Machine Learning Assignment 2

Team Members:

1	Arnav Jain	2020A3PS2125H
2	Tawish Singh	2020A3PS1900H
3	Bhaskar Mishra	2021B4A72427H

Part A - Naive Bayes Classifier to predict income

Naive Bayes Classifier is a probabilistic machine learning algorithm that is widely used for classification tasks. It works by calculating the probability of a data point belonging to each class based on the probability of its features. The algorithm assumes that the features are independent of each other, hence the name "naive." Although this assumption is often not true in real-world scenarios, Naive Bayes Classifier still performs remarkably well in many cases. It is simple, fast, and efficient, making it a popular choice for many data scientists and machine learning practitioners.

Naive Bayes Classifier has a wide range of applications in various fields, such as natural language processing, sentiment analysis, email filtering, spam detection, and medical diagnosis. It is particularly useful in scenarios where the number of features is large and the amount of data is limited. Naive Bayes Classifier can handle such situations efficiently because it requires only a small amount of training data, and it can learn the probability distribution of the features for each class quickly.

Task 1: Data Preprocessing

1. Use the dataset shared.

2. Load the dataset into a pandas DataFrame.

```
data = pd.read_csv('adult.csv')
```

3. Check for missing values and handle them appropriately.

```
print(data.isnull().sum())
print("total                          ", data.isnull().sum().sum())
```

4. Split the dataset into training and testing sets (66% for training, 33% for testing).

```
X_train, X_test, y_train, y_test = train_test_split(
    X1, Y1, test_size=0.33, random_state=123)
```

Task 2: Naive Bayes Classifier Implementation

1. Implement a function to calculate the prior probability of each class (benign and malignant) in the training set.

```
# calculate posterior probability for each class
for idx, c in enumerate(self._classes):
    prior = np.log(self._priors[idx])
    posterior = np.sum(np.log(self._pdf(idx, x)))
    posterior = posterior + prior
    posteriors.append(posterior)

# return class with the highest posterior
return self._classes[np.argmax(posteriors)]
```

2. Implement a function to calculate the conditional probability of each feature given to each class in the training set.

```
def _pdf(self, class_idx, x):
    mean = self._mean[class_idx]
    var = self._var[class_idx]
    numerator = np.exp(-((x - mean) ** 2) / (2 * var))
    denominator = np.sqrt(2 * np.pi * var)
    return numerator / denominator
```

3. Implement a function to predict the class of a given instance using the Naive Bayes algorithm.

```
def fit(self, X, y):
    n_samples, n_features = X.shape
    self._classes = np.unique(y)
    n_classes = len(self._classes)

    # calculate mean, var, and prior for each class
    self._mean = np.zeros((n_classes, n_features), dtype=np.float64)
    self._var = np.zeros((n_classes, n_features), dtype=np.float64)
    self._priors = np.zeros(n_classes, dtype=np.float64)

    for idx, c in enumerate(self._classes):
        X_c = X[y == c]
        self._mean[idx, :] = X_c.mean(axis=0)
        self._var[idx, :] = X_c.var(axis=0)
        self._priors[idx] = X_c.shape[0] / float(n_samples)
```

4. Implement a function to calculate the accuracy of your Naive Bayes classifier on the testing set.

```
def accuracy(y_true, y_pred):
    accuracy = np.sum(y_true == y_pred) / len(y_true)
    return accuracy
```

Task 3: Evaluation and Improvement

1. Evaluate the performance of your Naive Bayes classifier using accuracy, precision, recall, and F1-score.

```
precision = precision_score(y_test, predictions)
recall = recall_score(y_test, predictions)
f1score = f1_score(y_test, predictions)

print(f"Precision = {precision}")
print(f"Recall = {recall}")
print(f"F1 Score = {f1score}")
```

```
Precision = 0.676081200353045
Recall = 0.3974059662775616
F1 Score = 0.5005718019931383
```

2. Experiment with different smoothing techniques to improve the performance of your classifier. You need to study and understand the different smoothing techniques on your own.

We have used 2 smoothing techniques:

1. **Laplace smoothing:** Laplace smoothing, also known as add-one smoothing, is a technique used to smooth the probabilities in a Naive Bayes Classifier. It adds a small value (usually one) to the numerator and denominator of the probability calculation for each feature. This helps to avoid zero probabilities, which can cause problems in the calculations, especially when dealing with small datasets. Laplace smoothing improves the accuracy of the model by preventing overfitting and underfitting. However, it also introduces a bias in the model by assuming that each feature has equal importance, which may not be true in all cases.

```
Accuracy: 0.8366422633081028
Precision: 0.62873999157185
Recall: 0.7740596627756161
F1-score: 0.693872805487734
```

2. **Lidstone smoothing**: Lidstone smoothing is a generalization of Laplace smoothing that allows for non-uniform distribution of probabilities. It adds a small value (usually denoted by lambda) to the numerator and (lambda times the number of possible feature values) to the denominator of the probability calculation for each feature. This allows for a more fine-tuned adjustment of probabilities based on the available data. Lidstone smoothing can be used to prevent overfitting and underfitting, and it can also help to handle imbalanced datasets. However, choosing an appropriate value for lambda can be challenging and may require experimentation.

```
Accuracy: 0.7853331678868346
Precision: 0.8790786948176583
Recall: 0.1188674448767833
F1-score: 0.2093235831809871 8
```

3. Compare the performance of your Naive Bayes classifier with other classification algorithms like logistic regression and k-nearest neighbours.

Using Sckit Learn to code Logistic Regression and K-Nearest Neighbour:

Model	Smoothing Tech.	Accuracy
Naive Bayes	–	81.03%
Naive Bayes	Laplace smoothing	83.66%
Naive Bayes	Lidstone smoothing	78.55%
Logistic Regression	–	79.79%
K-Nearest Neighbour	–	84.86%

Part B: Building a Basic Neural Network for Image Classification

MNIST database(Modified National Institute of Standards and Technology database): a large database of handwritten digits that is commonly used for training various image processing systems. The database is also widely used for training and testing in the field of machine learning.

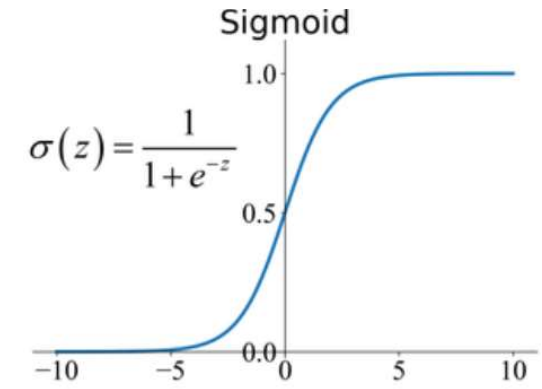
Here we have used Tensorflow to implement the Neural Network using the following code:

```
model = keras.Sequential([
    keras.layers.Dense(50, input_shape=(784,), activation='sigmoid'),
    keras.layers.Dense(50, input_shape=(784,), activation='sigmoid'),
    keras.layers.Dense(50, input_shape=(784,), activation='sigmoid'),
    keras.layers.Dense(10, activation='sigmoid')
])

model.compile(optimizer='adam',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

model.fit(X_train_flattened, y_train, epochs=10)
```

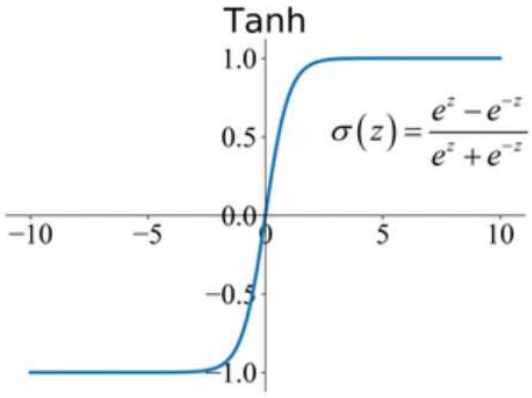
Here there are 3 Hidden Layers with 50 Nodes each and have the activation function “Sigmoid”.

The Optimizer used is “ADAM Optimizer” ADAM (Adaptive Moment Estimation) optimizer is a popular optimization algorithm used in deep learning to update the weights of neural networks during the training process. It is an adaptive learning rate method that combines the benefits of both AdaGrad and RMSProp optimizers.

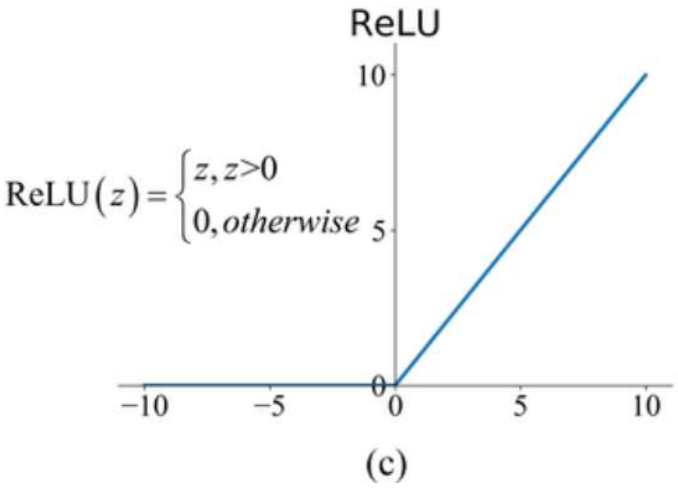
The Activation Functions used:



(a)



(b)



(c)

Model No.	Hidden Layer(No.)	Neuron in 1st layer	Neuron in 2nd layer	Neuron in 3rd layer	Activation func. in 1st layer	Activation func. in 2nd layer	Activation func. in 3rd layer
1	2	50	50	-	tanh	tanh	-
2	2	50	50	-	tanh	sigmoid	-
3	2	50	50	-	tanh	ReLu	-
4	2	50	50	-	sigmoid	sigmoid	-
5	2	50	50	-	sigmoid	ReLu	-
6	2	50	50	-	ReLu	ReLu	-
7	3	50	50	50	tanh	tanh	tanh
8	3	50	50	50	ReLu	tanh	tanh
9	3	50	50	50	ReLu	ReLu	tanh
10	3	50	50	50	sigmoid	tanh	tanh
11	3	50	50	50	sigmoid	sigmoid	tanh
12	3	50	50	50	sigmoid	sigmoid	sigmoid
13	3	50	50	50	ReLu	sigmoid	sigmoid
14	3	50	50	50	ReLu	ReLu	sigmoid
15	3	50	50	50	ReLu	ReLu	ReLu

Architecture 1 has two hidden layers with 50 neurons in each layer, both using the tanh activation function. It has moderate complexity and may perform well on a wide range of problems.

Architecture 2 is similar to Architecture 1 but uses the sigmoid activation function in the second layer instead of tanh. This may make it better suited to binary classification problems.

Architecture 3 is also similar to Architecture 1 but uses the ReLU activation function in the second layer instead of tanh. This may make it faster to train, but it may not perform as well on problems with negative input values.

Architecture 4 uses the sigmoid activation function in both hidden layers, which may make it better suited to binary classification problems. However, it may take longer to train due to the saturation of the sigmoid function.

Architecture 5 is similar to Architecture 4 but uses the ReLU activation function in the first layer. This may make it faster to train, but it may not perform as well on problems with negative input values.

Architecture 6 uses the ReLU activation function in both hidden layers, which may make it faster to train and perform well on problems with positive input values.

Architecture 7 has three hidden layers with 50 neurons in each layer, all using the tanh activation function. It is more complex than architectures 1-6 and may perform well on problems with more complex relationships.

Architecture 8 is similar to Architecture 7 but uses the ReLU activation function in the first layer instead of tanh. This may make it faster to train and perform well on problems with positive input values.

Architecture 9 is similar to Architecture 8 but uses the ReLU activation function in the second layer instead of tanh. This may make it faster to train and perform well on problems with negative input values.

Architecture 10 uses the sigmoid activation function in the first layer and the tanh activation function in the second and third layers. This may make it better suited to binary classification problems with non-linear decision boundaries.

Architecture 11 is similar to Architecture 10 but uses the sigmoid activation function in both hidden layers. This may make it better suited to binary classification problems with highly non-linear decision boundaries.

Architecture 12 uses the sigmoid activation function in all three hidden layers, which may make it better suited to binary classification problems with highly non-linear decision boundaries. However, it may take longer to train due to the saturation of the sigmoid function.

Architecture 13 uses the ReLU activation function in the first layer and the sigmoid activation function in the second and third layers. This may make it faster to train and perform well on problems with positive input values.

Architecture 14 is similar to Architecture 13 but uses the ReLU

Architecture 15 has three hidden layers with 50 neurons in each layer, all using the ReLU activation function. This may make it faster to train and perform well on problems with positive input values.

Model No.	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6	Iteration 7	Iteration 8	Iteration 9	Iteration 10	Final Accuracy
1	90.44	89.54	89.49	90.28	90.10	89.63	90.61	89.37	89.90	90.43	89.979
2	90.25	90.27	90.38	90.13	90.17	90.11	90.84	89.78	91.08	90.22	90.323
3	89.54	89.33	90.50	90.64	90.93	90.11	89.58	91.83	89.79	89.48	90.173
4	91.20	91.60	91.24	90.18	91.38	90.53	90.53	90.78	91.83	91.01	91.028
5	92.07	91.98	91.23	91.34	92.34	91.14	91.24	91.25	92.12	91.88	91.659
6	74.95	83.33	89.42	62.78	74.71	91.24	85.64	74.85	73.41	74.81	81.514
7	90.01	88.82	90.69	87.50	89.25	90.37	89.05	89.87	90.83	89.50	89.589
8	89.68	89.45	90.15	89.58	88.49	90.89	89.17	89.69	89.47	89.10	89.567
9	91.56	91.88	89.51	89.90	91.17	89.75	89.32	91.30	89.34	90.99	90.472
10	90.78	92.22	91.49	91.60	90.72	88.53	91.60	91.46	92.01	91.86	91.230
11	91.00	90.98	90.58	91.70	91.37	91.50	92.09	91.52	91.74	91.28	91.376
12	90.03	90.77	92.16	90.67	90.31	91.34	90.69	91.26	90.85	90.74	90.982
13	81.30	93.76	93.64	92.98	93.34	90.07	93.83	91.94	93.79	93.47	91.812
14	91.01	94.21	94.72	93.95	94.74	93.40	94.27	94.34	93.36	94.49	93.649
15	89.50	86.88	95.53	92.81	77.36	94.76	91.68	91.45	81.76	95.82	89.755

Based on the architectures listed in the table, we cannot definitively say that the faster architectures are less accurate than the slower ones. The table provides information on the number of hidden layers, the number of neurons in each layer, and the activation functions used in each layer for different architectures, but it does not provide information on the performance of these architectures on specific tasks or datasets.

The performance of a neural network architecture depends on the specific problem it is being applied to, the quality and quantity of the available data, and the choice of hyperparameters such as the learning rate and regularization. Therefore, it is not accurate to say that faster architectures are less accurate or slower ones are more…

It's difficult to determine the best classifier based solely on the provided accuracy scores for each model across 10 iterations. However, you can compare the average accuracy and standard deviation of each model across the iterations to get a sense of their relative performance.

To calculate the average accuracy and standard deviation, you can use the following formulae:

Here's the table with the calculated average accuracy and standard deviation for each model:

Based on the average accuracy, models 4, 5, 13,10,11 and 14 appear to be the best performers, with average accuracies above 91%. However, model 13 has a high standard deviation, indicating that its accuracy varies significantly across iterations. On the other hand, models 1, 3, 6, and 15 have relatively low average accuracies and high standard deviations, suggesting that they are less reliable.

Therefore, it is important to also consider other factors such as training time and complexity of the model before selecting the best classifier for a particular task.