UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

Artificial Intelligence - MSc

ET5003 - MACHINE LEARNING APPLICATIONS

Instructor: Enrique Naredo

ET5003_Etivity-1

Introduction

Student_ID: 20188196
Student_full_name: Jim Gibbons
Team 3

Explanation of the problem

The problem presented is to use Bayesian multinomial logistic regression to classify images from the MNIST database of handwritten digits.

Dataset

The MNIST database is a set of images of handwritten digits. The database contains 60,000 training images and 10,000 testing images. The database is based on the NIST database. The NIST database was compiled from American Census Bureau employees and high school students. The training set comes from the American Census Bureau employees and the testing set has been taken from American high school students. As a result of the difference between the groups it was posited that this database may not be efficient for machine learning.

The MNIST database is compiled as follows;

- 50% of the training data is taken from the NIST training set.
- 50% of the training data is taken from the NIST test set.
- 50% of the testing data is taken from the NIST training set.
- 50% of the testing data is taken from the NIST test set.

The MNIST database is maintained by Yann LeCun, (Courant Institute, NYU) Corinna Cortes, (Google Labs, New York) and Christopher J.C. Burges, (Microsoft Research, Redmond).

Method

Multinomial Logistic Regression (MLR) is used to classify the images in the MNIST database.

MLR is an extension of Binary Logistic Regression (BLR) in which numerous binary models are deployed simultaneously.
Multinomial logistic regression is used to classify categorial outcomes rather than continuous outcomes.
Multinomial models do not assume normality, linearity or homoscedasticity. This makes it a strong modelling choice as real world data can often display these imperfections.

Code

Imports

In [1]:

```
# Suppressing Warnings:
import warnings
warnings.filterwarnings("ignore")
```

In [2]:

```
! pip install opencv-python
! pip install scikit-image
! pip install arviz
```

```
Requirement already satisfied: opencv-python in /home/jim/anaconda3/
envs/ET5003/lib/python3.7/site-packages (4.5.3.56)
Requirement already satisfied: numpy>=1.14.5 in /home/jim/anaconda3/
envs/ET5003/lib/python3.7/site-packages (from opencv-python) (1.19.
1)
Requirement already satisfied: scikit-image in /home/jim/anaconda3/e
nvs/ET5003/lib/python3.7/site-packages (0.16.2)
Requirement already satisfied: networkx>=2.0 in /home/jim/anaconda3/
envs/ET5003/lib/python3.7/site-packages (from scikit-image) (2.5)
Requirement already satisfied: scipy>=0.19.0 in /home/jim/anaconda3/
envs/ET5003/lib/python3.7/site-packages (from scikit-image) (1.5.2)
Requirement already satisfied: PyWavelets>=0.4.0 in /home/jim/anacon
da3/envs/ET5003/lib/python3.7/site-packages (from scikit-image) (1.
1.1)
Requirement already satisfied: imageio>=2.3.0 in /home/jim/anaconda
3/envs/ET5003/lib/python3.7/site-packages (from scikit-image) (2.9.
0)
Requirement already satisfied: pillow>=4.3.0 in /home/jim/anaconda3/
envs/ET5003/lib/python3.7/site-packages (from scikit-image) (7.2.0)
Requirement already satisfied: matplotlib!=3.0.0,>=2.0.0 in /home/ji
m/anaconda3/envs/ET5003/lib/python3.7/site-packages (from scikit-ima
ge) (3.3.1)
Requirement already satisfied: decorator>=4.3.0 in /home/jim/anacond
a3/envs/ET5003/lib/python3.7/site-packages (from networkx>=2.0->scik
it-image) (4.4.2)
Requirement already satisfied: numpy>=1.14.5 in /home/jim/anaconda3/
envs/ET5003/lib/python3.7/site-packages (from scipy>=0.19.0->scikit-
image) (1.19.1)
Requirement already satisfied: kiwisolver>=1.0.1 in /home/jim/anacon
da3/envs/ET5003/lib/python3.7/site-packages (from matplotlib!=3.0.0,
>=2.0.0->scikit-image) (1.2.0)
Requirement already satisfied: python-dateutil>=2.1 in /home/jim/ana
conda3/envs/ET5003/lib/python3.7/site-packages (from matplotlib!=3.
0.0,>=2.0.0->scikit-image) (2.8.1)
Requirement already satisfied: certifi>=2020.06.20 in /home/jim/anac
onda3/envs/ET5003/lib/python3.7/site-packages (from matplotlib!=3.0.
0,>=2.0.0->scikit-image) (2020.6.20)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.
0.3 in /home/jim/anaconda3/envs/ET5003/lib/python3.7/site-packages
(from matplotlib!=3.0.0,>=2.0.0->scikit-image) (2.4.7)
Requirement already satisfied: cycler>=0.10 in /home/jim/anaconda3/e
nvs/ET5003/lib/python3.7/site-packages (from matplotlib!=3.0.0,>=2.
0.0->scikit-image) (0.10.0)
Requirement already satisfied: six>=1.5 in /home/jim/anaconda3/envs/
ET5003/lib/python3.7/site-packages (from python-dateutil>=2.1->matpl
otlib!=3.0.0,>=2.0.0->scikit-image) (1.15.0)
Requirement already satisfied: arviz in /home/jim/anaconda3/envs/ET5
003/lib/python3.7/site-packages (0.11.2)
Requirement already satisfied: netcdf4 in /home/jim/anaconda3/envs/E
T5003/lib/python3.7/site-packages (from arviz) (1.5.7)
Requirement already satisfied: xarray>=0.16.1 in /home/jim/anaconda
3/envs/ET5003/lib/python3.7/site-packages (from arviz) (0.19.0)
Requirement already satisfied: scipy>=0.19 in /home/jim/anaconda3/en
vs/ET5003/lib/python3.7/site-packages (from arviz) (1.5.2)
Requirement already satisfied: setuptools>=38.4 in /home/jim/anacond
a3/envs/ET5003/lib/python3.7/site-packages (from arviz) (49.6.0.post
20200814)
Requirement already satisfied: matplotlib>=3.0 in /home/jim/anaconda
3/envs/ET5003/lib/python3.7/site-packages (from arviz) (3.3.1)
Requirement already satisfied: pandas>=0.23 in /home/jim/anaconda3/e
nvs/ET5003/lib/python3.7/site-packages (from arviz) (1.1.1)
```

```
Requirement already satisfied: typing-extensions<4,>=3.7.4.3 in /hom
e/jim/anaconda3/envs/ET5003/lib/python3.7/site-packages (from arviz)
(3.7.4.3)
Requirement already satisfied: packaging in /home/jim/anaconda3/env
s/ET5003/lib/python3.7/site-packages (from arviz) (20.4)
Requirement already satisfied: numpy>=1.12 in /home/jim/anaconda3/en
vs/ET5003/lib/python3.7/site-packages (from arviz) (1.19.1)
Requirement already satisfied: cftime in /home/jim/anaconda3/envs/ET
5003/lib/python3.7/site-packages (from netcdf4->arviz) (1.5.0)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.
0.3 in /home/jim/anaconda3/envs/ET5003/lib/python3.7/site-packages
(from matplotlib>=3.0->arviz) (2.4.7)
Requirement already satisfied: pillow>=6.2.0 in /home/jim/anaconda3/
envs/ET5003/lib/python3.7/site-packages (from matplotlib>=3.0->arvi
z) (7.2.0)
Requirement already satisfied: certifi>=2020.06.20 in /home/jim/anac
onda3/envs/ET5003/lib/python3.7/site-packages (from matplotlib>=3.0-
>arviz) (2020.6.20)
Requirement already satisfied: python-dateutil>=2.1 in /home/jim/ana
conda3/envs/ET5003/lib/python3.7/site-packages (from matplotlib>=3.0
->arviz) (2.8.1)
Requirement already satisfied: cycler>=0.10 in /home/jim/anaconda3/e
nvs/ET5003/lib/python3.7/site-packages (from matplotlib>=3.0->arviz)
(0.10.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /home/jim/anacon
da3/envs/ET5003/lib/python3.7/site-packages (from matplotlib>=3.0->a
rviz) (1.2.0)
Requirement already satisfied: pytz>=2017.2 in /home/jim/anaconda3/e
nvs/ET5003/lib/python3.7/site-packages (from pandas>=0.23->arviz) (2
020.1)
Requirement already satisfied: six in /home/jim/anaconda3/envs/ET500
3/lib/python3.7/site-packages (from packaging->arviz) (1.15.0)
```

In [3]:

```python
# Used to perform logistic regression, scoring, shuffle & split of dataset
#  to training & test
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split
from sklearn.utils import shuffle
# Used to fetch MNIST dataset, works locally & on Google Colab
from sklearn.datasets import fetch_openml

# Used to generate probabilistic multinomial model
import pymc3 as pm
# Used to view plots of posterior
import arviz as az
# Used for numerical operations, generating tensors for PyMC3 usage
import theano as tt
# Used for numerical operations
import numpy as np
# Used to generate random numbers to draw samples from dataset
import random
# Used in plotting images & graphs
import matplotlib.pyplot as plt

from IPython.display import HTML
%matplotlib inline
```

Load Data

In [4]:

```
mnist = fetch_openml('mnist_784', cache=False)
```

In [5]:

```
X = mnist.data.astype('float32')
y = mnist.target.astype('int64')
```

In [6]:

```
X /= 255.0
```

In [7]:

```
X.min(), X.max()
```

Out[7]:

```
(0.0, 1.0)
```

The use of `sklearn.datasets` through `fetch_openml()` to gather the MNIST dataset allows the notebook to run on both Google Colab and locally without change to the code.

Preprocessing

We split the MNIST data into a train and test set with 75% as training data and 25% as test data.

In [8]:

```
# assigning features and labels
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random
_state=42)
```

In [9]:

```
assert(X_train.shape[0] + X_test.shape[0] == mnist.data.shape[0])
```

In [10]:

```
X_train.shape, y_train.shape
```

Out[10]:

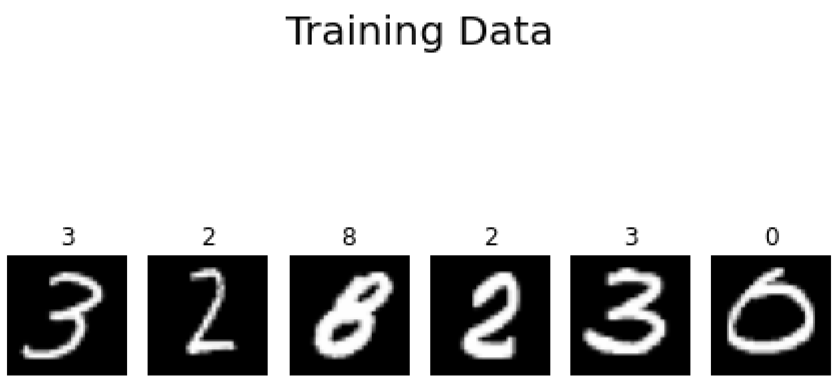
```
((52500, 784), (52500,))
```

In [11]:

```
def plot_example(X: np.array, y: np.array, n: int=5, plot_title: str=None) -> No
ne:
    """Plots the first 'n' images and their labels in a row.

    Args:
        X (numpy array): Image data with each row of array contining an image
                         as a flat vector
        y (numpy array): Image labels
        n (int): Number of images to display
        plot_title (str): Title of the plot
    Returns:
        None
    """
    fig, axs = plt.subplots(1, n)
    fig.suptitle(plot_title, fontsize=20)

    axs = axs.ravel()
    for i, (img, y) in enumerate(zip(X[:n].reshape(n, 28, 28), y[:n])):
        axs[i].axis('off')
        axs[i].imshow(img, cmap="Greys_r")
        axs[i].set_title(y)

    fig.tight_layout()
    fig.subplots_adjust(top=0.88)
    plt.show()
```

In [12]:

```
plot_example(X=X_train, y=y_train, n=6, plot_title="Training Data")
```



Building a Number Classifier from the MNIST Database

A Bayesian Multinomial Logistic Regression (BMLR) shall be built to classify the handwritten numbers in the MNIST Database.

Multinomial logistic regression is a classifiaction technique that is used to predict the category of an input or the probability of its membership to a category. This is calculated based on multiple independent variables that are either binary or continuous. Multinomial logistic regression allows for the dependent variable to be part of more than two categories (Czepiel, n.d.)(Carpita, et al., 2014).

To build the classifier we must first understand its basic construction. The formula for BMLR is:

$$Pr(Y_{ik} = Pr(Y_i = k \mid x_i; \beta_1, \beta_2, \ldots, \beta_m) = \frac{\exp(\beta_{0k} + x_i\beta_k)}{\sum\limits_{j=1}^{m} \exp(\beta_{0j} + x_i\beta_j)} \text{ with } k = 1, 2, \ldots$$

where β_k is a row vector of regression coefficients of x for the kth category of y

Since multinomial logistic regression is an expansion of binary logistic regression we will first define a binary model.

Logistic regression assumes that for a single data point (x, y):

$$P(Y = 1 \mid X = x) = \sigma(z) \text{ where } z = \theta_0 + \sum_{i=1}^{m} \theta_i x_i$$

where θ is a vector of parameters of length m the values of these parameters is found from n training examples.

This is equivalent to:
$P(Y = 1 \mid X = x) = \sigma(\theta^T x)$

Maximum likelihood estimation (MLE) is used to choose the parameter values of the logistic regression. To do this we calculate the log-likelihood and find the values of θ that maximise it.

Since the predictions being made are binary we can define each label as a Bernoulli random variable. The probability of one data point can thus be written as:

$$P(Y = y \mid X = x) = \sigma(\theta^T x)^y \cdot [1 - \sigma(\theta^T x)]^{(1-y)}$$

The likelihood of all of the data is defined as follows:

The likelihood of the independent training labels:

$$L(\theta) = \prod_{i=1}^{n} P(Y = y^{(i)} \mid X = x^{(i)})$$

Using the likelihood of a Bernoulli we get

$$L(\theta) = \prod_{i=1}^{n} P(Y = y^{(i)} \mid X = x^{(i)}) = \prod_{i=1}^{n} \sigma(\theta^T x^{(i)})^{y^{(i)}} \cdot [1 - \sigma(\theta^T x^{(i)})]^{(1-y^{(i)})}$$

Therefore the log-likelihood of the logistic regression is:

$$LL(\theta) = \sum_{(i=1)}^{n} y^{(i)} \log[\sigma(\theta^T x^{(i)}) + (1 - y^{(i)}) \log[1 - \sigma(\theta^T x^{(i)})]$$

By using a partial derivative of each parameter we can find the values of θ that maximise the log-likelihood.

The partial derivative of $LL(\theta)$ is:

$$\frac{\partial LL(\theta)}{\partial \theta_j} = \sum_{i=1}^{n} [y^{(i)} - \sigma(\theta^T x^{(i)})] x_j^{(i)}$$

Using this various optimisation techniques can be deployed to identify the maximum likelihood. A typical binary logistic regression might use gradient descent. However multinomial classifiers will likely use more sophisticated techniques (Monroe, 2017).

Classifier

Dataset Summary

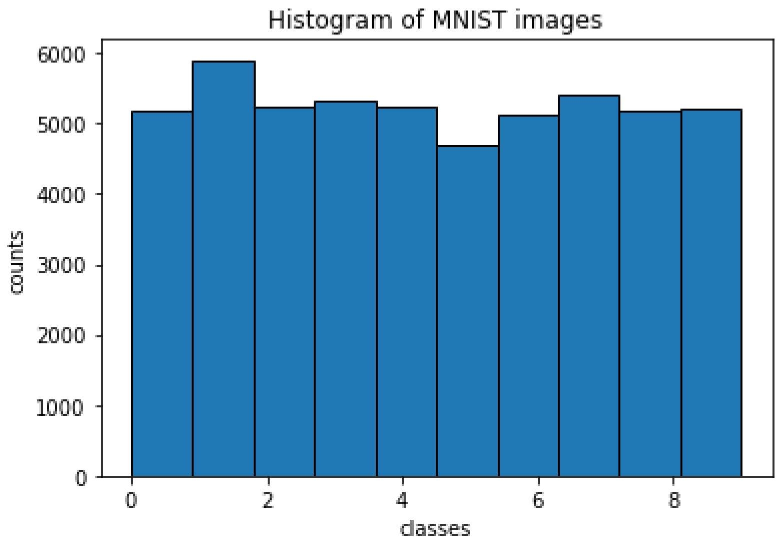
In [13]:

```
# Number of training examples
n_train =  len(X_train)

# Number of testing examples.
n_test = len(X_test)

# Shape of an MNIST image
image_shape =X_train[0].shape

# unique classes/labels in the dataset.
alltotal = set(y_train )

# number of classes
n_classes = len(alltotal )

# print information
print("Number of training examples =", n_train)
print("Number of testing examples =", n_test)
print("Image data shape =", image_shape)
print("Number of classes =", n_classes)
```

```
Number of training examples = 52500
Number of testing examples = 17500
Image data shape = (784,)
Number of classes = 10
```

In [14]:

```
## plot histogram
fig, ax = plt.subplots()
# array with evenly spaced classes
ind = np.arange(n_classes)

# histogram
n, bins, patches = ax.hist(y_train, n_classes, ec='black')
# horizontal axis label
ax.set_xlabel('classes')
# vertical axis label
ax.set_ylabel('counts')
# plot title
ax.set_title(r'Histogram of MNIST images')
# show plot
plt.figure(figsize=(10,8))
plt.show()
```



```
<Figure size 720x576 with 0 Axes>
```

We can see from the histogram that we have a relatively balanced dataset which should create good conditions for classification

Data Preparation

In [15]:

```
# Seed the run for repeatability
np.random.seed(0)

# Classes we will retain
n_classes = 3
classes = [3, 7, 9]

# The number of instances we'll keep for each of our 3 digits:
N_per_class = 500

X = []
labels = []
for d in classes:
    imgs = X_train[np.where(y_train==d)[0],:]
    X.append(imgs[np.random.permutation(imgs.shape[0]),:][0:N_per_class,:])
    labels.append(np.ones(N_per_class)*d)

X_train2 = np.vstack(X).astype(np.float64)
y_train2 = np.hstack(labels)
```

We reduce the number of classes to 3 and rather than randomly select them for each notebook run we have explicitly selected them. This allows us to to discuss findings as a group based on the same data.
We select all image indices of each desired class from `X_train` , randomly arrange them and append the first `inst_class` of them to the `inputs` array.

In [16]:

```
print(X_train2.shape,y_train2.shape)
```

```
(1500, 784) (1500,)
```

In [17]:

```
# plot digits
def plot_digits(instances, images_per_row=5, **options):
    """Plots images in rows

    Args:
        instances (numpy array): Numpy array of image data
        images_per_row (int): Number of images to print on each row
    Returns:
        None
    """
    size = 28
    images_per_row = min(len(instances), images_per_row)
    images = [instance.reshape(size,size) for instance in instances]
    n_rows = (len(instances) - 1) // images_per_row + 1
    row_images = []
    n_empty = n_rows * images_per_row - len(instances)
    images.append(np.zeros((size, size * n_empty)))
    for row in range(n_rows):
        rimages = images[row * images_per_row : (row + 1) * images_per_row]
        row_images.append(np.concatenate(rimages, axis=1))
    image = np.concatenate(row_images, axis=0)
    plt.imshow(image,  cmap='gist_yarg', **options)
    plt.axis("off")
```

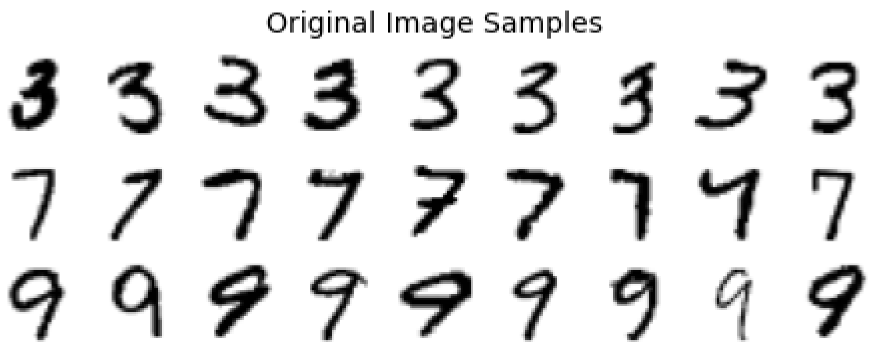
In [18]:

```
# Show random instances from each Digit:
plt.figure(figsize=(8,8))

# Selecting a few label indices from each of the 3 classes to show:
n_sample = 9
label_indices = []
for i in range(n_classes):
    label_indices += random.sample(range(i*N_per_class, (i+1)*N_per_class), n_sa
mple)

print(label_indices)
# Plotting 'original' image
plot_digits(X_train2[label_indices,:], images_per_row=9)
plt.title("Original Image Samples", fontsize=14)
```

```
[151, 489, 425, 230, 370, 187, 155, 145, 159, 674, 763, 760, 775, 61
4, 905, 582, 735, 630, 1263, 1450, 1159, 1481, 1190, 1277, 1120, 121
7, 1345]
```

Out[18]:

```
Text(0.5, 1.0, 'Original Image Samples')
```



In [19]:

```
### we split the dataset in training and validation
X_tr, X_val, y_tr, y_val = train_test_split(X_train2, y_train2, test_size=0.2, r
andom_state=0)

X_tr, y_tr = shuffle(X_tr, y_tr)

print(X_tr.shape)
print(X_val.shape)
print(y_tr.shape)
print(y_val.shape)
```

```
(1200, 784)
(300, 784)
(1200,)
(300,)
```

In [20]:

```
# transform images into vectors
X_trv = X_tr.flatten().reshape(X_tr.shape[0],X_tr.shape[1])
X_valv = X_val.flatten().reshape(X_val.shape[0],X_tr.shape[1])
print(X_trv.shape)
print(X_valv.shape)
print(y_tr.shape)
print(y_val.shape)
```

```
(1200, 784)
(300, 784)
(1200,)
(300,)
```

Given that the MNIST dataset is already in flat vector form, i.e. each image is already a one dimensional vector, the `flatten` step is not required. However we retain this step for future reference when using other datasets that may require flattening.

Algorithm

In [21]:

```
#General-recipe ML logistic regression
clf = LogisticRegression(random_state=0, max_iter=2000, C=100, solver='lbfgs', m
ulti_class='multinomial').fit(X_trv, y_tr)
y_pred_logi = clf.predict(X_valv)
y_pred_logi_prob = clf.predict_proba(X_valv)
prob_classmax = np.max(y_pred_logi_prob,axis=1)
print("Accuracy =", accuracy_score(y_pred_logi, y_val))
```

```
Accuracy = 0.92
```

Accuracy achieved of 0.92 is relatively good.
We'll review the highest probablities for correctly and incorrectly classified images.

In [22]:

```
# probability of general-recipe logistic regression in correct instances
highest_prob_matches = np.sort(prob_classmax[y_val==y_pred_logi])
print(f"Probabilities of best scoring matches:\n{highest_prob_matches}")
```

```
Probabilities of best scoring matches:
[0.52755263 0.53651543 0.61147213 0.75729678 0.84315684 0.86164767
 0.88587022 0.92341305 0.93858534 0.94589416 0.95209961 0.95804424
 0.96446475 0.97066191 0.97541154 0.97773248 0.97777823 0.97854496
 0.98038644 0.98251636 0.98474088 0.98546662 0.98731741 0.99084641
 0.99120044 0.9921186  0.99227746 0.99236169 0.99389956 0.99499082
 0.99529104 0.99585387 0.99606414 0.99622749 0.9963743  0.99659315
 0.99674926 0.99678234 0.99759062 0.99772546 0.99773171 0.99826024
 0.99869092 0.99869497 0.99892498 0.99910542 0.99923728 0.9993865
 0.99941269 0.99943435 0.99964409 0.99966471 0.9997007  0.9997175
 0.99972826 0.99977057 0.9997818  0.99982017 0.99982751 0.99984262
 0.99986223 0.99989291 0.99989366 0.99989684 0.9999031  0.99990592
 0.99990678 0.99991314 0.99991908 0.99992435 0.9999379  0.99993973
 0.99994336 0.99994713 0.99994803 0.9999542  0.99995821 0.99996428
 0.99996571 0.99996911 0.99996926 0.99997199 0.99997719 0.99997791
 0.99997853 0.99998012 0.99998334 0.99998405 0.9999859  0.99998626
 0.99998847 0.99998848 0.99998945 0.99999106 0.99999192 0.99999305
 0.99999311 0.99999341 0.99999385 0.99999558 0.99999587 0.99999626
 0.99999627 0.99999658 0.99999675 0.99999676 0.99999692 0.999997
 0.99999711 0.99999734 0.99999742 0.99999772 0.99999786 0.999998
 0.99999813 0.99999823 0.99999833 0.99999849 0.99999849 0.99999869
 0.99999874 0.99999886 0.99999887 0.99999911 0.99999913 0.99999922
 0.99999927 0.99999932 0.99999932 0.99999938 0.99999939 0.99999943
 0.99999951 0.99999952 0.99999954 0.99999956 0.99999959 0.9999996
 0.99999964 0.99999965 0.99999967 0.99999968 0.99999969 0.99999973
 0.99999973 0.99999976 0.99999977 0.99999979 0.99999981 0.99999982
 0.99999982 0.99999983 0.99999986 0.99999987 0.99999987 0.99999988
 0.99999988 0.9999999  0.99999991 0.99999991 0.99999992 0.99999994
 0.99999995 0.99999995 0.99999995 0.99999995 0.99999996 0.99999996
 0.99999997 0.99999997 0.99999998 0.99999998 0.99999998 0.99999998
 0.99999998 0.99999998 0.99999998 0.99999999 0.99999999 0.99999999
 0.99999999 0.99999999 0.99999999 0.99999999 0.99999999 0.99999999
 0.99999999 0.99999999 0.99999999 0.99999999 1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.
 1.         1.         1.         1.         1.         1.        ]
```

In [23]:

```
# probability of general-recipe logistic regression in wrong instances
highest_prob_mismatches = np.sort(prob_classmax[y_val!=y_pred_logi])
print(f"Probabilities of best scoring mismatches:\n{highest_prob_mismatches}")
```

```
Probabilities of best scoring mismatches:
[0.57017208 0.58908621 0.5966738  0.62079818 0.63602062 0.69950432
 0.72635479 0.74412721 0.77734738 0.78849857 0.81360898 0.86038406
 0.88517528 0.89706347 0.93157945 0.96972717 0.96976152 0.99619916
 0.99638036 0.9974274  0.99901122 0.99945519 0.99999838 0.99999904]
```

In [24]:

```
mismatch_indices_gt_99 = np.intersect1d(np.where(y_val!=y_pred_logi), np.where(p
rob_classmax > 0.99))
print(f"Mismatch count above 99% probability : {len(mismatch_indices_gt_99)}")
```

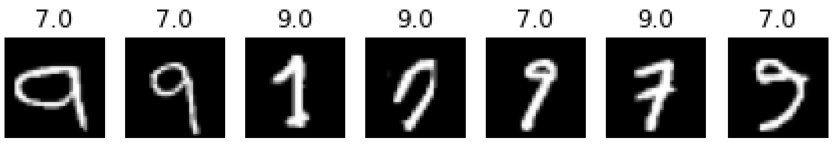
```
Mismatch count above 99% probability : 7
```

In [25]:

```
# Display mismatches above 99% probability
display_cnt = len(mismatch_indices_gt_99)

X_valv_mismatches = []
y_val_mismatches = []
y_pred_mismatches = []
compare = 'Comparison of actual vs predicted: \n'
for idx in mismatch_indices_gt_99:
    X_valv_mismatches.append(X_valv[idx])
    y_val_mismatches.append(y_val[idx])
    y_pred_mismatches.append(y_pred_logi[idx])
    compare += (f"y_pred:{y_pred_logi[idx]} y_val:{y_val[idx]}" +\
           f", Pr({classes[0]}):{y_pred_logi_prob[idx][0]:.8f}" +\
           f", Pr({classes[1]}):{y_pred_logi_prob[idx][1]:.8f}" +\
           f", Pr({classes[2]}):{y_pred_logi_prob[idx][2]:.8f}\n")

X_valv_mismatches = np.array(X_valv_mismatches)
y_val_mismatches = np.array(y_val_mismatches)
y_pred_mismatches = np.array(y_pred_mismatches)

print(compare)
plot_example(X=X_valv_mismatches, y=y_pred_mismatches,
             n=display_cnt, plot_title="Mismatches >99% probability (predictions
labelled)")
```

```
Comparison of actual vs predicted:
y_pred:7.0 y_val:9.0, Pr(3):0.00000160, Pr(7):0.99999838, Pr(9):0.00
000002
y_pred:7.0 y_val:9.0, Pr(3):0.00007013, Pr(7):0.99638036, Pr(9):0.00
354951
y_pred:9.0 y_val:7.0, Pr(3):0.00051846, Pr(7):0.00205414, Pr(9):0.99
742740
y_pred:9.0 y_val:7.0, Pr(3):0.00212730, Pr(7):0.00167353, Pr(9):0.99
619916
y_pred:7.0 y_val:9.0, Pr(3):0.00000096, Pr(7):0.99945519, Pr(9):0.00
054385
y_pred:9.0 y_val:7.0, Pr(3):0.00000448, Pr(7):0.00098430, Pr(9):0.99
901122
y_pred:7.0 y_val:9.0, Pr(3):0.00000096, Pr(7):0.99999904, Pr(9):0.00
000000
```

Mismatches >99% probability (predictions labelled)



We observe seven of the wrong predictions by logistic regression having a confidence above 99%. On reviewing the images it is difficult to understand how these have been labelled incorrectly with such high confidence. the probability values for the correct labels are very low given our total probabiliy must sum to 1.

Probabilistic ML

In [26]:

```
X_trv.shape[1]
```

Out[26]:

```
784
```

In [27]:

```
import sklearn.preprocessing
## We use LabelBinarizer to transfor classes into counts
# neg_label=0, pos_label=1
y_2_bin = sklearn.preprocessing.LabelBinarizer().fit_transform(y_tr.reshape(-1,1
))
nf = X_trv.shape[1]
# number of classes
nc = len(classes)
# floatX = float32
floatX = tt.config.floatX

init_b = np.random.randn(nf, nc-1).astype(floatX)
init_a = np.random.randn(nc-1).astype(floatX)


with pm.Model() as multi_logistic:
    # Prior
    β = pm.Normal('beta', 0, sigma=100, shape=(nf, nc-1), testval=init_b)
    α = pm.Normal('alpha', 0, sigma=100, shape=(nc-1,), testval=init_a)

    # we need to consider nc-1 features because the model is not identifiable
    # the softmax turns a vector into a probability that sums up to one
    # therefore we add zeros to go back to dimension nc
    # so that softmax returns a vector of dimension nc
    β1  = tt.tensor.concatenate([np.zeros((nf,1)),β ],axis=1)
    α1  = tt.tensor.concatenate([[0],α ],)

    # Likelihood
    mu = pm.math.matrix_dot(X_trv,β1) + α1
    # It doesn't work if the problem is binary
    p = tt.tensor.nnet.nnet.softmax(mu)
    observed = pm.Multinomial('likelihood', p=p, n=1, observed=y_2_bin)
```

We set our priors as normal distributions with mean of 0 and σ of 100.
For α we specify a vector size of the class count minus one, i.e. $3 - 1 = 2$.
For β we specify a matrix size of the input pixel count times the class count minux one, i.e. $784x2$.

In [28]:

```
y_2_bin
```

Out[28]:

```
array([[0, 0, 1],
       [0, 0, 1],
       [1, 0, 0],
       ...,
       [0, 1, 0],
       [1, 0, 0],
       [1, 0, 0]])
```

In [29]:

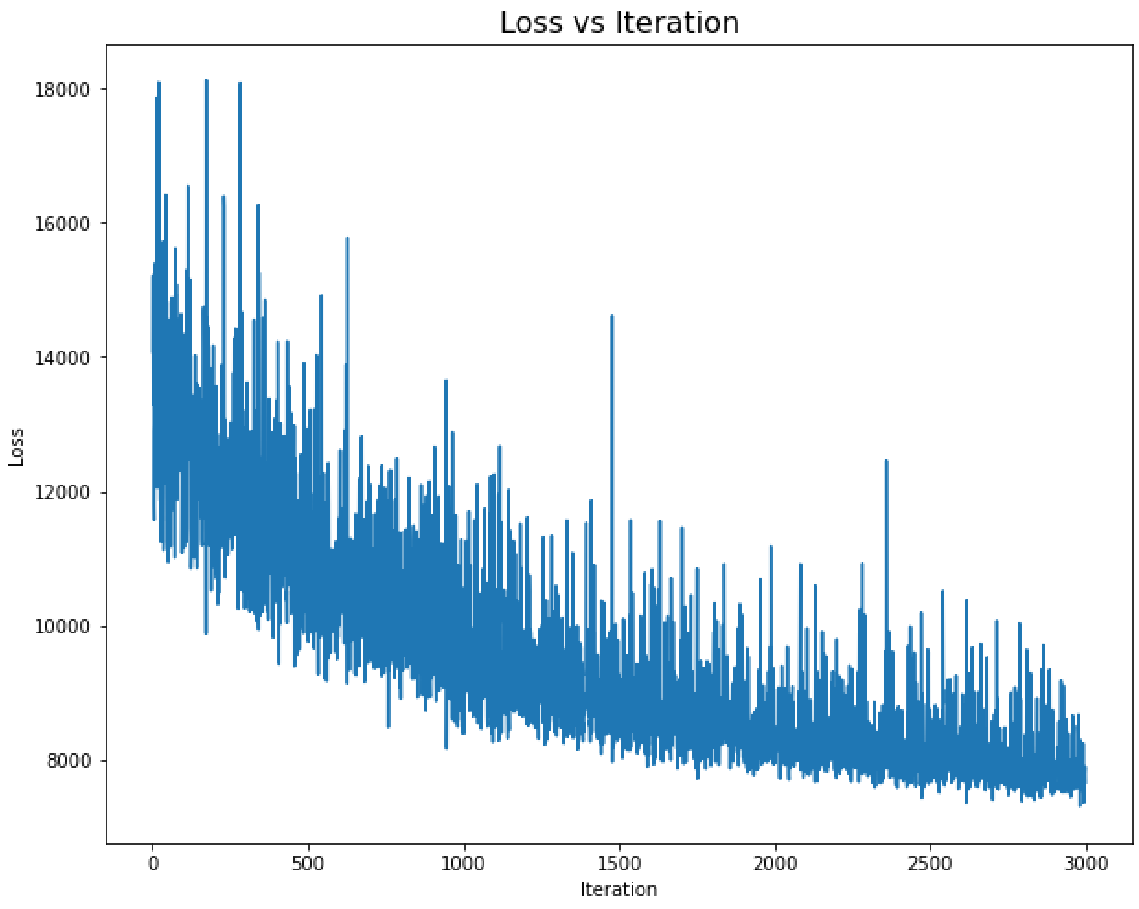
```
with multi_logistic:
    #approx = pm.fit(300000, method='advi') # takes longer
    approx = pm.fit(3000, method='advi')
```

100.00% [3000/3000 00:03<00:00 Average Loss = 8,169.5]

```
Finished [100%]: Average Loss = 8,161.6
```

In [30]:

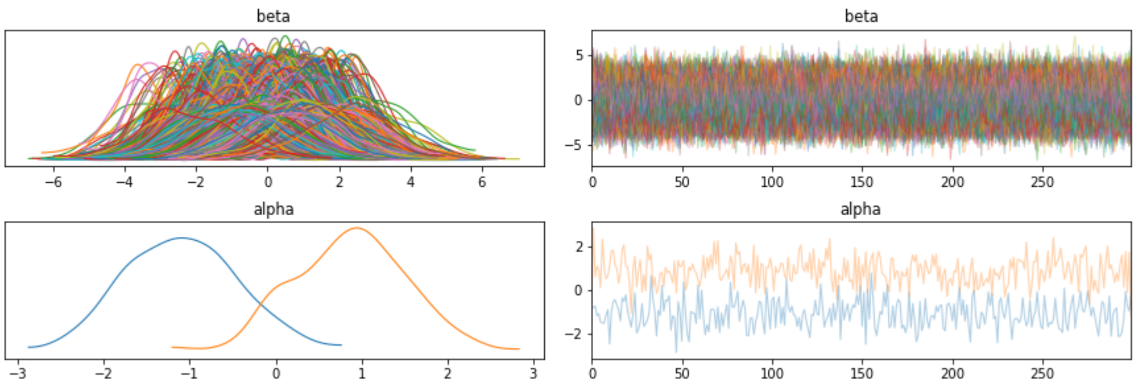
```
plt.figure(figsize=(10,8))
plt.ylabel('Loss')
plt.xlabel('Iteration')
plt.plot(approx.hist)
plt.title('Loss vs Iteration', fontsize=16)
plt.show()
```



The loss is seen to decrease as we iterate further on the model.

In [31]:

```
# View graph of the posterior alpha & beta values
dd = 300
posterior = approx.sample(draws=dd)
az.plot_trace(posterior);
```



In [32]:

```
# View summary table of the posterior
with multi_logistic:
    display(az.summary(posterior, round_to=2))
```

```
arviz - WARNING - Shape validation failed: input_shape: (1, 300), mi
nimum_shape: (chains=2, draws=4)
```

	mean	sd	hdi_3%	hdi_97%	mcse_mean	mcse_sd	ess_bulk	ess_tail	r_hat
beta[0,0]	2.05	1.21	-0.53	4.13	0.07	0.05	292.23	254.82	NaN
beta[0,1]	-0.04	1.32	-2.63	2.21	0.08	0.06	283.08	246.66	NaN
beta[1,0]	1.02	1.17	-1.06	3.47	0.06	0.04	340.54	278.92	NaN
beta[1,1]	-0.13	1.30	-2.62	2.32	0.08	0.06	258.19	270.97	NaN
beta[2,0]	-1.12	1.29	-3.23	1.57	0.07	0.05	313.39	182.98	NaN
...	...	...	...	...	...	...	...	...	...
beta[782,1]	2.30	1.25	0.10	4.79	0.09	0.06	205.83	233.21	NaN
beta[783,0]	0.88	1.22	-1.31	3.24	0.07	0.05	287.22	245.57	NaN
beta[783,1]	0.68	1.28	-1.57	3.15	0.08	0.06	255.56	182.18	NaN
alpha[0]	-1.09	0.65	-2.15	0.22	0.04	0.03	288.59	292.65	NaN
alpha[1]	0.86	0.66	-0.27	2.06	0.04	0.03	261.74	322.24	NaN

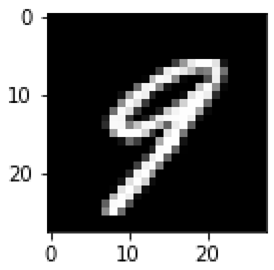
1570 rows × 9 columns

The summary table and plots show our two alpha values for our multinomial three class problem. The 784 beta values correpond to the input feature set size of $28x28 = 784$ pixels per image. The right hand size of the plot shows the samples of the Markov chain plotted for beta and alpha values.

In [33]:

```
## The softmax function transforms each element of a collection by computing the exponential
#  of each element divided by the sum of the exponentials of all the elements.
from scipy.special import softmax

#select an image in the test set
i = 10
#i = random.randint(0, dd)

#select a sample in the posterior
s = 100
#s = random.randint(0, dd)


beta  = np.hstack([np.zeros((nf,1)),  posterior['beta'][s,:] ])
alpha = np.hstack([[0],  posterior['alpha'][s,:] ])
image = X_valv[i,:].reshape(28,28)
plt.figure(figsize=(2,2))
plt.imshow(image,cmap="Greys_r")
np.set_printoptions(suppress=True)

print("test image #" + str(i))
print("posterior sample #" + str(s))
print("true class=", y_val[i])
print("classes: " + str(classes))
print("estimated prob=",softmax((np.array([X_valv[i,:].dot(beta) + alpha])))[0
,:])
```

```
test image #10
posterior sample #100
true class= 9.0
classes: [3, 7, 9]
estimated prob= [0.0000122  0.00063673 0.99935107]
```



In [34]:

```
# Bayesian prediction
# return the class that has the highest posterior probability
y_pred_Bayesian=[]

for i in range(X_valv.shape[0]):
    val=np.zeros((1,len(classes)))

    for s in range(posterior['beta'].shape[0]):
        beta = np.hstack([np.zeros((nf,1)),  posterior['beta'][s,:] ])
        alpha = np.hstack([[0],  posterior['alpha'][s,:] ])
        val = val + softmax((np.array([X_valv[i,:].dot(beta) + alpha])))

    mean_probability = val/posterior['beta'].shape[0]
    y_pred_Bayesian.append( np.argmax(mean_probability))
```

In [35]:

```
print(y_pred_Bayesian)
```

```
[0, 0, 2, 0, 2, 2, 1, 1, 0, 2, 2, 2, 2, 0, 1, 1, 2, 2, 0, 2, 0, 1,
0, 0, 2, 2, 1, 1, 1, 2, 2, 0, 2, 1, 2, 2, 0, 1, 0, 0, 2, 2, 2, 2, 0,
1, 2, 1, 1, 2, 1, 2, 1, 2, 1, 2, 1, 1, 2, 2, 0, 0, 2, 0, 1, 1, 2, 2,
0, 2, 0, 0, 1, 2, 0, 1, 1, 0, 0, 2, 2, 0, 0, 0, 1, 0, 2, 0, 1, 0, 1,
1, 0, 0, 2, 2, 2, 0, 0, 1, 2, 2, 0, 1, 2, 1, 0, 0, 1, 2, 0, 0, 0, 0,
2, 1, 0, 2, 1, 1, 0, 1, 0, 2, 1, 1, 0, 0, 2, 1, 1, 1, 0, 2, 1, 0, 0,
0, 2, 1, 0, 1, 1, 1, 0, 0, 2, 0, 2, 0, 0, 2, 0, 1, 1, 1, 2, 0, 1, 1,
2, 1, 0, 2, 2, 1, 1, 2, 0, 0, 0, 2, 2, 0, 1, 0, 0, 2, 2, 1, 2, 0, 1,
2, 1, 0, 1, 0, 0, 0, 0, 2, 2, 0, 1, 0, 2, 0, 0, 0, 0, 0, 1, 0, 1, 0,
0, 0, 1, 1, 2, 1, 0, 2, 2, 0, 2, 1, 1, 2, 1, 0, 1, 1, 0, 0, 2, 2, 0,
0, 1, 2, 2, 1, 1, 1, 2, 2, 0, 2, 1, 0, 1, 2, 0, 2, 2, 0, 2, 0, 2, 1,
0, 1, 1, 2, 0, 2, 1, 2, 0, 2, 1, 0, 1, 1, 2, 1, 1, 2, 2, 2, 2, 2, 2,
2, 2, 0, 0, 2, 2, 2, 1, 1, 2, 1, 2, 1, 2, 0, 2, 1, 0, 1, 0, 1, 1, 1,
2, 0]
```

In [36]:

```
# recall the classes we are using
print(classes)
```

```
[3, 7, 9]
```

In [37]:

```
# prediction array (using classes)
nn = 10 # just an example
np.array(classes)[y_pred_Bayesian[0:nn]]
```

Out[37]:

```
array([3, 3, 9, 3, 9, 9, 7, 7, 3, 9])
```

In [38]:

```
# using validation: y_val
print("Accuracy=", accuracy_score(np.array(classes)[y_pred_Bayesian], y_val))
```

```
Accuracy= 0.9166666666666666
```

Selecting Differences

In [39]:

```
y_predB=[]

for i in range(X_valv.shape[0]):
    #print(i)
    val=[]

    for s in range(posterior['beta'].shape[0]):
        beta = np.hstack([np.zeros((nf,1)),  posterior['beta'][s,:] ])
        alpha = np.hstack([[0],  posterior['alpha'][s,:] ])
        val.append(softmax((np.array([X_valv[i,:].dot(beta) + alpha])))[0,:])

    #mean probability
    valmean = np.mean(val,axis=0)
    #class with maximum mean probability
    classmax = np.argmax(valmean)
    #ranks
    ranks = np.array(val.copy())
    ranks   = ranks  *0 #init
    colmax = np.argmax(np.array(val),axis=1)
    ranks[np.arange(0,len(colmax)),colmax]=1

    y_predB.append( [classmax, valmean[classmax], np.std(ranks,axis=0)[classmax
]])


y_predB= np.array(y_predB)
```

In [40]:

```
# prediction array
mm = 10
y_predB[0:mm,:]
```

Out[40]:

```
array([[0.        , 0.9347046 , 0.22469733],
       [0.        , 0.98743297, 0.08137704],
       [2.        , 0.96950078, 0.16110728],
       [0.        , 0.9990545 , 0.        ],
       [2.        , 0.99309007, 0.08137704],
       [2.        , 0.70696767, 0.44395946],
       [1.        , 0.89223965, 0.29550334],
       [1.        , 0.55868374, 0.49553562],
       [0.        , 0.99452616, 0.05763872],
       [2.        , 0.99727114, 0.        ]])
```

In [41]:

```
#sorting in descending order
difficult = np.argsort(-y_predB[:,2])
y_predB[difficult[0:mm],:]
```

Out[41]:

```
array([[2.        , 0.50281749, 0.49995555],
       [0.        , 0.48651759, 0.49989999],
       [1.        , 0.50941751, 0.49989999],
       [2.        , 0.47250075, 0.49982219],
       [1.        , 0.48348163, 0.49982219],
       [1.        , 0.47129259, 0.49982219],
       [1.        , 0.49934503, 0.49982219],
       [2.        , 0.51614901, 0.49972215],
       [1.        , 0.52203914, 0.49959984],
       [1.        , 0.50332557, 0.49959984]])
```

In [42]:

```
#probability of general-recipe logistic regression in wrong instances
prob_classmax[y_pred_logi != y_val]
```

Out[42]:

```
array([0.81360898, 0.96976152, 0.99999838, 0.99638036, 0.58908621,
       0.72635479, 0.88517528, 0.9974274 , 0.62079818, 0.99619916,
       0.96972717, 0.69950432, 0.5966738 , 0.57017208, 0.99945519,
       0.99901122, 0.78849857, 0.93157945, 0.89706347, 0.77734738,
       0.99999904, 0.63602062, 0.74412721, 0.86038406])
```

In [43]:

```
y_predB[y_pred_logi != y_val,:]
```

Out[43]:

```
array([[2.        , 0.73872016, 0.43679387],
       [2.        , 0.78849031, 0.41197357],
       [1.        , 0.50566451, 0.49909919],
       [2.        , 0.99990406, 0.        ],
       [2.        , 0.98518848, 0.11469767],
       [2.        , 0.78052135, 0.4       ],
       [0.        , 0.93321646, 0.21794495],
       [2.        , 0.86720174, 0.32881944],
       [1.        , 0.50941751, 0.49989999],
       [2.        , 0.83029309, 0.36030851],
       [2.        , 0.72986088, 0.43863424],
       [2.        , 0.72042592, 0.44221664],
       [0.        , 0.85297756, 0.35041246],
       [2.        , 0.50281749, 0.49995555],
       [1.        , 0.57071711, 0.49355851],
       [2.        , 0.79029544, 0.40247843],
       [2.        , 0.62868471, 0.48095969],
       [2.        , 0.53642543, 0.49638695],
       [0.        , 0.75469335, 0.42503595],
       [2.        , 0.75869955, 0.41647996],
       [1.        , 0.97986564, 0.15095989],
       [2.        , 0.83799916, 0.36660606],
       [0.        , 0.8143858 , 0.36660606],
       [0.        , 0.82988176, 0.3756328 ]])
```

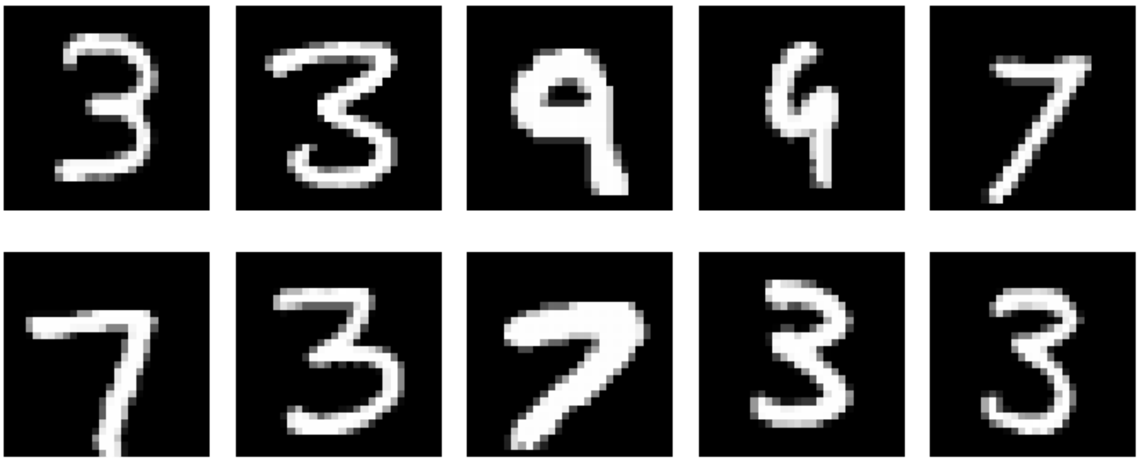
In [44]:

```
## Difficult & easy instances

easy = np.argsort(y_predB[:,2])
print("Accuracy in easy instances =", accuracy_score(y_pred_logi[easy[0:100]], y
_val[easy[0:100]]))

difficult = np.argsort(-y_predB[:,2])
print("Accuracy in difficult instances =", accuracy_score(y_pred_logi[difficult[
0:100]], y_val[difficult[0:100]]))
```

```
Accuracy in easy instances = 0.97
Accuracy in difficult instances = 0.83
```

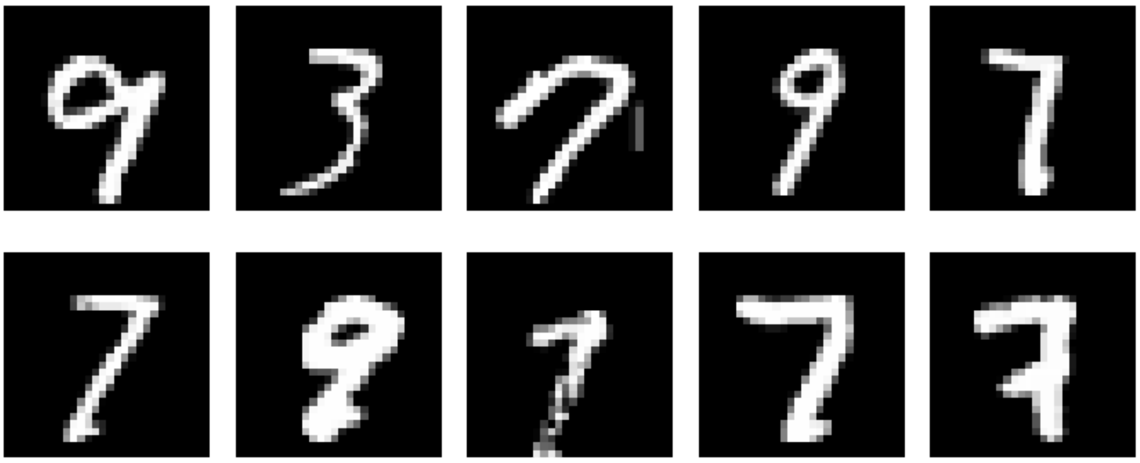
In [45]:

```
# show 10 random 'easy' images
fig, axs = plt.subplots(2,5, figsize=(15, 6))
fig.subplots_adjust(hspace = .2, wspace=.001)
axs = axs.ravel()

for i in range(10):
    index = easy[i]
    image = X_valv[index,:].reshape(28,28)
    axs[i].axis('off')
    axs[i].imshow(image,cmap="Greys_r")
```



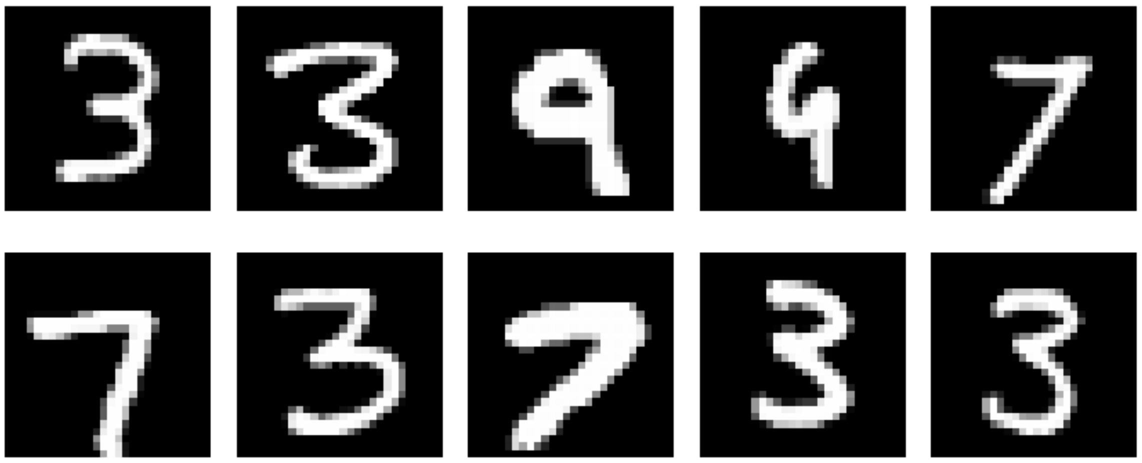
In [46]:

```
# show 10 random 'difficult' images
fig, axs = plt.subplots(2,5, figsize=(15, 6))
fig.subplots_adjust(hspace = .2, wspace=.001)
axs = axs.ravel()
for i in range(10):
    index = difficult[i]
    image = X_valv[index,:].reshape(28,28)
    axs[i].axis('off')
    axs[i].imshow(image,cmap="Greys_r")
```



In [47]:

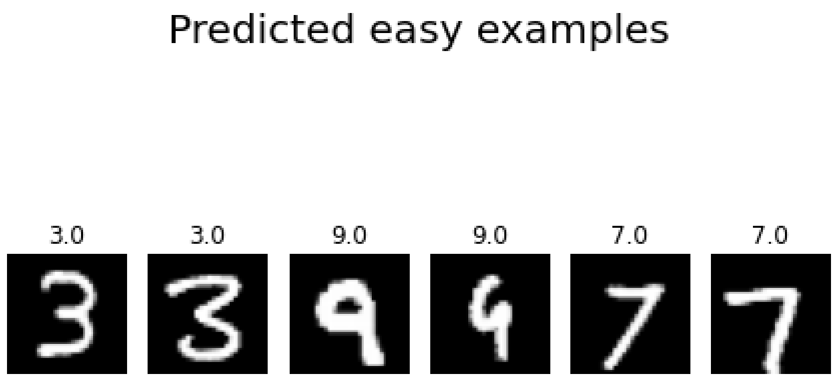
```
# show 10 random 'easy' images
fig, axs = plt.subplots(2,5, figsize=(15, 6))
fig.subplots_adjust(hspace = .2, wspace=.001)
axs = axs.ravel()

for i in range(10):
    index = easy[i]
    image = X_valv[index,:].reshape(28,28)
    axs[i].axis('off')
    axs[i].imshow(image,cmap="Greys_r")
```



Predicted answers - easy

In [48]:

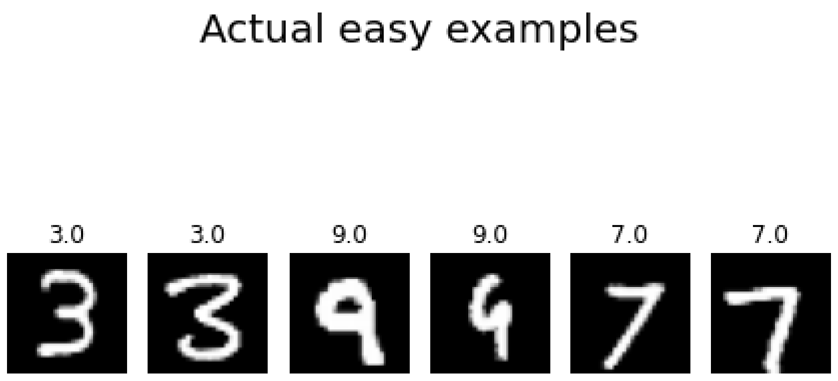
```
plot_example(X=X_valv[easy], y=y_pred_logi[easy], n=6, plot_title="Predicted eas
y examples")
```



Actual answers - easy

In [49]:

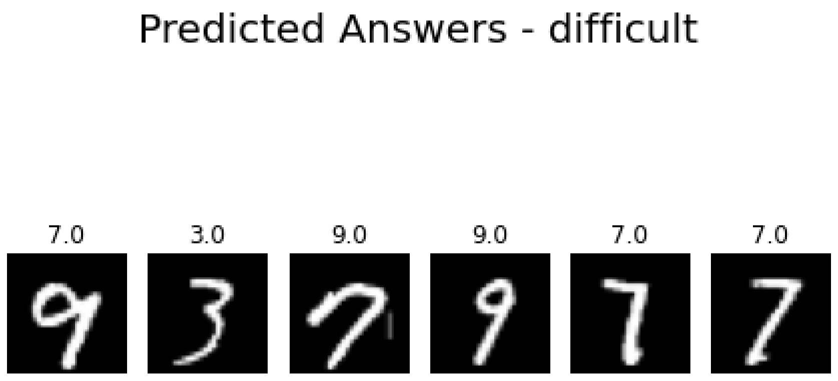
```
plot_example(X=X_valv[easy], y=y_val[easy], n=6, plot_title="Actual easy example
s")
```



Predicted answers - difficult

In [50]:

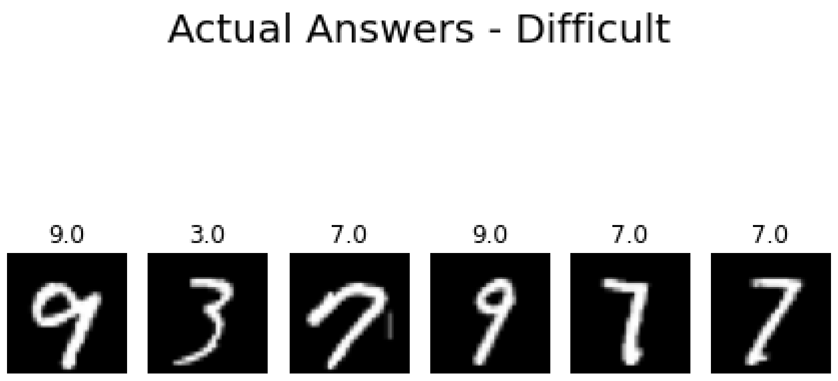
```
plot_example(X=X_valv[difficult], y=y_pred_logi[difficult], n=6, plot_title="Pre
dicted Answers - difficult" )
```



Actual answers - difficult

In [51]:

```
plot_example(X=X_valv[difficult], y=y_val[difficult], n=6, plot_title="Actual An
swers - Difficult")
```



Summary

We initially developed a Logistic Regression model to classify the MNIST dataset using `scikit learn` . With a reduced dataset of 1500 samples equally distributed over the 3 classes, split to produce 400 training samples per class, the trained model achieved 92% accuracy. In our opinion some of the wrongly classified images were not difficult to classifiy correctly. Our belief is that if we chose a larger sample size than 500 per class the performance of the logistic regression classifier may improve.

We then developed a Probabilistic Multinomial Bayesian Logistic Regression model using `pymc3` to classify the MNIST Database using the lbfgs optimiser. The lbfgs optimiser is a limited memory algorithm used for solving optimisation problems in nonlinear problems. This was developed by Ciyou Zhu, Richard H. Byrd, Peihuang Lu and Jorge Nocedal in 1994. We reuse the same reduced 3 class dataset as used with the Logistic Regression model. The Multinomial Bayesian Logistic Regression model achieves 91.6% accuracy when considering the classes with highest probability. This is comparable to the 92% of the Logistic Regression model which as noted could itself be improved with a larger training set.

An advantage to using Bayesian models is in the ability to add a prior. This allows us to add our knowledge of a problem domain to the model. We cannot do this with the Logistic Regression model in which we must rely on the dataset to include this knowledge. From our work with this etivity we set our priors as normal distributions with mean of 0 and σ of 100.
For α we specify a vector size of the class count minus one, i.e. $3 - 1 = 2$.
For β we specify a matrix size of the input pixel count times the class count minux one, i.e. $784x2$.

However we are free to choose appropriate distributions as priors which match the problem domain knowledge we wish to transfer to the model. This provides an advantage over non-Bayesian methods. Domains such as automated driving allow the incorporation of hardware sensing errors to the model in this manner. To set an appropriate prior requires a level of knowledge of the problem domain as a poorly chosen prior could adversely influence the posterior. This could be viewed as a disadvantage if appropriate domain knowledge is not available.

Conclusion

We believe Bayesian models offer a very useful tool in machine learning through the usage of priors. For our immediate work within this task we have been able to achieve an accuracy level of 92% which is relatively high. However using large convolutional neural networks other researchers have achieved accuracy rates above 99%.

References

Bibliography

Böhning, D., 1992. Multinomial logistic regression algorithm. Annals of the Institute of Statistical Mathematics, Volume 44, pp. 197-200.

Carpita, M., Sandri, M., Simonetto, A. & Zuccolotto, P., 2014. Chapter 14 - Football Mining with R. In: Y. Zhao & Y. Cen, eds. Data Mining Applications with R. s.l.:Academic Press, pp. 397-433.

Czepiel, S. A., n.d. Maximum Likelihood Estimation of Logistic Regression Models: Theory and Implementation. [Online] Available at: https://czep.net/stat/mlelr.pdf (https://czep.net/stat/mlelr.pdf) [Accessed 12 September 2021].

Koehrsen, W., 2018. Introduction to Bayesian Linear Regression. [Online] Available at: https://towardsdatascience.com/introduction-to-bayesian-linear-regression-e66e60791ea7 (https://towardsdatascience.com/introduction-to-bayesian-linear-regression-e66e60791ea7) [Accessed 12 September 2021].

Monroe, W., 2017. Stanford.edu. [Online] Available at: https://web.stanford.edu/class/archive/cs/cs109/cs109.1178/lectureHandouts/220-logistic-regression.pdf (https://web.stanford.edu/class/archive/cs/cs109/cs109.1178/lectureHandouts/220-logistic-regression.pdf) [Accessed 12 September 2021].

Scikit, 2021. sklearn.linear_model.LogisticRegression. [Online] Available at: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html) [Accessed 12 September 2021].

Sperandei, S., 2014. Understanding logistic regression analysis. Biochem med, 24(1), pp. 12-18.
Starkweather, J. & Moske, A. K., 2011. Multinomial Logistic Regression. [Online] Available at: https://it.unt.edu/sites/default/files/mlr_jds_aug2011.pdf (https://it.unt.edu/sites/default/files/mlr_jds_aug2011.pdf) [Accessed 14 September 2021].