * I_w
    return hess
```

Figure 1: Defined Functions

```python
## Algorithm of Newton-Raphson
def optimize(X, y, lamb, theta, epsilon, max_iter):
    iter = 0
    stop = False
    loss = loss_function(X, y, lamb, theta)

    while not stop and iter < max_iter:
        grad_now = gradient(X, y, lamb, theta)
        hess_now = hessian(X, y, lamb, theta)

        theta_1 = theta - np.dot(np.linalg.inv(hess_now), grad_now)
        loss_new = loss_function(X, y, lamb, theta_1)
# If the relative error is greater than the epsilon, then stop the algorithm and return num of iteration, parameter, and loss
        if np.abs(loss_new - loss)/loss > epsilon:
            loss = loss_new
            iter = iter + 1
            theta = theta_1

        else:
            stop = True

    return iter, theta, loss
```

Figure 2: Defined algorithm

```
[262] result = optimize(X_train, y_train, 1, theta, 1e-6, 20)
```

```
[263] result
```

```
(8,
 array([-2.58806081e+00, -9.11988582e-03, -9.11988582e-03, -8.93511048e-03,
        -1.48965485e-02,  7.38205647e-02,  8.29722054e-02,  1.07957483e-02,
        -8.11671865e-02, -1.62742719e-01, -3.45216747e-01, -7.56742306e-01,
        -2.94121536e-02, -2.39173836e-01,  1.10497863e+00,  9.65350711e-01,
         2.44069231e-01, -1.17701870e-01, -5.95029136e-01, -7.03733257e-01,
         1.45356555e-01, -1.77060657e-01, -1.89297231e-01,  5.69168538e-02,
         7.87684365e-02, -1.78875295e-02, -1.80059721e-02, -9.08129851e-03,
        -9.11988582e-03, -7.56699123e-03, -6.91006962e-03, -6.58837105e-03,
        -1.08277126e-02,  5.89575564e-02,  8.91282604e-03, -6.52544064e-02,
        -1.23278004e-01, -4.46551020e-01, -6.10190336e-01, -2.68108384e-01,
         1.66838011e-01, -1.05406754e-01,  3.67761114e-01,  2.28763179e-01,
         4.21365706e-01,  1.40957743e-01,  1.32269280e-01, -2.05844698e-01,
        -3.50433824e-01, -2.42864715e-01, -5.67848006e-01, -2.77596271e-01,
         5.10699374e-02, -2.07684409e-02, -1.08892216e-02, -2.44354680e-03,
        -2.18123059e-03, -6.99699836e-03, -6.87033737e-03, -6.67115473e-03,
        -1.61773401e-02,  6.95270933e-02,  3.33451430e-02, -4.41287377e-01,
        -5.30155022e-01, -2.30026087e-01, -4.16162969e-01, -1.89665199e-01,
        -7.70076397e-01,  3.23535406e-01,  8.57114937e-01,  3.27775995e-01,
         3.28996155e-01, -1.58171108e-02, -5.68544328e-01,  4.06486261e-01,
        -1.78753160e-01,  1.56409812e-01, -3.81467052e-01, -4.25485839e-01,
        -6.64794861e-03, -2.19072709e-02, -1.41347955e-02, -1.93636620e-03,
        -2.14981379e-03, -1.05117784e-02, -6.80441368e-03, -7.24886962e-03,
         4.11599654e-03,  1.08651993e-01, -1.44294612e-01, -5.72946604e-01,
        -1.00360138e-01, -4.86100732e-01,  5.38296302e-01,  1.06323415e-01,
...
        -1.94216710e-02, -2.01835369e-01,  1.10363914e-02, -2.79813053e-01,
        -4.56679888e-01, -2.64355766e-01,  5.06118195e-01,  9.14991400e-01,
         5.48409443e-01,  1.21687272e-01, -4.00500617e-03,  6.25759425e-02,
        -3.06729172e-02]),
 451.2632670172324)
```

Figure 3: Running output

```
# Plug the theta into the logistic function
def prob_test(X, y):
    final_result = []
    for i in range(X.shape[0]):
        #Expression of logistic regression
        probs = 1/(1 + np.exp(-np.dot(theta_final, X[i])))
        final_result.append(probs)

    return final_result

def trans_prob(input_list):
    indicator = []
    for i in range(1000):
        if input_list[i] >= 0.5:
            indicator.append(1)
        else:
            indicator.append(-1)
    return indicator

# Get the final accuracy
def valid(X, Y):
    accuracy = []
    for i in range(1000):
        if X[i] == Y[i]:
            accuracy.append(1)
        else:
            accuracy.append(0)
    return sum(accuracy)/1000
[265]
```

```
# Get the probability of label 1 for test set
final_prob = prob_test(X_test, y_test)
final_label = trans_prob(final_prob)
[270]
```

```
y_test_result = y_test.ravel().tolist()
test_error = 1 - valid(final_label, y_test_result)
print("The test error is:", test_error)
[271]
```

```
The test error is: 0.03600000000000003
```

Figure 4: Value of objective function

## Part(b)

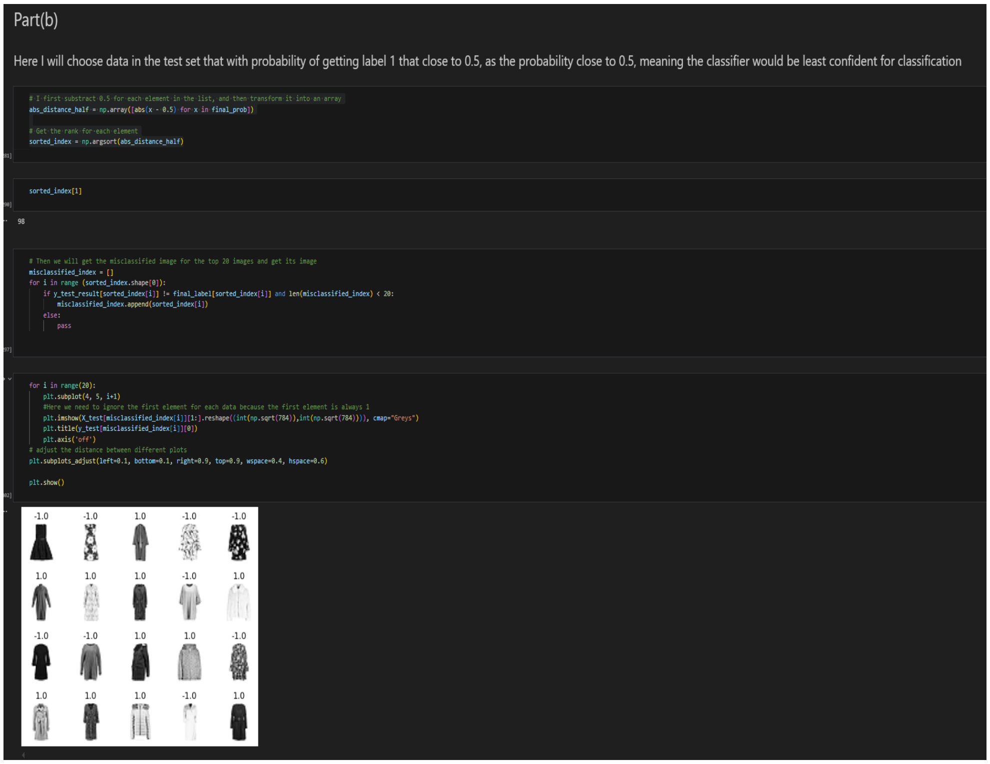


Figure 5: Code for getting top20 Misclassified image

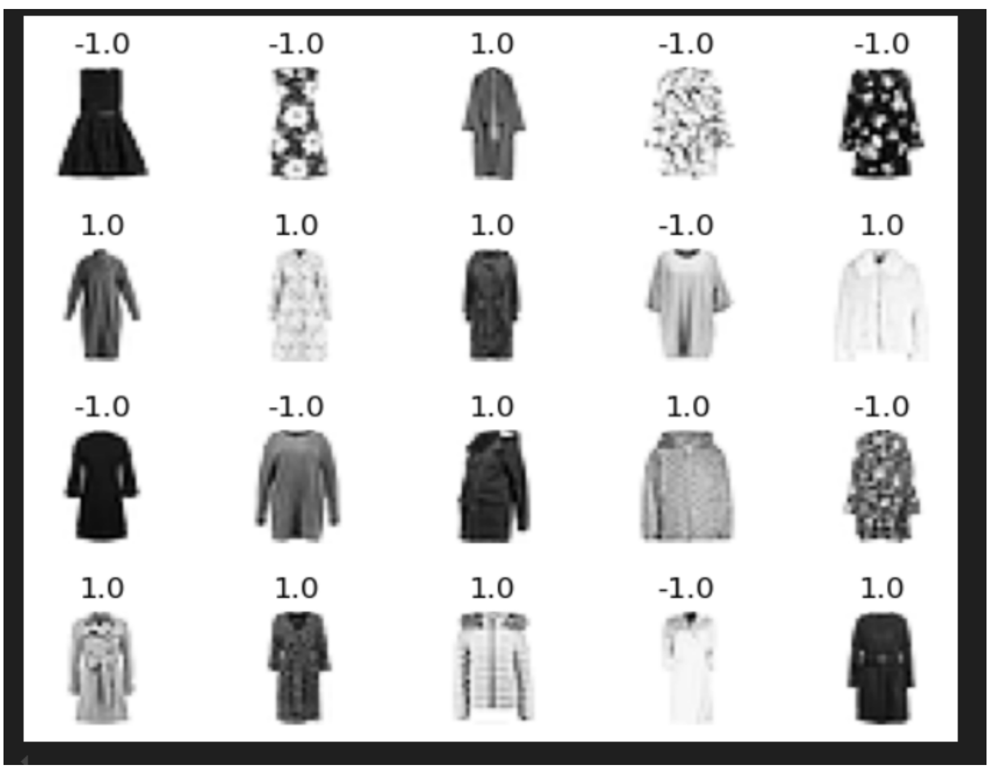


Figure 6: Top 20 misclassified image

```
import numpy as np
import matplotlib.pyplot as plt

##-----------------------------------Part(a)
x = np.load("fashion_mnist_images.npy")
y = np.load("fashion_mnist_labels.npy")
d, n= x.shape
i = 1#Index of the image to be visualized
plt.imshow(np.reshape(x[:,i], (int(np.sqrt(d)),int(np.sqrt(d)))), cmap="
    Greys")
plt.show()

# reshape y to shape = (n,1)
y = y.reshape(-1,1)

# reshape x to shape = (n,d)
x_1 = np.transpose(x)

# append 1 to get x_tilde
x_2 = np.concatenate((np.ones(n).reshape(-1,1), x_1), axis = 1)

# split the training data and test data
X_train = x_2[:5000]
X_test = x_2[5000:]
y_train = y[:5000]
y_test = y[5000:]

# initialize Theta
theta = np.zeros(785)


# Define the loss function
def loss_function(X, y, lamb, theta):
    loss = 0
    w = theta.copy()

    for i in range(X.shape[0]):
        # Compute the loss for each data point
        loss += np.log(1 + np.exp(-y[i] * np.dot(theta, X[i])))

    w[0] = 0  # Exclude the bias term from regularization

    # Add the regularization term to the loss
    loss += lamb * np.dot(w, w)

    return loss.item()

# Define the function of computing Gradient
def gradient(X, y, lamb, theta):
```

```
    grad = 0
    w = theta.copy()

    for i in range(X.shape[0]):
        grad += -y[i] * np.exp(-y[i] * np.dot(theta, X[i]))/(1 + np.exp(-y[
            i] * np.dot(theta, X[i]))) * X[i]

    w[0] = 0

    grad = grad + 2 * lamb * w

    return grad

# Define the function of computing Hessian
def hessian(X, y, lamb, theta):
    hess = np.zeros((785, 785))
    w = theta.copy()
# Compute the Hessian according to the formula
    for i in range(X.shape[0]):
        hess += np.exp(y[i] * np.dot(theta, X[i]))/((1 + np.exp(y[i] * np.
            dot(theta, X[i])))**2) * np.dot(X[i].reshape(785, 1), X[i].
            reshape(785, 1).T)

    # Define the matrix for w
    I_w = np.identity(785)
    I_w[0, 0] = 0

    hess = hess + 2 * lamb * I_w
    return hess


## Algorithm of Newton-Raphson
def optimize(X, y, lamb, theta, epsilon, max_iter):
    iter = 0
    stop = False
    loss = loss_function(X, y, lamb, theta)

    while not stop and iter < max_iter:
        grad_now = gradient(X, y, lamb, theta)
        hess_now = hessian(X, y, lamb, theta)

        theta_1 = theta - np.dot(np.linalg.inv(hess_now), grad_now)
        loss_new = loss_function(X, y, lamb, theta_1)
# If the relative error is greater than the epsilon, then stop the
    algorithm and return num of iteration, parameter, and loss
        if np.abs(loss_new - loss)/loss > epsilon:
            loss = loss_new
            iter = iter + 1
            theta = theta_1
```

```
        else:
            stop = True

    return iter, theta, loss

result = optimize(X_train, y_train, 1, theta, 1e-6, 20)

result

## Extract the parameter theta
theta_final = result[1]

# Plug the theta into the logistic function
def prob_test(X, y):
    final_result = []
    for i in range(X.shape[0]):
        #Expression of logistic regression
        probs = 1/(1 + np.exp(-np.dot(theta_final, X[i])))
        final_result.append(probs)

    return final_result

def trans_prob(input_list):
    indicator = []
    for i in range(1000):
        if input_list[i] >= 0.5:
            indicator.append(1)
        else:
            indicator.append(-1)
    return indicator

# Get the final accuracy
def valid(X, Y):
    accuracy = []
    for i in range(1000):
        if X[i] == Y[i]:
            accuracy.append(1)
        else:
            accuracy.append(0)
    return sum(accuracy)/1000


# Get the probability of label 1 for test set
final_prob = prob_test(X_test, y_test)
final_label = trans_prob(final_prob)

y_test_result = y_test.ravel().tolist()
test_error = 1 - valid(final_label, y_test_result)
print("The test error is:", test_error)
```

```
###-------------------------------------Part(b)
# I first substract 0.5 for each element in the list, and then transform it
     into an array
abs_distance_half = np.array([abs(x - 0.5) for x in final_prob])

# Get the rank for each element
sorted_index = np.argsort(abs_distance_half)


# Then we will get the misclassified image for the top 20 images and get
    its image
misclassified_index = []
for i in range (sorted_index.shape[0]):
    if y_test_result[sorted_index[i]] != final_label[sorted_index[i]] and
        len(misclassified_index) < 20:
        misclassified_index.append(sorted_index[i])
    else:
        pass


for i in range(20):
    plt.subplot(4, 5, i+1)
    #Here we need to ignore the first element for each data because the
        first element is always 1
    plt.imshow(X_test[misclassified_index[i]][1:].reshape((int(np.sqrt(784)
        ),int(np.sqrt(784)))), cmap="Greys")
    plt.title(y_test[misclassified_index[i]][0])
    plt.axis('off')
# adjust the distance between different plots
plt.subplots_adjust(left=0.1, bottom=0.1, right=0.9, top=0.9, wspace=0.4,
    hspace=0.6)

plt.show()
```