Privacy Preserving Decision Tree Prediction

Reem Younis, Assia Khateeb, Atheer Abo Foul, Aya Miari
Lecturer : Dr. Adi Akavia, Laboratory in Privacy Preserving Machine Learning, University of Haifa
Email: reembyounis@gmail.com, assia.khteb@gmail.com, 19aether6@gmail.com, aia-m-211@hotmail.com

Part A:

Contents

***Abstract*—In machine learning, the decision tree is an algorithm for supervised learning for classification. The algorithm allows for learning, in that it processes elements in the training set one at a time.We implement decision tree algorithm on clear-text dataset. Then, we test accuracy of our decision tree algorithm classification results.**

1. Introduction

Reference to our Github repo.
Decision Tree algorithm belongs to the family of supervised learning algorithms. Unlike other supervised learning algorithms, decision tree algorithm can be used for solving regression and classification problems too.

The general motive of using Decision Tree is to practice it .
Decision Tree represents a procedure for classifying data based on attributes or features. It is also an efficient way of processing data ,for this very reason it has wide application in data mining.
In Decision Tree structure each internal node represents a "test" on an attribute, each branch represents the outcome of the test, and each leaf node represents a class label. The paths from root to leaf represent classification rules.
This data structure is quite intuitive and easy to assimilate by humans.

2. Methodology

The working model of decision tree is quite easy to implement and can be very effective in most of the classification problems.
In our decision tree, for predicting a class label for a dataset We compare the values of the root attribute with dataset's attribute. On the basis of comparison, we follow the branch corresponding to that value and jump to the next node.

We continue comparing our dataset's attribute values with other internal nodes of the tree until we reach a leaf node with predicted class value.

2.1. Prediction And Samples

Assumptions we make while using Decision Tree :
1. At the beginning, we consider the whole training set as the root.

2. Attributes are assumed to be categorical for information gain and for gini index, attributes are assumed to be continuous.
3. On the basis of attribute values records are distributed recursively.
4. We use statistical methods for ordering attributes as root or internal node.

2.2. Attribute Selection Measures

Attribute selection measure is a heuristic for selecting the splitting criterion that partition data into the best possible manner. It is also known as splitting rules because it helps us to determine breakpoints for tuples on a given node. ASM provides a rank to each feature(or attribute) by explaining the given data set. Best score attribute will be selected as a splitting attribute (Source). In the case of a continuous-valued attribute, split points for branches also need to define. Most popular selection measures are Information Gain, Gain Ratio, and Gini Index.

2.3. Entropy

In physics and mathematics, entropy referred as the randomness or the impurity in the system. In information theory, it characterizes the impurity of an arbitrary collection of examples. Entropy is the measure of uncertainty of a random variable, The higher the entropy the more the information content.
The entropy can explicitly be written as:
H(X) = $\sum_{n=1}^{N} p(x_i) \log_2 p(x_i)$
By calculating entropy measure of each attribute we can calculate their information gain. Information Gain calculates the expected reduction in entropy due to sorting on the attribute. Information gain can be calculated.

2.4. Information Gain

Information gain is the decrease in entropy. Information gain computes the difference between entropy before split and average entropy after split of the data set based on given attribute values.
Definition: Suppose S is a set of instances, A is an attribute, $\S_v$ is the subset of S with A=v and Values(A) is the set of all possible of A, then
Gain(S,A) = Entropy(S) - $\sum_{v:val(A)} |S_v| Entropy(|S_v|)$
$|S|$ denotes the size of set S

2.5. Gini index

Another decision tree algorithm CART (Classification and Regression Tree) uses the Gini method to create split points.
Gini index and Information Gain both of these methods are used to select from the n attributes of the dataset which attribute would be placed at the root node or the internal node.
Gini index = 1 - $\sum_j P_j^2$

3. Algorithm

How the algorithm works?
1. Select the best attribute using Attribute Selection Measures(ASM) to split the records.
2. Make that attribute a decision node and breaks the dataset into smaller subsets.
3. Starts tree building by repeating this process recursively for each child until one of the condition will match:
–All the tuples belong to the same attribute value.
–There are no more remaining attributes.

3.1. Pruning Strategy

To prune each node one by one (except the root and the leaf nodes), and check weather pruning helps in increasing the accuracy, if the accuracy is increased, prune the node which gives the maximum accuracy at the end to construct the final tree (if the accuracy of 100% is achieved by pruning a node, stop the algorithm right there and do not check for further new nodes).

4. Decision Tree - Python

4.1. driver.py

This file gets input from online sources (for example IRIS data "https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data").
In addition, all the functions in DecisionTree.py are called in this file. For example : build_tree, getLeafNodes, getInnerNodes, computreAccuracy, print_tree.

4.2. DecisionTree.py

function build_tree : a recursice function that returns the final tree.
rows - contains number of objects.
header - contains number of columns/features/labels.
function find_best_split : returns best question that could be asked so far, in addition to best information gain.
rows,header refer to the same as in build_tree.
function Leaf : initializes the leaf data.
function partition : checks if a question matches an object, if yes then add to true_rows, if no add to false_rows.
returns two arrays, true_rows,false_rows.
function Decision_Node : This holds a reference to the question, and to the two child nodes.

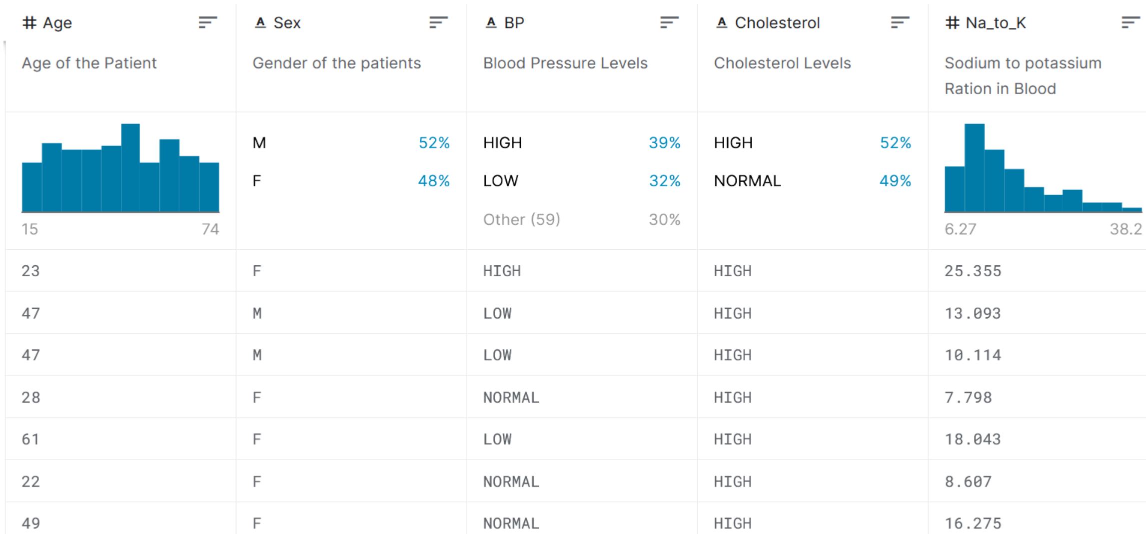
5. Datasets

Drug

The dataset contains various information that effect the predictions like Age, Sex, BP, Cholesterol levels, Na to Potassium Ratio and finally the drug type.

The target feature is:
–Drug type

The feature sets are:
1. Age
2. Sex
*3.*Blood Pressure Levels (BP)
4. Cholesterol Levels
5. Na to Potassium Ration



# Age	Sex	BP	Cholesterol	# Na_to_K
Age of the Patient	Gender of the patients	Blood Pressure Levels	Cholesterol Levels	Sodium to potassium Ration in Blood
15 – 74	M 52%, F 48%	HIGH 39%, LOW 32%, Other (59) 30%	HIGH 52%, NORMAL 49%	6.27 – 38.2
23	F	HIGH	HIGH	25.355
47	M	LOW	HIGH	13.093
47	M	LOW	HIGH	10.114
28	F	NORMAL	HIGH	7.798
61	F	LOW	HIGH	18.043
22	F	NORMAL	HIGH	8.607
49	F	NORMAL	HIGH	16.275

Figure 1. Drug Classification Dataset

6. Results

6.1. Sample outputs (Drug Classification Dataset)

–Accuracy before pruning: 97.0%
–Accuracy after pruning: 97.0%
Pruning strategy did not increased accuracy
–Final Tree with accuracy: 97.0%

Part B - Implementation of Algorithm 1

August 20, 2021

Contents

Abstract.
In part b we implement cleartext soft decision tree prediction algorithm; algorithm specified in Akavia et al ECML'2020.

1 Introduction

Reference to our Github repo.
Part b of the lab is done by dividing the problem into multiple phases, each one we called it pre-processing phase.

- First pre-procssing is scaling the data.
- Second pre-procssing is preparing the polynom which is required for Algorithm 1.
- Third pre-procssing is building a tree with 1-hot encoding labels.

We predict on set of samples using Algorithm 1. Then, we present the main results of Algorithm 1 and compare it to scikit learn results.

2 Preprocessing Phase

In this section, by using preprocessing, input data is processed to produce an output data that is used in our program.

2.1 SCALING

We used the sklearn.preprocessing package which provides several common utility functions and transformer classes to change raw feature vectors into a representation that is more suitable for the downstream estimators.

We start by re-scaling each value in the given data set, to a value in range $[-1, 1]$ (according to the article) using MinMaxScaler package from sklearn. The general formula to rescale a range between a of values [a, b] is given as:

$$x' = a + \frac{(x - min(x))(b - a)}{max(x) - min(x)} \tag{1}$$

In our case $[a, b]$ is $[-1, 1]$ as explained above.
Reference to Feature Scaling in Wikipedia: Feature Scaling

Figure 1: DATA BEFORE SCALING

```
#Load Data and store it into pandas DataFrame objects
iris = load_iris()
X = pd.DataFrame(iris.data[:, :], columns=iris.feature_names[:])
print(X)
```

```
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                  5.1               3.5                1.4               0.2
1                  4.9               3.0                1.4               0.2
2                  4.7               3.2                1.3               0.2
3                  4.6               3.1                1.5               0.2
4                  5.0               3.6                1.4               0.2
..                 ...               ...                ...               ...
145                6.7               3.0                5.2               2.3
146                6.3               2.5                5.0               1.9
147                6.5               3.0                5.2               2.0
148                6.2               3.4                5.4               2.3
149                5.9               3.0                5.1               1.8

[150 rows x 4 columns]
```

Figure 2: DATA AFTER SCALING

```
# preprocessing 1 - Feature scaling
# rescale a range between an arbitrary set of values [a, b] where a=-1, b=1
scaler = MinMaxScaler(feature_range=(-1, 1)) # build the scaler model
X_rescaled_features = scaler.fit_transform(X)
X_rescaled_features = pd.DataFrame(X_rescaled_features[:, :], columns=iris.feature_names[:])
print(X_rescaled_features)
```

```
     sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0            -0.555556          0.250000          -0.864407         -0.916667
1            -0.666667         -0.166667          -0.864407         -0.916667
2            -0.777778          0.000000          -0.898305         -0.916667
3            -0.833333         -0.083333          -0.830508         -0.916667
4            -0.611111          0.333333          -0.864407         -0.916667
..                 ...               ...                ...               ...
145           0.333333         -0.166667           0.423729          0.833333
146           0.111111         -0.583333           0.355932          0.500000
147           0.222222         -0.166667           0.423729          0.583333
148           0.055556          0.166667           0.491525          0.833333
149          -0.111111         -0.166667           0.389831          0.416667

[150 rows x 4 columns]
```

We next use train_test_split model from sklearn to Split data set into random train and test subsets.
Using the train parameter we build the decision tree using DecisionTreeClassifier package from sklearn, choosing 4 as the tree's max depth.

2.1.1 Code

```
def scaling(X):
    scaler = MinMaxScaler(feature_range=(-1, 1))
    X_rescaled_features = scaler.fit_transform(X)
    return X_rescaled_features
```

2.2 POLYNOM

We construct a low-degree polynomial approximation in order to approximate the step function, aiming to replace it with a soft step function.
This is done by using mean square integral solution which is the soft step function :

$$\phi = \min_{p \in p_n} \int_{-2}^{2} (I_0(x) - p(x))^2 \, dx \tag{2}$$

Then, by adding an importance to weight of the approximation interval we maintain the sensitivity to error in the approximation is uniform over the domain. This leads to the optimization problem given in the article :

$$\phi = \min_{p \in p_n} \int_{-2}^{2} (I_0(x) - p(x))^2 w(x) \, dx \tag{3}$$

2.2.1 Code

We present our code for the polynomial approximation:

```
def polynom(degree, window):
    X = np.linspace(-2, 2, num=201)
    y1 = np.zeros(100)
    y2 = np.ones(101)
    Y = np.concatenate((y1, y2), axis=None)
    # weights functions
    w1 = np.concatenate((window * (np.ones(75)), np.zeros(51)), axis=None)
```

```
    w2 = window * np.ones(75)
    weight = np.concatenate((w1, w2), axis=None)
    pf = np.polyfit(X, Y, degree, w=weight)
    phi = np.poly1d(pf)
    return phi
```

Using sklearn numpy model we used linspace function to create an evenly spaced samples which are calculated over an interval[start,stop], the returned value is stored in parameter X.
Followed by parameter Y, the step function which maps the values $[1, 100]$ to 0 and values $[101, 201]$ to 1.

X represents a numpy array with values in range $[-2, 2]$ with 201 samples.
If we choose window to be equal to 0.25 then, the array X will be divided to 3 ranges: $[-2 : -0.25]$, $[-0.25 : 0.25]$ and $[0.25 : 2]$. The size of ranges $[-2 : -0.25] + [0.25 : 2]$ is 7/2. The values in ranges $[1 : 75]$ and $[127 : 201]$ have a weight of 2/7 according to the equation:

$$\int_{-2}^{2} w(x)\,dx = 1 \tag{4}$$

while the values in range $[76 : 126]$ have weights of 0. Using polyfit model, X is fitted to Y with degree = 34 and the calculated weights above.

We perform polynomial fitting on data set using polyfit model which returns a vector of coefficients p that minimizes the squared error. Then we create a polynomial model using poly1d function, which it's return value indicates whether the polynomial's coefficients powers are in a decreasing order. The result is stored in parameter phi.

2.3 Build Tree With 1-Hot Encoding Labels

We start building our tree by calling the builtTree function in file my_tree.py.
Each tree's inner node has a field for threshold, feature and node_id.
Parameters in builtTree function :
lenHot = length of label's value.
index_of_max = index of argmax value.
Using these two parameters, leaf array is built, so it's size is equal to the size of lenHot array. It's initialized with zeros, and ones in index_of_max.
Thus, we get an array of values as a form of 1 hot encoding.
Example:

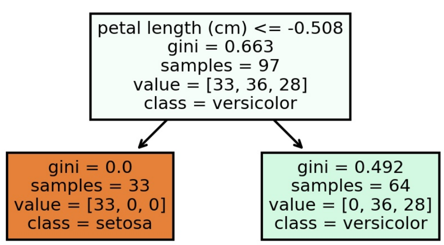


Figure 3:
Subtree labels before 1-hot encoding

Labels of the subtree after 1-hot encoding.
leaf = [1. 0. 0.]
leaf = [0. 1. 0.]

builtTree function This function receives as input, the tree and node_id which is initialized to 0. Tree's nodes are numbered by post-order traversal.
Recursively, leaves values are updated to 1 hot encoding format.

printTree function:This function receives as input, the tree and tree's depth which is initialized to 0.
For every inner node, threshold and feature are printed. However, for every leaf, only 1 hot value is printed.

printTree function recognizes whether the node is an inner node or leaf using $isinstance(leaf, np.ndarray)$ which returns if there's an array in the current node, if True then it's a leaf.

3 Algorithm 1

3.0.1 Code

```
def Tree_Predict(T, x, phi):
    if T is None:
        return
    feature, threshold, leaf, left, right = T.getNode()
    if isinstance(leaf, np.ndarray):
        return leaf
    else:
        return (phi(x[feature] - threshold)) * Tree_Predict(right, x, phi) +
        + (phi(threshold - x[feature])) * Tree_Predict(left, x, phi)
```

Akavia's algorithm traverses all paths in the tree and computes a weighted combination of all the leaves values, where each leaf value is the 1-hot encoding of the label associated with the leaf. The output is a length L vector assigning a likelihood score to each label, which is in turn interpreted as outputting the label with the highest score.
Here we used polyval

4 Accuracy

The accuracy is calculated as the percentage of correct classification on test samples.

```
algorithm1_score = (counter / len(res_vec))*100
```

Counter is equal to the sum of correct classifications and res_vec is equal to the sum of all classifications.
To know if current classification is correct or not we compare it to Y target value of the relevant sample if they are eqaul then the prediction of algorithm 1 is correct .

test samples are decided to sent to algorithm 1 as 35% of the samples in the data-set and they are chosen randomly from the data-set.
test samples are assigned to variable X_test.
So algorithm 1 predict over all test samples, after that we calculate accuracy according to the percentage of correct predictions on test samples that algorithm 1 predict.

```
for x in X_test:
    res = algorithm1_predict(myTree, x, phi)
    res_vec.append(res)
```

We trained trees up to depth 6 and compare algorithm 1 prediction's accuracy vs. scikit learn prediction's accuracy for three data sets: iris, wine and cancer.

Table 1: iris

Tree depth	Algorithm 1 acc	scikit learn acc
0	22.64151	22.64151
1	62.26415	62.26415
2	94.33962	94.33962
3	86.7925	96.22642
4	100	96.22642
5	94.3396	96.22642
6	98.1132	94.33962

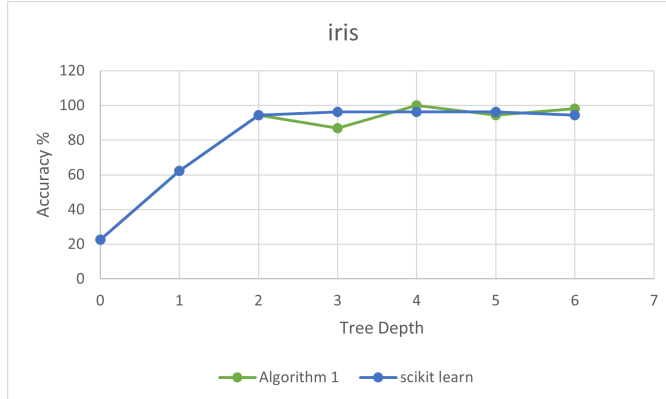


Figure 4:
Accuracy of ours vs. Scikit-learn on iris dataset and tree depth 0-6

Table 2: wine

Tree depth	Algorithm 1 acc	scikit learn acc
0	38.09524	38.09524
1	50.79365	50.79365
2	90.4762	87.30159
3	92.0635	90.47619
4	90.4762	88.88889
5	93.6508	92.06349
6	96.8254	92.06349

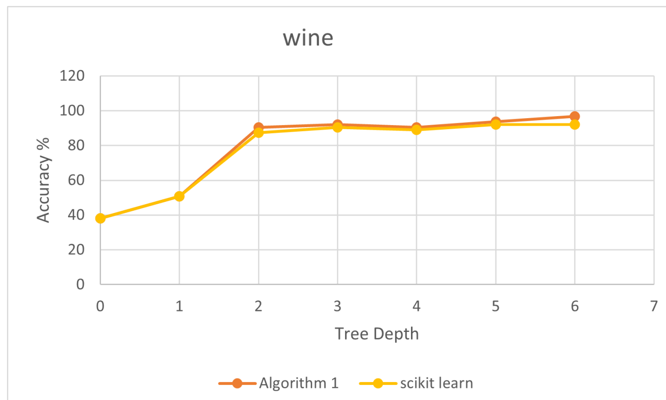


Figure 5:
Accuracy of ours vs. Scikit-learn on Wine dataset and tree depth 0-6

Table 3: cnacer

Tree depth	Algorithm 1 acc	scikit learn acc
0	61.5	61.5
1	87.5	87.5
2	94.5	91
3	96	96
4	94.5	91.5
5	92	92
6	96.5	92

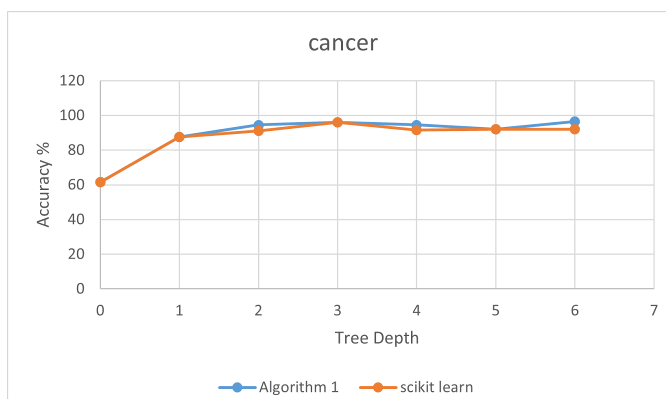


Figure 6:
Accuracy of ours vs. Scikit-learn on Cancer dataset and tree depth 0-6

5 Results

Example of results we got :

data_type = iris
max depth = 4
scikit_learn_score = 92.45283 %

WINDOW = 0

window = 0
polynom order = 0
run time = 0.0180 seconds
algorithm1_score = 32.0755 %

---

window = 0
polynom order = 1
run time =0.0150 seconds
algorithm1_score = 90.5660 %

---

window = 0
polynom order = 2
run time =0.0180 seconds
algorithm1_score = 90.5660 %

---

window = 0
polynom order = 3
run time = 0.0190 seconds
algorithm1_score = 92.4528 %

.
.
.
window = 0
polynom order = 33
run time = 0.0349 seconds
algorithm1_score = 92.4528 %

WINDOW = 0.25

window = 0.25
polynom order = 0
run time = 0.0439 seconds
algorithm1_score = 32.0755 %

.
.
.
window = 0.25
polynom order = 33
run time = 0.0499 seconds
algorithm1_score = 92.4528 %

WINDOW = 0.5

window = 0.5
polynom order = 0
run time = 0.0429 seconds
algorithm1_score = 32.0755 %

.
.
.
window = 0.5
polynom order = 33
run time = 0.0499 seconds
algorithm1_score = 92.4528 %

WINDOW = 0.75

window = 0.75
polynom order = 0
run time = 0.0259 seconds
algorithm1_score = 32.0755 %

.
.
.
window = 0.75
polynom order = 33
run time = 0.0289 seconds
algorithm1_score = 92.4528 %

For each window [0, 0.25, 0.5, 0.75] We calculate the algorithm's 1 accuracy for every polynom's degree in range [0,33]. Then we arranged all the results in tables (table 4 to 15).

5.1 Results For Cancer Data-Set

Cancer Tree:

```
feature = 22 threshold = -0.41192
   feature = 27 threshold = 0.28384
      feature = 27 threshold = -0.23814
         feature = 28 threshold = -0.99980
            leaf = [1. 0.]
            leaf = [0. 1.]
         feature = 1 threshold = -0.23266
            leaf = [0. 1.]
            leaf = [1. 0.]
      leaf = [1. 0.]
   feature = 6 threshold = -0.65742
      feature = 1 threshold = -0.38282
         leaf = [0. 1.]
         feature = 15 threshold = -0.72109
            leaf = [1. 0.]
            leaf = [0. 1.]
      feature = 7 threshold = -0.57465
         feature = 9 threshold = -0.49326
            leaf = [1. 0.]
            leaf = [0. 1.]
         feature = 1 threshold = -0.70172
            leaf = [1. 0.]
            leaf = [1. 0.]
```

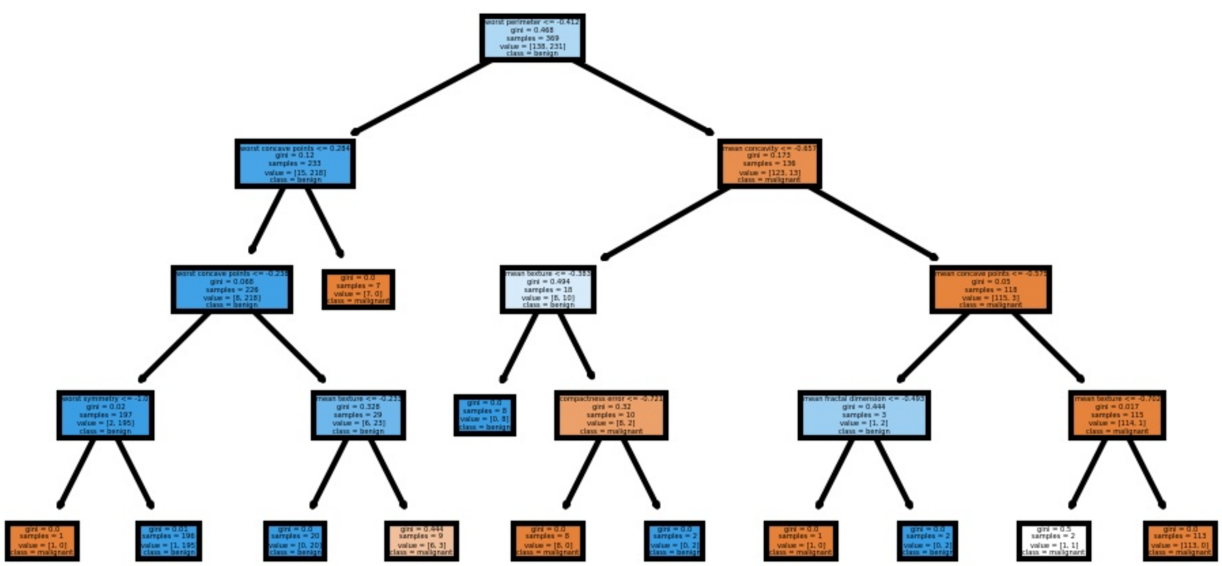


Figure 7: Cancer Tree Figure (Before 1 Hot Encoding) With Max Depth = 4

Table 4:

Dataset: Cancer			
scikit learn score = 94.50000 %			
max depth = 4			
Window	Polynom Degree	Runtime (sec)	Score (%)
0	0	0.0788	37
	1	0.0798	75.5
	2	0.0927	
	3	0.1087	92
	4	0.1077	
	5	0.1197	92.5
	6	0.1137	
	7	0.1117	95.5
	8	0.1137	
	9	0.1177	
	10	0.1027	
	11	0.1137	96
	12	0.1117	
	13	0.1167	96.5
	14	0.1117	
	15	0.1227	97
	16	0.1206	
	17	0.1316	96.5
	18	0.1326	
	19	0.1247	
	20	0.1426	
	21	0.1416	96
	22	0.1326	
	23	0.1277	96.5
	24	0.1526	
	25	0.1307	96
	26	0.1735	
	27	0.1706	
	28	0.167	
	29	0.128	
	30	0.148	
	31	0.152	
	32	0.156	
	33	0.144	

Table 5:

Dataset: Cancer			
scikit learn score =94.50000 %			
max depth = 4			
Window	Polynom Degree	Runtime (sec)	Score (%)
0.25	0	0.084	37
	1	0.08	75.5
	2	0.104	
	3	0.1	92
	4	0.092	
	5	0.104	92.5
	6	0.104	
	7	0.092	95.5
	8	0.116	
	9	0.096	
	10	0.1	
	11	0.12	96
	12	0.104	
	13	0.12	96.5
	14	0.128	
	15	0.108	97
	16	0.12	
	17	0.12	96.5
	18	0.144	
	19	0.128	
	20	0.128	
	21	0.124	96
	22	0.136	
	23	0.144	96.5
	24	0.124	
	25	0.164	96
	26	0.136	
	27	0.14	
	28	0.136	
	29	0.144	
	30	0.132	
	31	0.152	
	32	0.168	
	33	0.16	

Table 6:

Dataset: Cancer			
scikit learn score = 94.50000 %			
max depth = 4			
Window	Polynom Degree	Runtime (sec)	Score (%)
0.5	0	0.08	37
	1	0.104	75.5
	2	0.08	
	3	0.104	92
	4	0.092	
	5	0.084	92.5
	6	0.112	
	7	0.1	95.5
	8	0.096	
	9	0.12	
	10	0.096	
	11	0.104	96
	12	0.124	
	13	0.1	96.5
	14	0.1	
	15	0.2	97
	16	0.128	
	17	0.132	96.5
	18	0.116	
	19	0.136	
	20	0.124	
	21	0.12	96
	22	0.136	
	23	0.14	96.5
	24	0.14	
	25	0.136	96
	26	0.132	
	27	0.136	
	28	0.14	
	29	0.156	
	30	0.148	
	31	0.16	
	32	0.148	
	33	0.152	

Table 7:

Dataset: Cancer			
scikit learn score = 94.50000 %			
max depth = 4			
Window	Polynom Degree	Runtime (sec)	Score (%)
0.75	0	0.084	37
	1	0.1	75.5
	2	0.08	
	3	0.108	92
	4	0.088	
	5	0.084	92.5
	6	0.104	
	7	0.104	95.5
	8	0.092	
	9	0.116	
	10	0.104	
	11	0.112	96
	12	0.104	
	13	0.144	96.5
	14	0.12	
	15	0.12	97
	16	0.112	
	17	0.124	96.5
	18	0.128	
	19	0.128	
	20	0.124	
	21	0.14	96
	22	0.156	
	23	0.1721	96.5
	24	0.1728	
	25	0.1754	96
	26	0.1767	
	27	0.1746	
	28	0.2059	
	29	0.1549	
	30	0.156	
	31	0.136	
	32	0.156	
	33	0.148	

5.2 Results For Iris Data-Set

Iris Tree:

feature = 2 threshold = -0.45762
leaf = [1. 0. 0.]
feature = 2 threshold = 0.30508
feature = 3 threshold = 0.29166
leaf = [0. 1. 0.]
feature = 1 threshold = -0.08333
leaf = [0. 0. 1.]
leaf = [0. 1. 0.]
feature = 3 threshold = 0.33333
feature = 0 threshold = -0.02777
leaf = [0. 1. 0.]
leaf = [0. 0. 1.]
leaf = [0. 0. 1.]

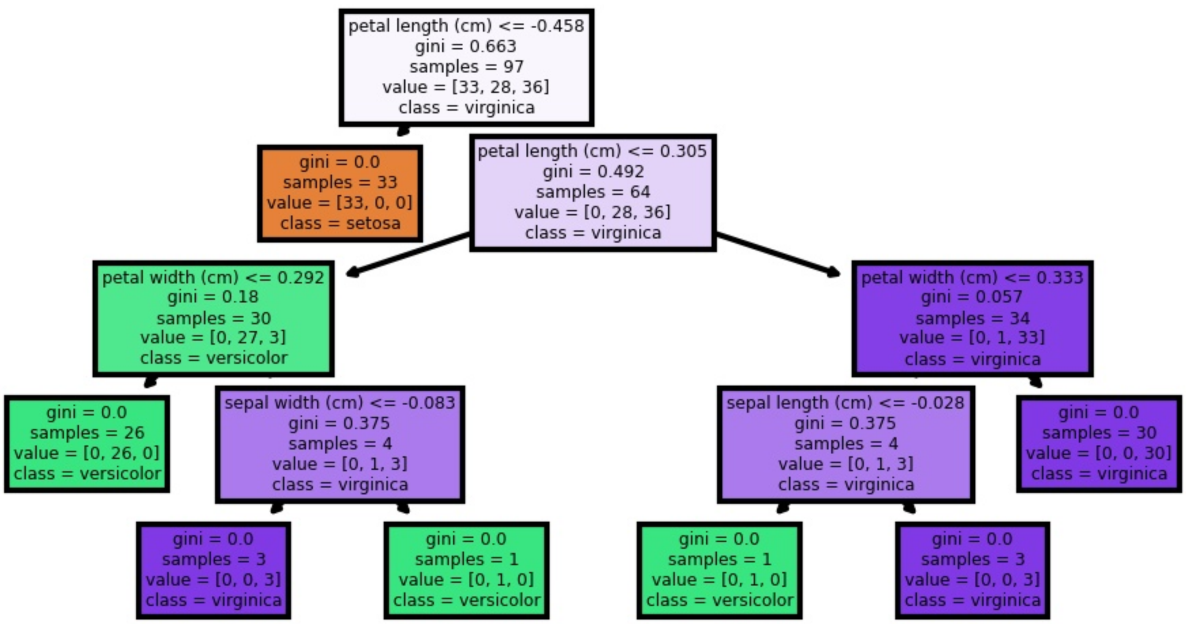


Figure 8: Iris Tree Figure (Before 1 Hot Encoding) With Max Depth = 4

Table 8:

Dataset: Iris			
scikit learn score = 92.45283%			
max depth = 4			
Window	Polynom Degree	Runtime (sec)	Score (%)
0	0	0.018	32.0755
	1	0.015	90.566
	2	0.018	
	3	0.019	92.4528
	4	0.016	
	5	0.0279	
	6	0.018	
	7	0.017	
	8	0.017	
	9	0.016	94.3396
	10	0.015	
	11	0.0159	
	12	0.015	
	13	0.0189	92.4528
	14	0.019	
	15	0.0239	
	16	0.0199	
	17	0.0249	
	18	0.018	
	19	0.0179	
	20	0.0209	
	21	0.0189	
	22	0.0209	
	23	0.0249	
	24	0.0309	
	25	0.0259	
	26	0.024	
	27	0.0219	
	28	0.0209	
	29	0.0219	
	30	0.02	
	31	0.0219	
	32	0.0299	
	33	0.0349	

Table 9:

Dataset: Iris			
scikit learn score = 92.45283%			
max depth = 4			
Window	**Polynom Degree**	**Runtime (sec)**	**Score (%)**
0.25	0	0.0439	32.0755
	1	0.0259	90.566
	2	0.017	
	3	0.0149	92.4528
	4	0.016	
	5	0.0289	
	6	0.0339	
	7	0.0229	
	8	0.0289	
	9	0.0299	94.3396
	10	0.0369	
	11	0.0219	
	12	0.0279	
	13	0.0439	92.4528
	14	0.0329	
	15	0.0489	
	16	0.0379	
	17	0.0349	
	18	0.0219	
	19	0.0189	
	20	0.0209	
	21	0.0219	
	22	0.0209	
	23	0.0489	
	24	0.1007	
	25	0.0718	
	26	0.0349	
	27	0.0349	
	28	0.0479	
	29	0.0329	
	30	0.0449	
	31	0.0349	
	32	0.0559	
	33	0.0499	

Table 10:

Dataset: Iris			
scikit learn score = 92.45283%			
max depth = 4			
Window	Polynom Degree	Runtime (sec)	Score (%)
0.5	0	0.0429	32.0755
	1	0.022	90.566
	2	0.0229	
	3	0.0209	92.4528
	4	0.0199	
	5	0.0219	
	6	0.0239	
	7	0.0219	
	8	0.0329	
	9	0.0249	94.3396
	10	0.0249	
	11	0.0379	
	12	0.0349	
	13	0.0299	92.4528
	14	0.0239	
	15	0.0239	
	16	0.0319	
	17	0.0279	
	18	0.0219	
	19	0.0219	
	20	0.0199	
	21	0.0189	
	22	0.0748	
	23	0.1237	
	24	0.0219	
	25	0.0259	
	26	0.0229	
	27	0.0279	
	28	0.0249	
	29	0.0239	
	30	0.0279	
	31	0.0349	
	32	0.026	
	33	0.0209	

Table 11:

Dataset: Iris			
scikit learn score = 92.45283%			
max depth = 4			
Window	Polynom Degree	Runtime (sec)	Score (%)
0.75	0	0.0259	32.0755
	1	0.0319	90.566
	2	0.0269	
	3	0.0279	92.4528
	4	0.0309	
	5	0.0299	
	6	0.0269	
	7	0.0239	
	8	0.0199	
	9	0.0319	94.3396
	10	0.0239	
	11	0.0199	
	12	0.0279	
	13	0.0219	92.4528
	14	0.02	
	15	0.0249	
	16	0.0229	
	17	0.0239	
	18	0.0229	
	19	0.0219	
	20	0.0239	
	21	0.0239	
	22	0.0219	
	23	0.0349	
	24	0.0369	
	25	0.0419	
	26	0.0299	
	27	0.0279	
	28	0.0289	
	29	0.0369	
	30	0.0399	
	31	0.0309	
	32	0.0269	
	33	0.0289	

5.3 Results For Wine Data-Set

Wine Tree:

feature = 6 threshold = -0.16877
    feature = 9 threshold = -0.56569
        leaf = [0. 1. 0.]
        feature = 6 threshold = -0.55274
            leaf = [0. 0. 1.]
            feature = 10 threshold = -0.61788
                leaf = [0. 0. 1.]
                leaf = [0. 1. 0.]
    feature = 12 threshold = -0.36305
        feature = 1 threshold = 0.23320
            leaf = [0. 1. 0.]
            leaf = [1. 0. 0.]
        feature = 4 threshold = 0.42391
            leaf = [1. 0. 0.]
            leaf = [0. 1. 0.]

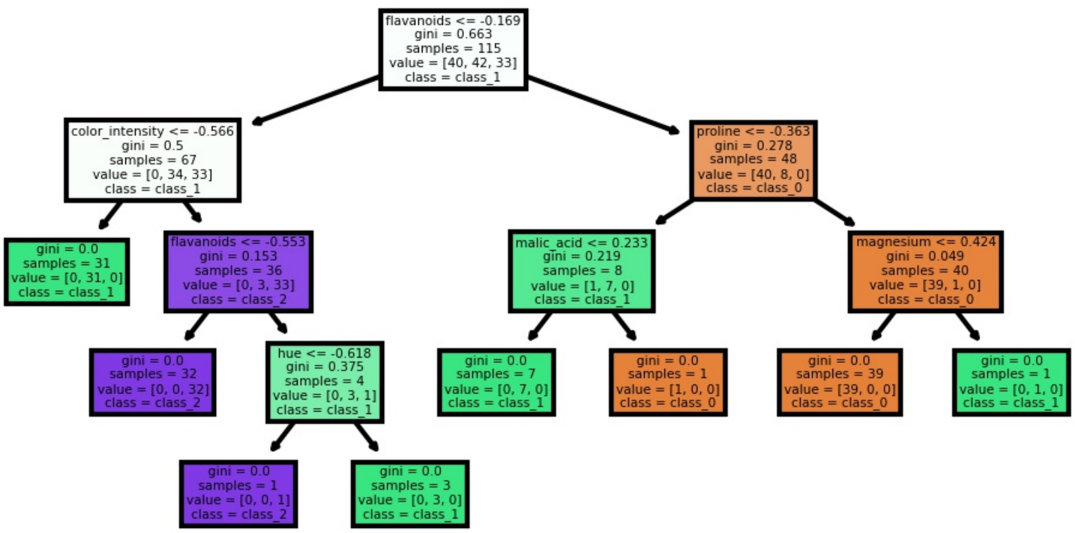


Figure 9: Wine Tree Figure (Before 1 Hot Encoding) With Max Depth = 4

Table 12:

Dataset: Wine			
scikit learn score = 93.65079 %			
max depth = 4			
Window	Polynom Degree	Runtime (sec)	Score (%)
0	0	0.0152	46.0317
	1	0.0112	55.5556
	2	0.0126	
	3	0.0116	76.1905
	4	0.0126	
	5	0.012	84.127
	6	0.0122	
	7	0.0124	92.0635
	8	0.0127	
	9	0.0247	
	10	0.0225	
	11	0.0203	95.2381
	12	0.0176	
	13	0.0178	
	14	0.0175	
	15	0.0157	
	16	0.0145	
	17	0.0147	
	18	0.0148	
	19	0.0152	
	20	0.0155	
	21	0.022	
	22	0.0219	
	23	0.0193	
	24	0.0225	
	25	0.0235	
	26	0.0178	
	27	0.0167	
	28	0.0166	
	29	0.0172	
	30	0.0173	
	31	0.0173	
	32	0.0233	
	33	0.0226	
	34	0.0234	

Table 13:

Dataset: Wine			
scikit learn score = 93.65079 %			
max depth = 4			
Window	Polynom Degree	Runtime (sec)	Score (%)
0.25	0	0.0159	46.0317
	1	0.0135	55.5556
	2	0.0146	
	3	0.0131	76.1905
	4	0.012	
	5	0.0124	84.127
	6	0.0123	
	7	0.0126	92.0635
	8	0.0131	
	9	0.0131	
	10	0.0154	
	11	0.0196	95.2381
	12	0.0175	
	13	0.0187	
	14	0.0202	
	15	0.0187	
	16	0.017	
	17	0.0148	
	18	0.0149	
	19	0.0153	
	20	0.0153	
	21	0.0152	
	22	0.0221	
	23	0.0205	
	24	0.0226	
	25	0.0205	
	26	0.02	
	27	0.0169	
	28	0.0167	
	29	0.0173	
	30	0.0176	
	31	0.0174	
	32	0.0175	
	33	0.0254	
	34	0.027	

Table 14:

Dataset: Wine			
scikit learn score = 93.65079 %			
max depth = 4			
Window	Polynom Degree	Runtime (sec)	Score (%)
0.5	0	0.0147	46.0317
	1	0.0259	55.5556
	2	0.025	
	3	0.0165	76.1905
	4	0.0123	
	5	0.0128	84.127
	6	0.0132	
	7	0.0134	92.0635
	8	0.0136	
	9	0.0136	
	10	0.0212	
	11	0.0159	95.2381
	12	0.0165	
	13	0.0182	
	14	0.0173	
	15	0.0238	
	16	0.0153	
	17	0.0152	
	18	0.0156	
	19	0.016	
	20	0.0162	
	21	0.0206	
	22	0.0191	
	23	0.0214	
	24	0.0243	
	25	0.0215	
	26	0.0185	
	27	0.0178	
	28	0.0174	
	29	0.0176	
	30	0.0179	
	31	0.0182	
	32	0.0249	
	33	0.0227	
	34	0.0202	

Table 15:

Dataset: Wine			
scikit learn score = 93.65079 %			
max depth = 4			
Window	Polynom Degree	Runtime (sec)	Score (%)
0.75	0	0.0122	46.0317
	1	0.0144	55.5556
	2	0.0149	
	3	0.0125	76.1905
	4	0.0119	
	5	0.0121	84.127
	6	0.0121	
	7	0.0124	92.0635
	8	0.0126	
	9	0.0129	
	10	0.013	
	11	0.0175	95.2381
	12	0.0185	
	13	0.0197	
	14	0.0173	
	15	0.0184	
	16	0.0211	
	17	0.0165	
	18	0.0163	
	19	0.0161	
	20	0.0159	
	21	0.016	
	22	0.0161	
	23	0.0198	
	24	0.0217	
	25	0.0219	
	26	0.023	
	27	0.0239	
	28	0.0172	
	29	0.0178	
	30	0.0173	
	31	0.0176	
	32	0.0178	
	33	0.0258	
	34	0.0243	

5.4 Conclusions

Finally we derive from the results above, the desired conclusion:
⋆The greater the tree's depth, the more branched which leads to a higher score.
⋆For all the 4 windows: [0, 0.25, 0.5, 0.75] that sent to the polynom we got the same accuracy results for all polynoms from degree 0 to degree 33, but they different in the running time.
⋆The greater the window, the higher the runtime by an approx difference of 1%.
⋆The greater the polynom's degree, the higher the score for Algorithm 1. Occasionally happens that the score is decreased by at most 3%.
⋆After several code runs, the average polynomial degree that gave us the best score is 15.
⋆According to Figure 4,5 and 6 we can observe that our score results are similar to scikit-learn results. For example, in Figure 5, when the tree's depth is in range [3,5] both results are identical.
⋆Since Cancer tree is more branched, we can observe that the accuracy score is higher than other's datasets.

6 Platforms

- Jupyter lab
- pycharm, python 3.9
- overleaf.com
- excel

Part C - Privacy Preserving Decision Tree Prediction On Encrypted Data

August 20, 2021

Contents

Abstract.
In part C of the project we used secure protocols for prediction and training of tree based models where the data for prediction is encrypted with FHE.

1 Introduction

Reference to our Github repo.
Part C is an extension to prediction on encrypted data. We implement the prediction by using an adjustment of Algorithm 1.
The adjustment is this subroutine which operates over encrypted data :

Subroutine $\mathsf{Enc\_Predict}(v, \mathbf{c_x})$ where v is a node in T, and $\mathbf{c_x}$ is a vector of L ciphertexts.

1. If v is not a leaf, homomorphically evaluate the following formula (using Eval_{pk}) and return the resulting ciphertext:

$$\phi(\mathbf{c_x}[v.feature] - v.\theta) \cdot \mathsf{Enc\_Predict}(v.right, \mathbf{c_x}) + \\ \phi(v.\theta - \mathbf{c_x}[v.feature]) \cdot \mathsf{Enc\_Predict}(v.left, \mathbf{c_x})$$

2. Otherwise return $v.leaf\_value$.

Figure 1:
The subroutine Enc-Predict

The protocols that we used are an adaptation of the algorithms in part B but with interactive settings where data is encrypted throughout the computation.
See the protocol for prediction over encrypted data in Figures 3 and 4 in Section 4.1 in Akavia's paper.

2 FHE - Fully Homomorphic Encryption

Homomorphic Encryption provides the ability to compute on encrypted data.

These resulting computations are left in an encrypted form which, when decrypted, result in an identical output to that produced had the operations been performed on the 'original' data.

This ground-breaking technology has enabled industry to provide capabilities for outsourced computation securely (Homomorphic encryption can be used for privacy-preserving outsourced storage and computation. This allows data to be encrypted and out-sourced to commercial cloud environments for processing, all while encrypted).

Homomorphic encryption includes multiple types of encryption schemes that can perform different classes of computations over encrypted data.

The computations are represented as either Boolean or arithmetic circuits. Some common types of homomorphic encryption are partially homomorphic, somewhat homomorphic, leveled fully homomorphic, and fully homomorphic encryption.

We will use the fully homomorphic encryption scheme, which allows the evaluation of arbitrary circuits composed of multiple types of gates of unbounded depth, and is the strongest notion of homomorphic encryption. Such a scheme enables the construction of programs for any desirable functionality, which can be run on encrypted inputs to produce an encryption of the result. Since such a program need never decrypt its inputs, it can be run by an untrusted party without revealing its inputs and internal state.

2.1 Understanding HE

- **"Homomorphic":** a mapping from plaintext space to ciphertext space that preserves arithmetic operations.

- **Mathematical Hardness:** Learning with Errors Assumption; every image (ciphertext) of this mapping looks uniformly random in range (ciphertext space).

- **"Security level":** the hardness of inverting this mapping without the secret key.
  Example: 128 bits → 2128 operations to break

- **public key (pk):** a key that can be obtained and used by anyone to encrypt messages intended for a particular recipient.

- **secret key (or "private key"):** a piece of information or a framework that is used to decrypt and encrypt messages. The encrypted messages can be deciphered only by using a second key that is known only to the recipient (the private key).

- **Plaintext** elements and operations of a polynomial ring.
  Example: $3x^5 + x^4 + 2x^3 + ...$

- **Ciphertext** elements and operations of a polynomial ring.
  Example: $7862x^5 + 5652x^4 + ...$

2.2 CKKS Scheme

CKKS scheme supports efficient rounding operations in encrypted state. The rounding operation controls noise increase in encrypted multiplication, which reduces the number of bootstrapping in a circuit. An important characteristic of CKKS scheme that encrypts approximate values rather than exact values. When computers store real-valued data, they remember approximate values with long significant bits, not real values exactly. CKKS scheme is constructed to deal efficiently with the errors arising from the approximations. The scheme is familiar to machine learning which has inherent noises in its structure.

2.3 STANDARDIZATION

There are several reasons why this is the right time to standardize homomorphic encryption.

- There is a need for easily available secure computation technology as more companies and individuals switch to cloud storage and computing. Homomorphic encryption is already ripe for mainstream use, but the current lack of standardization is making it difficult to start using it.

- Implementations of leading schemes (CKKS, BFV...) have been adopted to address the needs of privacy-protected computations.

Several open-source implementations of homomorphic encryption schemes exist today, we used Microsoft SEAL: A widely used open-source library from Microsoft that supports the BFV and CKKS schemes.

2.4 TenSeal

TenSEAL is a library for homomorphic encrypting operations on tensors, built on top of Microsoft SEAL. It provides ease of use through a Python API, while preserving efficiency by implementing most of its operations using C++.

3 Implementation Details

We start by creating a TenSEAL Context for specifying the scheme and the parameters we are going to use.

3.1 TenSEAL CKKS Context

```
context = ts.context(ts.SCHEME_TYPE.CKKS, 8192, coeff_mod_bit sizes=[40,21,21,21,21,21,
```

which specifies:

- scheme type: ts.SCHEME_TYPE.CKKS
- poly_modulus_degree: 8192.
- coeff_mod_bit_sizes: The coefficient modulus sizes, here [60, 40, 40, 60]. This means that the coefficient modulus will contain 8 primes of 40 bits, 6 primes of 21, and last one is 40 bits.
- global_scale: the scaling factor, here set to 2^{21}.
- generate_galois_keys: Galois keys are another type of public keys needed to perform encrypted vector rotation operations on batched ciphertexts.

3.1.1 Code

```
context = ts.context(ts.SCHEME_TYPE.CKKS, poly_modulus_degree=2 ** 13,
coeff_mod_bit_sizes=[40, 21, 21, 21, 21, 21, 21, 40]
)
context.generate_galois_keys()
context.global_scale = 2 ** 21

sk = context.secret_key()
context.make_context_public()
```

3.2 Trainset Encryption

```
ts.ckks_tensor(pk, vector)
```

The operation encodes vectors of complex or real numbers into plaintext polynomials to be encrypted and computed using the CKKS scheme. It converts a plaintext polynomial to a ciphertext.
We encrypt each element of the given trainset using the above command, which specifies:

- pk is public key used for encryption.
- vector is the element getting encrypted.

3.2.1 Code

```
def trainset_enc(X_test, context):
pk = context.copy()
encrypted_Xtest = []
for vec in X_test:
    print(vec)
    encrypted_Xtest.append(ts.ckks_tensor(pk, vec))
return encrypted_Xtest
```

3.3 Trainset Decryption

```
decrypt(sk).tolist()
```

- sk is secret key used for decryption.

3.3.1 Code

```
for i in range(len(X_test_encrypted)):
    decrypted_X_test = X_test_encrypted[i].decrypt(sk).tolist()
```

4 Algorithm

4.1 Code

```
def Tree_Predict_partC(T, x, phi):
if T is None:
    return
feature, threshold, leaf, left, right = T.getNode()
feature = int(abs(feature))
if isinstance(leaf, np.ndarray):
    return leaf
else:
    return (x[feature] - threshold).polyval(phi) * Tree_Predict_partC(right, x, phi
        (threshold - x[feature])).polyval(phi) * Tree_Predict_partC(left, x, phi)
```

Akavia's algorithm traverses all paths in the tree and computes a weighted combination of all the leaves values, where each leaf value is the 1-hot encoding of the label associated with the leaf. The output is a length L vector assigning a likelihood score to each label, which is in turn interpreted as outputting the label with the highest score.
Here we used polyval which is a polynomial evaluation with an encrypted tensor as variable.

5 Accuracy

The accuracy is calculated as the percentage of correct classification on test samples.

```
algorithm_score = (counter / len(res_vec))*100
```

Counter is equal to the sum of correct classifications and res_vec is equal to the sum of all classifications.
res_vec is the result of the encrypted test set prediction after decrypt and one hot encoding.
To know if current classification is correct or not we compare it to Y target value of the relevant sample if they are equal then the prediction of our algorithm is correct .

Test samples are decided to be sent to algorithm as 35% of the samples in the data-set and they are chosen randomly from it.
Our algorithm predict over all test samples, after that we calculate accuracy according to the percentage of correct predictions on test samples.

5.1 Code

```
def calc_accuracy_partC(res_vec, Y_test):
counter = 0
if len(res_vec[0]) == 3:
    for i in range(len(res_vec)):
        if np.logical_and(res_vec[i] == [1, 0, 0], Y_test[i] == 0).all() or np.logi
            counter += 1
if len(res_vec[0]) == 2:
```

```
        for i in range(len(res_vec)):
            if np.logical_and(res_vec[i] == [1, 0], Y_test[i] == 0).all() or np.logical
                counter += 1
    return (counter / len(res_vec)) * 100
```

6 Results

Dataset : Iris
Max depth : 4
Polynom degree : 8
Test size : 30 samples which is 0.2 of the data
Prediction accuracy = 93.33%

```
X_test =
    [[-1.66666667e-01  -4.16666667e-01   5.08474576e-02  -2.50000000e-01]
    [-5.00000000e-01   7.50000000e-01  -8.30508475e-01  -1.00000000e+00]
    [-2.22222222e-01  -5.83333333e-01   3.55932203e-01   5.83333333e-01]
    [-7.22222222e-01   1.66666667e-01  -7.96610169e-01  -9.16666667e-01]
    [-6.11111111e-01   1.66666667e-01  -8.30508475e-01  -9.16666667e-01]
    [-2.77777778e-01  -1.66666667e-01   1.86440678e-01   1.66666667e-01]
    [-6.11111111e-01   0.00000000e+00  -9.32203390e-01  -9.16666667e-01]
    .
    .
    .
    [-3.33333333e-01  -7.50000000e-01   1.69491525e-02   2.22044605e-16]
    [-2.77777778e-01  -3.33333333e-01   3.22033898e-01   5.83333333e-01]]

Y_test = [1 0 2 0 0 1 0 ... 1 2]

X_test encrypted =
    [<tenseal.tensors.ckkstensor.CKKSTensor object at 0x0000024951BEA880>,
    <tenseal.tensors.ckkstensor.CKKSTensor object at 0x0000024951BEA970>,
    <tenseal.tensors.ckkstensor.CKKSTensor object at 0x00000249523451C0>,
    <tenseal.tensors.ckkstensor.CKKSTensor object at 0x00000249523451F0>,
    <tenseal.tensors.ckkstensor.CKKSTensor object at 0x0000024952345250>,
    <tenseal.tensors.ckkstensor.CKKSTensor object at 0x0000024952345280>,
    <tenseal.tensors.ckkstensor.CKKSTensor object at 0x00000249523453D0>,
    .
    .
    .
    <tenseal.tensors.ckkstensor.CKKSTensor object at 0x0000024952345BE0>,
    <tenseal.tensors.ckkstensor.CKKSTensor object at 0x0000024952345C40>]

            ************ Prediction Results *************

run time for sample 1 = 15.7883 seconds
predict result for sample 1 =
[0.019447354196179698, 2.864138237961886, 0.6180982373065784]
```

```
run time for sample 2 = 15.5750 seconds
predict result for sample 2 =
[1.1053998903954065, 0.6059977397966027, -0.04298346622914129]

run time for sample 3 = 15.4051 seconds
predict result for sample 3 =
[-0.03223200151966879, 0.4562844833009126, 3.1470747547875613]

run time for sample 4 =15.2020 seconds
predict result for sample 4 =
[1.069321814007992, 0.6582081276819615, 0.002301989696042386]

run time for sample 5 =15.3733 seconds
predict result for sample 5 =
[1.1151917307792774, 0.5585006469020478, -0.013459729121453651]

run time for sample 6 =15.2289 seconds
predict result for sample 6 =
[-0.03276452806617886, 1.7905640163393228, 1.8392962898575334]

run time for sample 7 = 15.1823 seconds
predict result for sample 7 =
[1.2114346531269258, 0.3197190243064703, -0.009674884820227301]
.
.
.
run time for sample 29 = 15.2714 seconds
predict result for sample 29 =
[0.04109668749486619, 2.5464832814579212, 0.895996705268293]

run time for sample 30 = 15.2293 seconds
predict result for sample 30 =
[-0.0240047171361257, 0.4942109706631339, 3.061742831732956]

one hot encoding, prediction result = [
array 1 = ([0., 1., 0.]),
array 2 = ([1., 0., 0.]),
array 3 = ([0., 0., 1.]),
array 4 = ([1., 0., 0.]),
array 5 = ([1., 0., 0.]),
array 6 = ([0., 0., 1.]),
array 7 = ([1., 0., 0.]),
.
.
.
array 29 = ([0., 1., 0.]),
array 30 = ([0., 0., 1.])]
```

Dataset : Cancer
Max depth : 4
Polynom degree : 8
Test size : 36 samples which is 0.2 of the data
Prediction accuracy = 88.88%

```
X_test =
[[-0.58947368 -0.45454545  0.47593583  0.12371134  0.39130435 -0.57241379
  -0.7257384  -0.96226415 -0.27444795 -0.79180887 -0.23577236 -0.27472527
  -0.50499287]
 [-0.77894737 -0.34387352  0.13368984 -0.03092784 -0.43478261  0.32413793
   0.03375527 -0.28301887 -0.10410095 -0.66382253 -0.4796748   0.55311355
  -0.50499287]
 [ 0.07894737  0.24901186  0.06951872  0.12371134 -0.06521739 -0.70344828
  -0.55696203 -0.20754717 -0.53943218  0.38566553 -0.85365854 -0.95604396
  -0.61198288]
 [-0.12631579 -0.68774704 -0.03743316  0.04123711 -0.7826087  -0.72413793
  -0.52742616  0.69811321 -0.23659306 -0.69795222 -0.2195122  -0.42124542
  -0.69044223]
 [-0.11052632 -0.60079051 -0.01604278  0.22680412 -0.69565217 -0.72413793
  -0.40084388  0.32075472 -0.23028391 -0.6552901  -0.3495935  -0.15750916
  -0.70042796]
 [-0.35789474  0.57312253  0.26203209  0.07216495 -0.58695652 -0.72413793
  -0.94514768  0.50943396 -0.75394322 -0.56143345 -0.56097561 -1.
  -0.36947218]
 [ 0.34210526 -0.63636364  0.06951872 -0.12371134 -0.2173913   0.29655172
   0.20253165 -0.66037736 -0.02839117 -0.04095563 -0.00813008  0.17948718
   0.76462197]
.
.
.
.
[ 0.72105263 -0.53359684  0.45454545 -0.03092784  0.08695652  0.25517241
   0.1814346  -0.24528302 -0.01577287 -0.16040956 -0.04065041  0.01098901
   0.42938659]
 [-0.38421053 -0.09486166  0.02673797 -0.13402062 -0.43478261 -0.8137931
  -0.93670886  0.01886792 -0.79810726 -0.27986348 -0.70731707 -0.58974359
  -0.66904422]]

Y_test =

X_test encrypted =
[<tenseal.tensors.ckkstensor.CKKSTensor object at 0x0000023917D1B880>,
<tenseal.tensors.ckkstensor.CKKSTensor object at 0x0000023917D1B820>,
<tenseal.tensors.ckkstensor.CKKSTensor object at 0x0000023917D0A0D0>,
<tenseal.tensors.ckkstensor.CKKSTensor object at 0x000002397EDDBFD0>,
<tenseal.tensors.ckkstensor.CKKSTensor object at 0x0000023917D3A190>,
<tenseal.tensors.ckkstensor.CKKSTensor object at 0x0000023917D3AF70>,
<tenseal.tensors.ckkstensor.CKKSTensor object at 0x0000023917CC63D0>,
.
```

```
.
.
<tenseal.tensors.ckkstensor.CKKSTensor object at 0x0000023917D55BB0>,
<tenseal.tensors.ckkstensor.CKKSTensor object at 0x0000023917D55C10>]


            ************ Prediction Results *************

run time for sample 1 = 15.7645 seconds
predict result for sample 1 =
[0.31465705041981856, 1.4683726550171892, 0.34614773070371824]

run time for sample 2 = 15.4364 seconds
predict result for sample 2 =
[0.5331141271745162, 2.06354218818059, 0.00022079015987372863]

run time for sample 3 = 14.7958 seconds
predict result for sample 3 =
[0.526916049576809, 1.242715732828378, 1.0157657833612563]

run time for sample 4 = 15.0301 seconds
predict result for sample 4 =
[0.10977959931042937, 1.8602146637132575, 0.3252765061893235]
.
.
.
one hot encoding, prediction result = [
array 1 = ([0., 1., 0.]),
array 2 = ([0., 1., 0.]),
array 3 = ([0., 1., 0.]),
array 4 = ([0., 1., 0.]),
array 5 = ([0., 1., 0.]),
.
.
.
]
```

6.1 Conclusions

We got high accuracy for prediction with datasets that is encrypted with fully homomorphic encryption, in comparison to standard algorithms on cleartext data.

Prediction is slower (minutes to hours). But, the protocols we used support real-life enterprise use-cases.

7 Platforms

- pycharm, python 3.9
- overleaf.com

Part D - Prediction with NN

August 20, 2021

Contents

Abstract. In part D we implement the prediction of iris species using perceptron neural networks trained on the Iris data set. It includes write-ups of the different types of perceptrons and their accuracy.

Keywords— Perceptron: either a single-layer or mutli-layer feed-forward neural network.

1 Introduction

Reference to our Github repo.
In part D we'll describe the prediction of iris species using two different perceptron neural networks: a single-layer and multi-layer perceptron.
Each perceptron is trained and evaluated on the Iris data set split into train and test sets.

2 ARCHITECTURE

The main architecture of the neural networks are based on a single-layer perceptron and a multi-layer perceptron.
Each network was trained using the categorical cross entropy loss function, since the problem consists of multiple classes, and the Adam optimizer, due to its practical advantage over alternatives.
The Iris dataset was split beforehand into train and test sets. It consists of samples of 4 features, sepal length, sepal width, petal length, petal width, with 1 of 3 classes, 1.setosa, 2.versicolor, 3.virginica.

3 Single-layer Perceptron

The single-layer perceptron is modeled with an input layer of 4 nodes, one for each feature, and an output layer of 3 nodes, one for each class, with a softmax activation function, since the problem consists of multiple classes.
This was chosen to be a benchmark to see the performance increase with the multi-layer perceptron.
This single-layer perceptron should have a very low accuracy due to its extreme simplicity.

4 Multi-layer Perceptron

The multi-layer perceptron is modeled with an input layer of 4 nodes, one for each feature, 2 hidden layers of 10 nodes with a ReLU activation function, and an output layer of 3 nodes, one for each class, with a softmax activation function, since the problem consists of multiple classes. ReLU was used due to its practical advantage seen in research papers. Theoretically, the 2 hidden layers should be able to learn higher abstract information that the single-layer perceptron could not model. Thus, it is expected to produce a higher accuracy than the single-layer perceptron.

In addition to the change in the network architecture, the multi-layer perceptron also received proprocessed data, that is, data scaled down to the range of -1 to 1. This was done exclusively for the features as the classes are simply represented as a 0 or 1 and thus do not need any scaling. Theoretically, this change should allow the network to train faster as it does not need to give priority to training features that are high in value.

Finally, after each layer in the network, batch normalization was run to further allow the network to train faster and better.

5 RESULTS

```
Epoch 1/100
5/5 [==============================] - 1s 0s/step - loss: 5.1102 - accuracy: 0.3258
Epoch 2/100
5/5 [==============================] - 0s 4ms/step - loss: 5.0277 - accuracy: 0.3258
Epoch 3/100
5/5 [==============================] - 0s 0s/step - loss: 4.9489 - accuracy: 0.3258
```

```
Epoch 4/100
5/5 [==============================] - 0s 4ms/step - loss: 4.8675 - accuracy: 0.3258
Epoch 5/100
5/5 [==============================] - 0s 2ms/step - loss: 4.7872 - accuracy: 0.3258
Epoch 6/100
5/5 [==============================] - 0s 0s/step - loss: 4.7102 - accuracy: 0.3258
Epoch 7/100
5/5 [==============================] - 0s 4ms/step - loss: 4.6319 - accuracy: 0.3258
Epoch 8/100
5/5 [==============================] - 0s 0s/step - loss: 4.5566 - accuracy: 0.3258
Epoch 9/100
5/5 [==============================] - 0s 4ms/step - loss: 4.4817 - accuracy: 0.3258
Epoch 10/100
5/5 [==============================] - 0s 0s/step - loss: 4.4088 - accuracy: 0.3258
Epoch 11/100
5/5 [==============================] - 0s 4ms/step - loss: 4.3343 - accuracy: 0.3258
Epoch 12/100
5/5 [==============================] - 0s 0s/step - loss: 4.2627 - accuracy: 0.3258
Epoch 13/100
5/5 [==============================] - 0s 0s/step - loss: 4.1905 - accuracy: 0.3258
Epoch 14/100
5/5 [==============================] - 0s 0s/step - loss: 4.1239 - accuracy: 0.3258
Epoch 15/100
5/5 [==============================] - 0s 0s/step - loss: 4.0564 - accuracy: 0.3258
Epoch 16/100
5/5 [==============================] - 0s 4ms/step - loss: 3.9927 - accuracy: 0.3258
Epoch 17/100
5/5 [==============================] - 0s 0s/step - loss: 3.9270 - accuracy: 0.3258
Epoch 18/100
5/5 [==============================] - 0s 4ms/step - loss: 3.8681 - accuracy: 0.3258
Epoch 19/100
5/5 [==============================] - 0s 2ms/step - loss: 3.8133 - accuracy: 0.3258
Epoch 20/100
5/5 [==============================] - 0s 0s/step - loss: 3.7572 - accuracy: 0.3258
Epoch 21/100
5/5 [==============================] - 0s 0s/step - loss: 3.7059 - accuracy: 0.3258
Epoch 22/100
5/5 [==============================] - 0s 0s/step - loss: 3.6561 - accuracy: 0.3258
Epoch 23/100
5/5 [==============================] - 0s 0s/step - loss: 3.6096 - accuracy: 0.3258
Epoch 24/100
5/5 [==============================] - 0s 0s/step - loss: 3.5600 - accuracy: 0.3258
Epoch 25/100
5/5 [==============================] - 0s 0s/step - loss: 3.5156 - accuracy: 0.3258
Epoch 26/100
5/5 [==============================] - 0s 0s/step - loss: 3.4693 - accuracy: 0.3258
Epoch 27/100
5/5 [==============================] - 0s 0s/step - loss: 3.4267 - accuracy: 0.3258
Epoch 28/100
5/5 [==============================] - 0s 0s/step - loss: 3.3858 - accuracy: 0.3258
Epoch 29/100
5/5 [==============================] - 0s 0s/step - loss: 3.3485 - accuracy: 0.3258
```

```
Epoch 30/100
5/5 [==============================] - 0s 0s/step - loss: 3.3101 - accuracy: 0.3258
Epoch 31/100
5/5 [==============================] - 0s 5ms/step - loss: 3.2757 - accuracy: 0.3258
Epoch 32/100
5/5 [==============================] - 0s 2ms/step - loss: 3.2444 - accuracy: 0.3258
Epoch 33/100
5/5 [==============================] - 0s 2ms/step - loss: 3.2116 - accuracy: 0.3258
Epoch 34/100
5/5 [==============================] - 0s 2ms/step - loss: 3.1833 - accuracy: 0.3333
Epoch 35/100
5/5 [==============================] - 0s 857us/step - loss: 3.1499 - accuracy: 0.3333
Epoch 36/100
5/5 [==============================] - 0s 0s/step - loss: 3.1197 - accuracy: 0.3333
Epoch 37/100
5/5 [==============================] - 0s 0s/step - loss: 3.0895 - accuracy: 0.3333
Epoch 38/100
5/5 [==============================] - 0s 0s/step - loss: 3.0590 - accuracy: 0.3333
Epoch 39/100
5/5 [==============================] - 0s 0s/step - loss: 3.0295 - accuracy: 0.3333
Epoch 40/100
5/5 [==============================] - 0s 4ms/step - loss: 2.9974 - accuracy: 0.3333
Epoch 41/100
5/5 [==============================] - 0s 0s/step - loss: 2.9681 - accuracy: 0.3333
Epoch 42/100
5/5 [==============================] - 0s 0s/step - loss: 2.9407 - accuracy: 0.3409
Epoch 43/100
5/5 [==============================] - 0s 0s/step - loss: 2.9120 - accuracy: 0.3409
Epoch 44/100
5/5 [==============================] - 0s 0s/step - loss: 2.8824 - accuracy: 0.3409
Epoch 45/100
5/5 [==============================] - 0s 0s/step - loss: 2.8530 - accuracy: 0.3409
Epoch 46/100
5/5 [==============================] - 0s 0s/step - loss: 2.8241 - accuracy: 0.3409
Epoch 47/100
5/5 [==============================] - 0s 0s/step - loss: 2.7919 - accuracy: 0.3485
Epoch 48/100
5/5 [==============================] - 0s 4ms/step - loss: 2.7636 - accuracy: 0.3485
Epoch 49/100
5/5 [==============================] - 0s 0s/step - loss: 2.7353 - accuracy: 0.3561
Epoch 50/100
5/5 [==============================] - 0s 0s/step - loss: 2.7091 - accuracy: 0.3561
Epoch 51/100
5/5 [==============================] - 0s 0s/step - loss: 2.6801 - accuracy: 0.3561
Epoch 52/100
5/5 [==============================] - 0s 0s/step - loss: 2.6494 - accuracy: 0.3561
Epoch 53/100
5/5 [==============================] - 0s 0s/step - loss: 2.6184 - accuracy: 0.3561
Epoch 54/100
5/5 [==============================] - 0s 2ms/step - loss: 2.5881 - accuracy: 0.3561
Epoch 55/100
5/5 [==============================] - 0s 0s/step - loss: 2.5570 - accuracy: 0.3561
```

```
Epoch 56/100
5/5 [==============================] - 0s 4ms/step - loss: 2.5264 - accuracy: 0.3636
Epoch 57/100
5/5 [==============================] - 0s 0s/step - loss: 2.4954 - accuracy: 0.3712
Epoch 58/100
5/5 [==============================] - 0s 0s/step - loss: 2.4648 - accuracy: 0.3561
Epoch 59/100
5/5 [==============================] - 0s 0s/step - loss: 2.4381 - accuracy: 0.3561
Epoch 60/100
5/5 [==============================] - 0s 0s/step - loss: 2.4102 - accuracy: 0.3561
Epoch 61/100
5/5 [==============================] - 0s 4ms/step - loss: 2.3837 - accuracy: 0.3636
Epoch 62/100
5/5 [==============================] - 0s 0s/step - loss: 2.3581 - accuracy: 0.3712
Epoch 63/100
5/5 [==============================] - 0s 4ms/step - loss: 2.3293 - accuracy: 0.3712
Epoch 64/100
5/5 [==============================] - 0s 0s/step - loss: 2.3017 - accuracy: 0.3712
Epoch 65/100
5/5 [==============================] - 0s 4ms/step - loss: 2.2719 - accuracy: 0.3712
Epoch 66/100
5/5 [==============================] - 0s 0s/step - loss: 2.2419 - accuracy: 0.3864
Epoch 67/100
5/5 [==============================] - 0s 0s/step - loss: 2.2144 - accuracy: 0.3864
Epoch 68/100
5/5 [==============================] - 0s 0s/step - loss: 2.1865 - accuracy: 0.3864
Epoch 69/100
5/5 [==============================] - 0s 0s/step - loss: 2.1579 - accuracy: 0.3712
Epoch 70/100
5/5 [==============================] - 0s 4ms/step - loss: 2.1304 - accuracy: 0.3712
Epoch 71/100
5/5 [==============================] - 0s 0s/step - loss: 2.1048 - accuracy: 0.3636
Epoch 72/100
5/5 [==============================] - 0s 4ms/step - loss: 2.0768 - accuracy: 0.3561
Epoch 73/100
5/5 [==============================] - 0s 0s/step - loss: 2.0452 - accuracy: 0.3561
Epoch 74/100
5/5 [==============================] - 0s 0s/step - loss: 2.0167 - accuracy: 0.3636
Epoch 75/100
5/5 [==============================] - 0s 0s/step - loss: 1.9872 - accuracy: 0.3636
Epoch 76/100
5/5 [==============================] - 0s 0s/step - loss: 1.9575 - accuracy: 0.3636
Epoch 77/100
5/5 [==============================] - 0s 4ms/step - loss: 1.9277 - accuracy: 0.3864
Epoch 78/100
5/5 [==============================] - 0s 0s/step - loss: 1.8979 - accuracy: 0.3864
Epoch 79/100
5/5 [==============================] - 0s 4ms/step - loss: 1.8672 - accuracy: 0.3864
Epoch 80/100
5/5 [==============================] - 0s 2ms/step - loss: 1.8378 - accuracy: 0.3864
Epoch 81/100
5/5 [==============================] - 0s 0s/step - loss: 1.8088 - accuracy: 0.3864
```

```
Epoch 82/100
5/5 [==============================] - 0s 0s/step - loss: 1.7809 - accuracy: 0.3864
Epoch 83/100
5/5 [==============================] - 0s 0s/step - loss: 1.7501 - accuracy: 0.3864
Epoch 84/100
5/5 [==============================] - 0s 4ms/step - loss: 1.7235 - accuracy: 0.4091
Epoch 85/100
5/5 [==============================] - 0s 0s/step - loss: 1.6962 - accuracy: 0.4167
Epoch 86/100
5/5 [==============================] - 0s 4ms/step - loss: 1.6689 - accuracy: 0.4242
Epoch 87/100
5/5 [==============================] - 0s 0s/step - loss: 1.6422 - accuracy: 0.4394
Epoch 88/100
5/5 [==============================] - 0s 4ms/step - loss: 1.6151 - accuracy: 0.4470
Epoch 89/100
5/5 [==============================] - 0s 0s/step - loss: 1.5876 - accuracy: 0.4470
Epoch 90/100
5/5 [==============================] - 0s 4ms/step - loss: 1.5601 - accuracy: 0.4621
Epoch 91/100
5/5 [==============================] - 0s 0s/step - loss: 1.5331 - accuracy: 0.4621
Epoch 92/100
5/5 [==============================] - 0s 2ms/step - loss: 1.5058 - accuracy: 0.4621
Epoch 93/100
5/5 [==============================] - 0s 0s/step - loss: 1.4769 - accuracy: 0.4470
Epoch 94/100
5/5 [==============================] - 0s 0s/step - loss: 1.4504 - accuracy: 0.4242
Epoch 95/100
5/5 [==============================] - 0s 0s/step - loss: 1.4230 - accuracy: 0.4242
Epoch 96/100
5/5 [==============================] - 0s 0s/step - loss: 1.3961 - accuracy: 0.4242
Epoch 97/100
5/5 [==============================] - 0s 1ms/step - loss: 1.3708 - accuracy: 0.4318
Epoch 98/100
5/5 [==============================] - 0s 998us/step - loss: 1.3444 - accuracy: 0.4470
Epoch 99/100
5/5 [==============================] - 0s 997us/step - loss: 1.3182 - accuracy: 0.4545
Epoch 100/100
5/5 [==============================] - 0s 1ms/step - loss: 1.2951 - accuracy: 0.4545
1/1 [==============================] - 0s 213ms/step - loss: 0.8540 - accuracy: 0.6667
Epoch 1/100
5/5 [==============================] - 1s 4ms/step - loss: 1.8799 - accuracy: 0.2500
Epoch 2/100
5/5 [==============================] - 0s 4ms/step - loss: 1.6343 - accuracy: 0.3258
Epoch 3/100
5/5 [==============================] - 0s 4ms/step - loss: 1.5378 - accuracy: 0.3712
Epoch 4/100
5/5 [==============================] - 0s 0s/step - loss: 1.4903 - accuracy: 0.4242
Epoch 5/100
5/5 [==============================] - 0s 4ms/step - loss: 1.3035 - accuracy: 0.4773
Epoch 6/100
5/5 [==============================] - 0s 4ms/step - loss: 1.2276 - accuracy: 0.5379
Epoch 7/100
```

```
5/5 [==============================] - 0s 0s/step - loss: 1.1522 - accuracy: 0.5379
Epoch 8/100
5/5 [==============================] - 0s 4ms/step - loss: 1.0671 - accuracy: 0.5758
Epoch 9/100
5/5 [==============================] - 0s 0s/step - loss: 1.0572 - accuracy: 0.5379
Epoch 10/100
5/5 [==============================] - 0s 5ms/step - loss: 0.9462 - accuracy: 0.5985
Epoch 11/100
5/5 [==============================] - 0s 2ms/step - loss: 0.9056 - accuracy: 0.5758
Epoch 12/100
5/5 [==============================] - 0s 2ms/step - loss: 0.8131 - accuracy: 0.5985
Epoch 13/100
5/5 [==============================] - 0s 2ms/step - loss: 0.8191 - accuracy: 0.6136
Epoch 14/100
5/5 [==============================] - 0s 2ms/step - loss: 0.7529 - accuracy: 0.6667
Epoch 15/100
5/5 [==============================] - 0s 262us/step - loss: 0.6988 - accuracy: 0.6667
Epoch 16/100
5/5 [==============================] - 0s 4ms/step - loss: 0.6868 - accuracy: 0.6515
Epoch 17/100
5/5 [==============================] - 0s 0s/step - loss: 0.6499 - accuracy: 0.6894
Epoch 18/100
5/5 [==============================] - 0s 4ms/step - loss: 0.5962 - accuracy: 0.7197
Epoch 19/100
5/5 [==============================] - 0s 0s/step - loss: 0.5714 - accuracy: 0.7424
Epoch 20/100
5/5 [==============================] - 0s 0s/step - loss: 0.5506 - accuracy: 0.7576
Epoch 21/100
5/5 [==============================] - 0s 4ms/step - loss: 0.5127 - accuracy: 0.8258
Epoch 22/100
5/5 [==============================] - 0s 4ms/step - loss: 0.5111 - accuracy: 0.8106
Epoch 23/100
5/5 [==============================] - 0s 4ms/step - loss: 0.4799 - accuracy: 0.8333
Epoch 24/100
5/5 [==============================] - 0s 0s/step - loss: 0.4614 - accuracy: 0.8485
Epoch 25/100
5/5 [==============================] - 0s 0s/step - loss: 0.4414 - accuracy: 0.8485
Epoch 26/100
5/5 [==============================] - 0s 4ms/step - loss: 0.4581 - accuracy: 0.8409
Epoch 27/100
5/5 [==============================] - 0s 0s/step - loss: 0.4167 - accuracy: 0.8485
Epoch 28/100
5/5 [==============================] - 0s 4ms/step - loss: 0.4099 - accuracy: 0.8561
Epoch 29/100
5/5 [==============================] - 0s 4ms/step - loss: 0.4126 - accuracy: 0.8712
Epoch 30/100
5/5 [==============================] - 0s 0s/step - loss: 0.3506 - accuracy: 0.8939
Epoch 31/100
5/5 [==============================] - 0s 4ms/step - loss: 0.3735 - accuracy: 0.9015
Epoch 32/100
5/5 [==============================] - 0s 4ms/step - loss: 0.3579 - accuracy: 0.8939
Epoch 33/100
```

```
5/5 [==============================] - 0s 0s/step - loss: 0.3769 - accuracy: 0.8939
Epoch 34/100
5/5 [==============================] - 0s 4ms/step - loss: 0.3269 - accuracy: 0.9167
Epoch 35/100
5/5 [==============================] - 0s 0s/step - loss: 0.3213 - accuracy: 0.9091
Epoch 36/100
5/5 [==============================] - 0s 2ms/step - loss: 0.3181 - accuracy: 0.9091
Epoch 37/100
5/5 [==============================] - 0s 0s/step - loss: 0.3084 - accuracy: 0.9318
Epoch 38/100
5/5 [==============================] - 0s 0s/step - loss: 0.3253 - accuracy: 0.9242
Epoch 39/100
5/5 [==============================] - 0s 4ms/step - loss: 0.3335 - accuracy: 0.9167
Epoch 40/100
5/5 [==============================] - 0s 0s/step - loss: 0.2868 - accuracy: 0.9167
Epoch 41/100
5/5 [==============================] - 0s 0s/step - loss: 0.2752 - accuracy: 0.9394
Epoch 42/100
5/5 [==============================] - 0s 4ms/step - loss: 0.2774 - accuracy: 0.9470
Epoch 43/100
5/5 [==============================] - 0s 6ms/step - loss: 0.3153 - accuracy: 0.9091
Epoch 44/100
5/5 [==============================] - 0s 4ms/step - loss: 0.2885 - accuracy: 0.9167
Epoch 45/100
5/5 [==============================] - 0s 0s/step - loss: 0.3074 - accuracy: 0.9318
Epoch 46/100
5/5 [==============================] - 0s 0s/step - loss: 0.2612 - accuracy: 0.9394
Epoch 47/100
5/5 [==============================] - 0s 4ms/step - loss: 0.2433 - accuracy: 0.9394
Epoch 48/100
5/5 [==============================] - 0s 0s/step - loss: 0.2732 - accuracy: 0.9242
Epoch 49/100
5/5 [==============================] - 0s 0s/step - loss: 0.2604 - accuracy: 0.9167
Epoch 50/100
5/5 [==============================] - 0s 4ms/step - loss: 0.2491 - accuracy: 0.9318
Epoch 51/100
5/5 [==============================] - 0s 0s/step - loss: 0.2314 - accuracy: 0.9470
Epoch 52/100
5/5 [==============================] - 0s 2ms/step - loss: 0.2539 - accuracy: 0.9394
Epoch 53/100
5/5 [==============================] - 0s 0s/step - loss: 0.2223 - accuracy: 0.9242
Epoch 54/100
5/5 [==============================] - 0s 4ms/step - loss: 0.2067 - accuracy: 0.9545
Epoch 55/100
5/5 [==============================] - 0s 0s/step - loss: 0.2257 - accuracy: 0.9394
Epoch 56/100
5/5 [==============================] - 0s 0s/step - loss: 0.1948 - accuracy: 0.9697
Epoch 57/100
5/5 [==============================] - 0s 5ms/step - loss: 0.2136 - accuracy: 0.9545
Epoch 58/100
5/5 [==============================] - 0s 2ms/step - loss: 0.2149 - accuracy: 0.9621
Epoch 59/100
```

```
5/5 [==============================] - 0s 2ms/step - loss: 0.1907 - accuracy: 0.9621
Epoch 60/100
5/5 [==============================] - 0s 2ms/step - loss: 0.1968 - accuracy: 0.9545
Epoch 61/100
5/5 [==============================] - 0s 2ms/step - loss: 0.1941 - accuracy: 0.9394
Epoch 62/100
5/5 [==============================] - 0s 0s/step - loss: 0.2294 - accuracy: 0.9318
Epoch 63/100
5/5 [==============================] - 0s 4ms/step - loss: 0.2045 - accuracy: 0.9242
Epoch 64/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1784 - accuracy: 0.9545
Epoch 65/100
5/5 [==============================] - 0s 0s/step - loss: 0.1902 - accuracy: 0.9470
Epoch 66/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1882 - accuracy: 0.9470
Epoch 67/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1751 - accuracy: 0.9621
Epoch 68/100
5/5 [==============================] - 0s 0s/step - loss: 0.1750 - accuracy: 0.9394
Epoch 69/100
5/5 [==============================] - 0s 0s/step - loss: 0.1843 - accuracy: 0.9470
Epoch 70/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1715 - accuracy: 0.9545
Epoch 71/100
5/5 [==============================] - 0s 2ms/step - loss: 0.2200 - accuracy: 0.9242
Epoch 72/100
5/5 [==============================] - 0s 0s/step - loss: 0.1641 - accuracy: 0.9697
Epoch 73/100
5/5 [==============================] - 0s 0s/step - loss: 0.1988 - accuracy: 0.9470
Epoch 74/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1971 - accuracy: 0.9470
Epoch 75/100
5/5 [==============================] - 0s 0s/step - loss: 0.1560 - accuracy: 0.9621
Epoch 76/100
5/5 [==============================] - 0s 0s/step - loss: 0.1892 - accuracy: 0.9470
Epoch 77/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1928 - accuracy: 0.9545
Epoch 78/100
5/5 [==============================] - 0s 0s/step - loss: 0.1630 - accuracy: 0.9697
Epoch 79/100
5/5 [==============================] - 0s 0s/step - loss: 0.1838 - accuracy: 0.9545
Epoch 80/100
5/5 [==============================] - 0s 6ms/step - loss: 0.1537 - accuracy: 0.9621
Epoch 81/100
5/5 [==============================] - 0s 0s/step - loss: 0.1483 - accuracy: 0.9697
Epoch 82/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1469 - accuracy: 0.9621
Epoch 83/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1927 - accuracy: 0.9470
Epoch 84/100
5/5 [==============================] - 0s 0s/step - loss: 0.1749 - accuracy: 0.9545
Epoch 85/100
```

```
5/5 [==============================] - 0s 0s/step - loss: 0.1645 - accuracy: 0.9545
Epoch 86/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1411 - accuracy: 0.9545
Epoch 87/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1450 - accuracy: 0.9773
Epoch 88/100
5/5 [==============================] - 0s 0s/step - loss: 0.1746 - accuracy: 0.9545
Epoch 89/100
5/5 [==============================] - 0s 2ms/step - loss: 0.1279 - accuracy: 0.9697
Epoch 90/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1316 - accuracy: 0.9848
Epoch 91/100
5/5 [==============================] - 0s 0s/step - loss: 0.1670 - accuracy: 0.9470
Epoch 92/100
5/5 [==============================] - 0s 0s/step - loss: 0.1805 - accuracy: 0.9318
Epoch 93/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1345 - accuracy: 0.9621
Epoch 94/100
5/5 [==============================] - 0s 0s/step - loss: 0.1790 - accuracy: 0.9545
Epoch 95/100
5/5 [==============================] - 0s 0s/step - loss: 0.1490 - accuracy: 0.9621
Epoch 96/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1435 - accuracy: 0.9394
Epoch 97/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1634 - accuracy: 0.9470
Epoch 98/100
5/5 [==============================] - 0s 0s/step - loss: 0.1682 - accuracy: 0.9470
Epoch 99/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1984 - accuracy: 0.9167
Epoch 100/100
5/5 [==============================] - 0s 4ms/step - loss: 0.1511 - accuracy: 0.9470
1/1 [==============================] - 0s 250ms/step - loss: 0.2106 - accuracy: 0.9333

Naive Accuracy:  66.666%

Naive Predictions:
 [[0.01354977 0.5061691  0.48028105]
 [0.06246578 0.56894875 0.36858547]
 [0.0074184  0.44270876 0.5498728 ]
 [0.01080847 0.4541977  0.53499377]
 [0.02312316 0.55372    0.4231569 ]
 [0.00527738 0.39130107 0.60342157]
 [0.00576249 0.52103436 0.4732032 ]
 [0.00304265 0.31141222 0.68554515]
 [0.07634713 0.58648854 0.33716434]
 [0.00298612 0.3317028  0.6653111 ]
 [0.00340858 0.28069374 0.7158977 ]
 [0.01619304 0.5350266  0.44878045]
 [0.00580444 0.32919884 0.66499674]
 [0.01750648 0.56174    0.4207535 ]
 [0.00643218 0.49766505 0.49590284]]
```

```
Better Accuracy:  93.333%

Better Predictions:
 [[0.00862633 0.9620326  0.02934105]
 [0.9790445  0.01505063 0.00590489]
 [0.00578253 0.9523178  0.0418997 ]
 [0.00888783 0.97394866 0.01716346]
 [0.07783689 0.7403439  0.18181916]
 [0.01994696 0.21152395 0.7685291 ]
 [0.0148526  0.86186147 0.12328597]
 [0.00590803 0.02657974 0.96751225]
 [0.9846364  0.0105518  0.00481179]
 [0.00601991 0.05104782 0.94293225]
 [0.00143864 0.0452151  0.95334625]
 [0.01584265 0.9249697  0.0591877 ]
 [0.01459309 0.3523831  0.63302374]
 [0.01764467 0.9287313  0.05362389]
 [0.01701402 0.77023745 0.21274848]]
```

As seen in the example run, the multi-layer perceptron performed much better than the single-layer perceptron. The accuracy for the single-layer perceptron and multi-layer perceptron was 66.666% and 93.333%, respectively.

Thus, it can easily be stated that the hidden layers and preprocessing of data allowed the multi-layer perceptron to much better learn the data.

The 2 hidden layers allowed the multi-layer perceptron to learn higher abstract features about the Iris data set and thus produced a higher accuracy. Furthermore, the data scaling and batch normalization increased the network's learning rate.

Although a 93.333% accuracy is astounding, this is not likely to continue if the test data set size increased.

6 Conclusions

Overall, the results gathered follow the expectation/hypothesis, that is, that a more complex network, but not too complex, would be able to learn much more effectively than a simple one. This is likely in many scenarios.
However, it is likely that the bias-variance tradeoff effect would come into play here. That is, as the bias of a model increases, its variance will as well, and vise-versa. A simple model has extreme bias, but no variance.
A extremely complex model has high variance, but almost no bias. Thus, an extremely complex model is likely to overfit a data set, while a simple model is likely to underfit a data set.
Therefore, it is evident that caution must be taken to find a point where the bias and variance are minimized while still producing an effective model.
This can be done by using a technique like cross-validation to produce models that are each trained and validated against different "folds" of the train data set. Then, the model that minimizes the loss function (against the sum of error of the validation folds) can be used as the final model and can be run on the test set to produce the resulting accuracy.

7 Platforms

- pycharm, python 3.9
- overleaf.com