Assignment 3

October 30, 2021

1 Decision Tree and Random Forest

1.1 Reading data

[166]:
```
train_raw = pd.read_csv(train_path, low_memory=False, sep=';')
valid_raw = pd.read_csv(valid_path, low_memory=False, sep=';')
test_raw = pd.read_csv(test_path, low_memory=False, sep=';')
```

Here converting variables of type string into categorical variable.

[218]:
```
train_category(train_raw)
apply_category(test_raw, train_raw)
apply_category(valid_raw, train_raw)
```

1.1.1 Visualizing the data

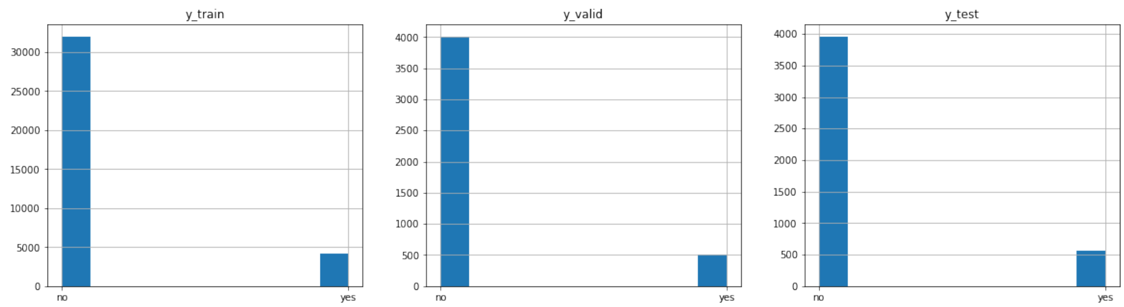
There we are dealing with tabular dataset.

[168]:
```
   age         job  marital  education default  balance housing loan  \
0   57  unemployed  married  secondary      no      890      no   no
1   56  technician  married  secondary      no     2558      no   no
2   50  technician  married   tertiary      no      267     yes   no
3   47  management  married    unknown      no     4567      no   no
4   49  management  married   tertiary      no     5887      no   no

     contact  day month  duration  campaign  pdays  previous poutcome    y
0   cellular    5   feb       343         4     -1         0  unknown   no
1    unknown   19   jun       288         1     -1         0  unknown   no
2   cellular   21   nov        30         1     -1         0  unknown   no
3  telephone   31   jul       921         4     -1         0  unknown   no
4   cellular    2   jun       181         3    293         2  failure  yes
```

Looking at the dependent variable we can see the target classes are very skewed.

[14]: <AxesSubplot:title={'center':'y_test'}>



1.2 Decision Tree

This is the class that implements the decision tree, the code below recursively grows the number of nodes in the decision tree

[ ]:
```
class DecisionTree():

    def __init__(self, x, y, is_cat_col=None, idxs=None, min_leaf=4,
                 depth=0, max_depth=None):
        self.x, self.y, self.idxs, self.is_cat_col, self.min_leaf = x, y, idxs, is_cat_col, min_leaf
        if idxs is None: self.idxs = np.arange(len(y))

        self.n, self.c = len(self.idxs), x.shape[1]
        self.score = float('inf')
        self.children = []
        self.depth = depth
        self.max_depth = max_depth

        labels, label_count = np.unique(self.y[self.idxs], return_counts=True)
        self.val = labels[np.argmax(label_count)]

        self.split_var, self.split_val, self.split_col = None, None, None
        self.find_varsplit()


    def find_varsplit(self):
        if self.max_depth is not None and self.depth >= self.max_depth:
            return

        for i in range(self.c):
            if self.is_cat_col is not None and self.is_cat_col[i]:
                self.find_split_category(i)
            else:
                self.find_split_numerical(i)
```

```
        if self.is_leaf:
            return

        x = self.split_col
        if self.is_cat_col is not None and self.is_cat_col[self.split_var]:
            for attr in sorted(self.split_val):
                attr_mask = np.nonzero(x == attr)[0]
                self.children.append(DecisionTree(self.x, self.y, self.
↪is_cat_col,
                                                  self.idxs[attr_mask], self.
↪min_leaf,
                                                  max_depth=self.max_depth,
                                                  depth=self.depth+1))
        else:
            lr_masks = []
            lr_masks.append( np.nonzero(x <= self.split_val)[0] )
            lr_masks.append( np.nonzero(x > self.split_val)[0] )
            for mask in lr_masks:
                self.children.append(DecisionTree(self.x, self.y, self.
↪is_cat_col,
                                                  self.idxs[mask], self.
↪min_leaf,
                                                  max_depth=self.max_depth,
                                                  depth=self.depth+1))


    def find_split_category(self, var_idx):
        x, y = self.x[self.idxs, var_idx], self.y[self.idxs]
        if len(y) < self.min_leaf or len(np.unique(y)) == 1:
            return

        idx_sort = np.argsort(x)
        sorted_x = x[idx_sort]
        attrs, idx_start, attrs_cnt = np.unique(sorted_x, return_index=True,␣
↪return_counts=True)
        attrs_idx = np.split(idx_sort, idx_start[1:])

        if len(attrs) == 1:
            return

        curr_score = 0
        attrs_dict = {}
        for i in range(len(attrs)):
            attr_prob = attrs_cnt[i]/self.n
            attr_entropy = entropy(y[attrs_idx[i]])
            curr_score += attr_prob*attr_entropy
```

```
            attrs_dict[attrs[i]] = i

        if curr_score < self.score:
            self.score, self.split_var = curr_score, var_idx
            self.split_val = attrs_dict
            self.split_col = x


    def find_split_numerical(self, var_idx):
        x, y = self.x[self.idxs, var_idx], self.y[self.idxs]
        if len(y) < self.min_leaf or len(np.unique(y)) == 1:
            return

        median = np.median(x)

        lr_mask = []
        lr_mask.append(x <= median)
        lr_mask.append(x > median)

        curr_score = 0
        for mask in lr_mask:
            count = mask.sum()
            if count < self.min_leaf:
                return
            entp = entropy(y[mask])
            prob = count/self.n
            curr_score += prob*entp

        if curr_score < self.score:
            self.score, self.split_var = curr_score, var_idx
            self.split_val = median
            self.split_col = x
```

1.2.1 Attribute value

Converting data into integers

[236]:
```
X_train, y_train, is_cat_col = proc_df(train_raw, y_label='y')
X_valid, y_valid, _ = proc_df(valid_raw, y_label='y')
X_test, y_test, _ = proc_df(test_raw, y_label='y')
```

Size of the dataset

[237]:

	train	valid	test
rows	36168	4522	4521
columns	16	16	16

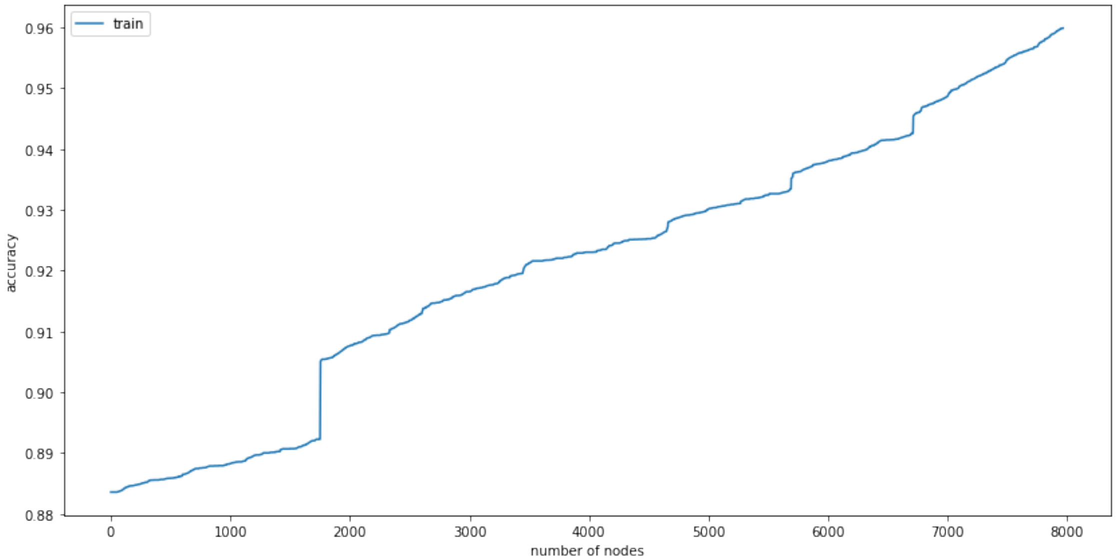
Training the decision tree

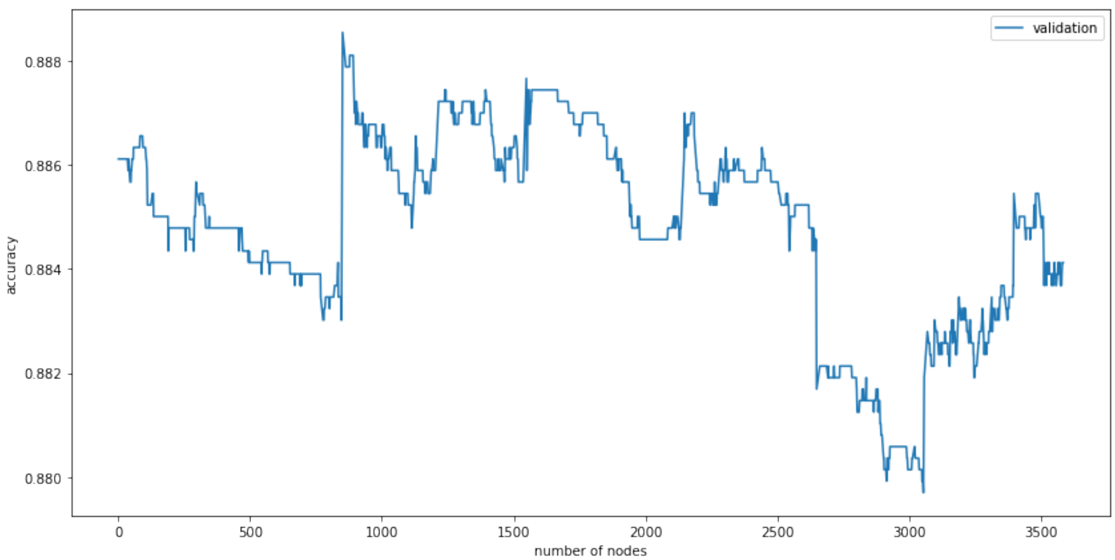
[238]:
```
tree = DecisionTree(X_train, y_train, is_cat_col)
```

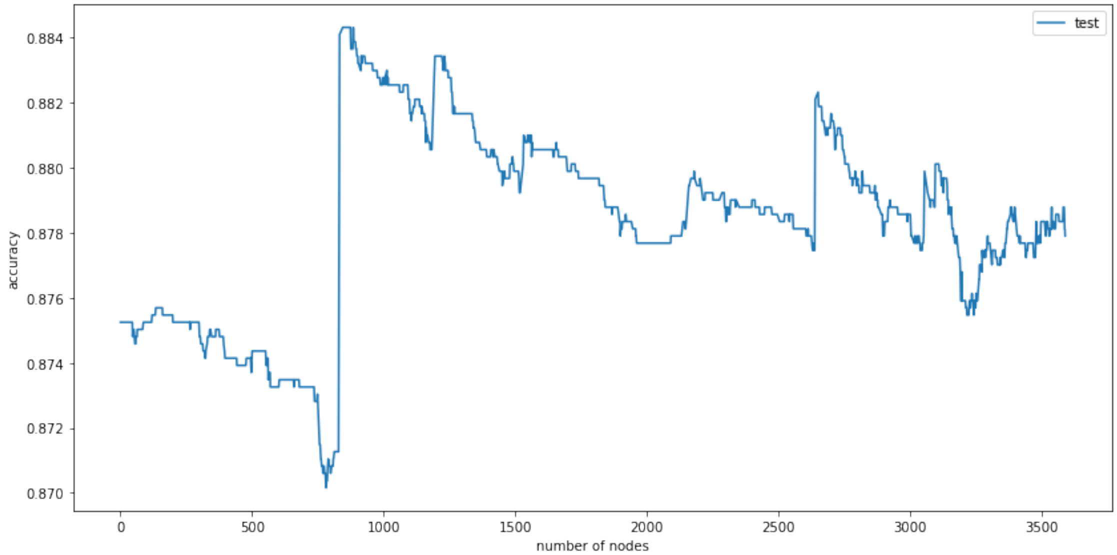
[239]:

	train	validation	test
accuracy	0.959909	0.883459	0.878567

Plots of accuracy as we increase the number of nodes in the tree







Conclusion As we can see from the above graphs,

- for training data, as the nodes increases accuracy increases
- whereas for validation and test data, as the node there is a trend of decrease in the accuracy

1.2.2 One hot encoding

Converting categorical variable into its one hot representation

[228]:
```
X_train, y_train, is_cat_col = proc_df(train_raw, y_label='y', min_n_ord=100)
X_valid, y_valid, _ = proc_df(valid_raw, y_label='y', min_n_ord=100)
X_test, y_test, _ = proc_df(test_raw, y_label='y', min_n_ord=100)
```

Size of the dataset

[229]:

	train	valid	test
rows	36168	4522	4521
columns	51	51	51

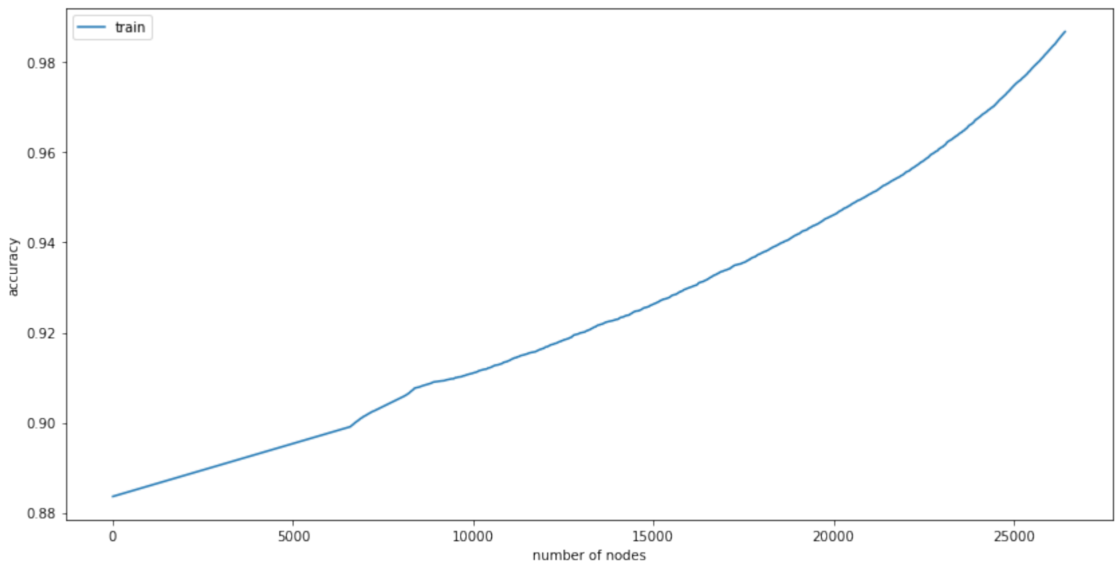
Training decision tree

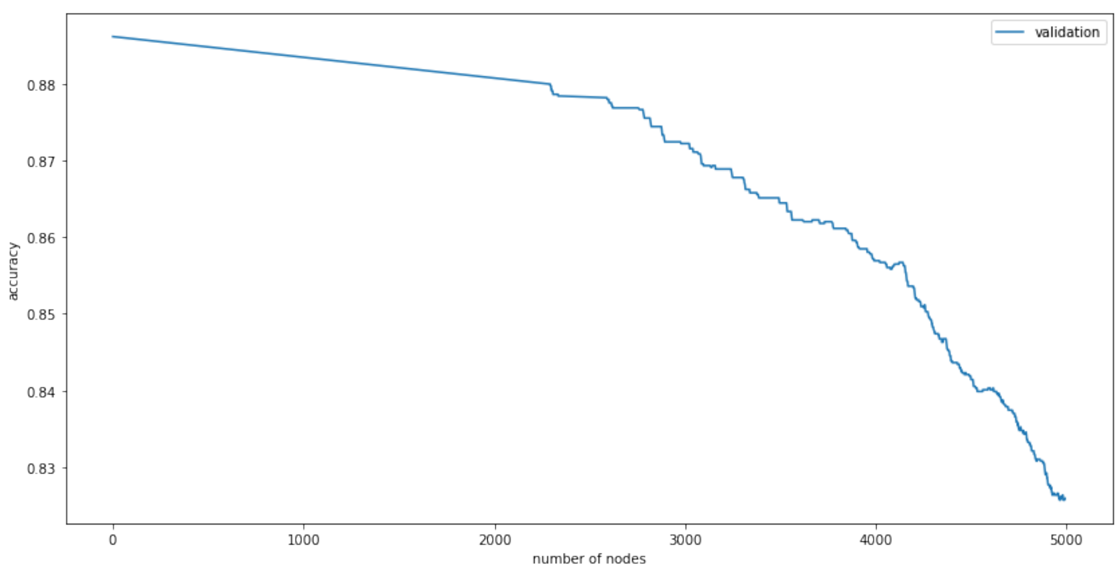
[230]:
```
tree = DecisionTree(X_train, y_train, is_cat_col)
```

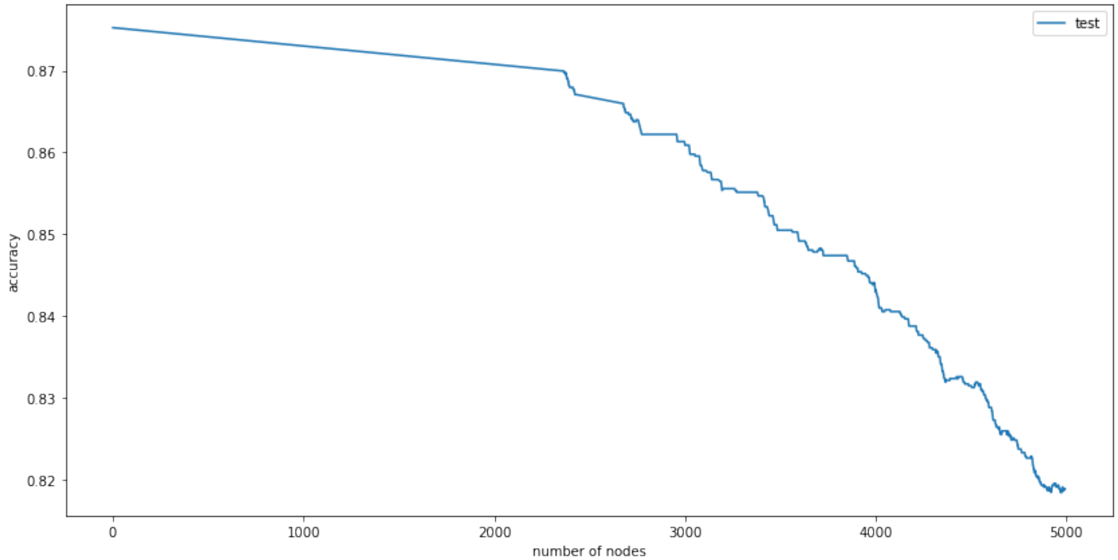
[231]:

	train	validation	test
accuracy	0.986812	0.822645	0.814864

Plots of accuracy as we increase the number of nodes in the tree


train
0.98
0.96
0.94
0.92
0.90
0.88
accuracy
0
5000
10000
15000
20000
25000
number of nodes


validation
0.88
0.87
0.86
0.85
0.84
0.83
accuracy
0
1000
2000
3000
4000
5000
number of nodes



Conclusion As we can see from the above graphs,

- for training data, as the nodes increases accuracy increases
- for validation and test data, as the number of nodes increases the accuracy decreases.

Here is model is overfitting the data

[245]:

	train	validation	test
attribute	0.959909	0.882574	0.876134
onehot	0.986812	0.814463	0.808007

As we can see above by using the one-hot encoding for categorical variables attributes the model has overfit the data.

1.3 Pruning

Converting data into integers

[201]:
```
X_train, y_train, is_cat_col = proc_df(train_raw, y_label='y')
X_valid, y_valid, _ = proc_df(valid_raw, y_label='y')
X_test, y_test, _ = proc_df(test_raw, y_label='y')
```

Training the decision tree

[205]:
```
tree = DecisionTree(X_train, y_train, is_cat_col)
```

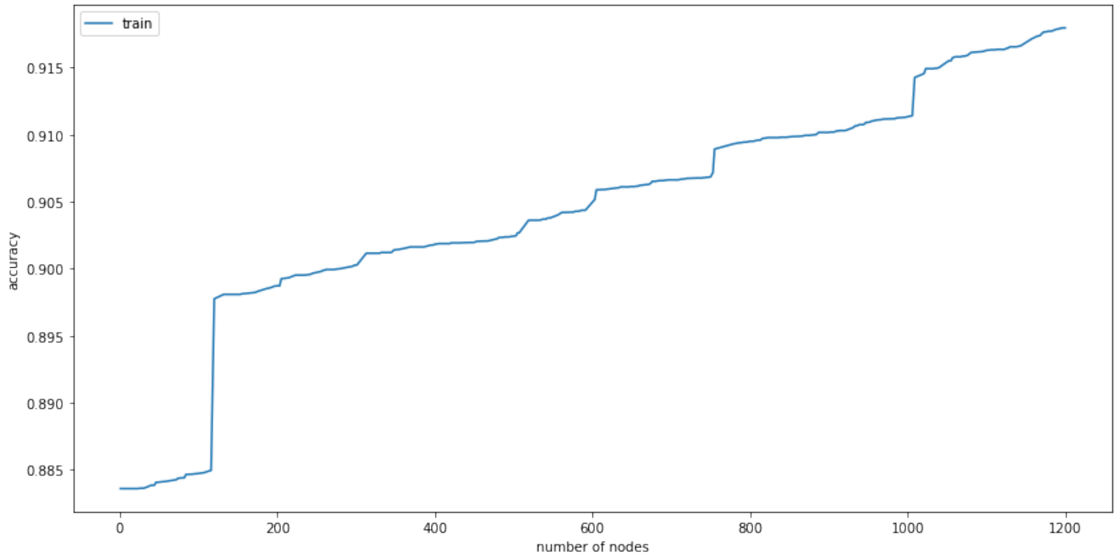
Pruning the decision tree

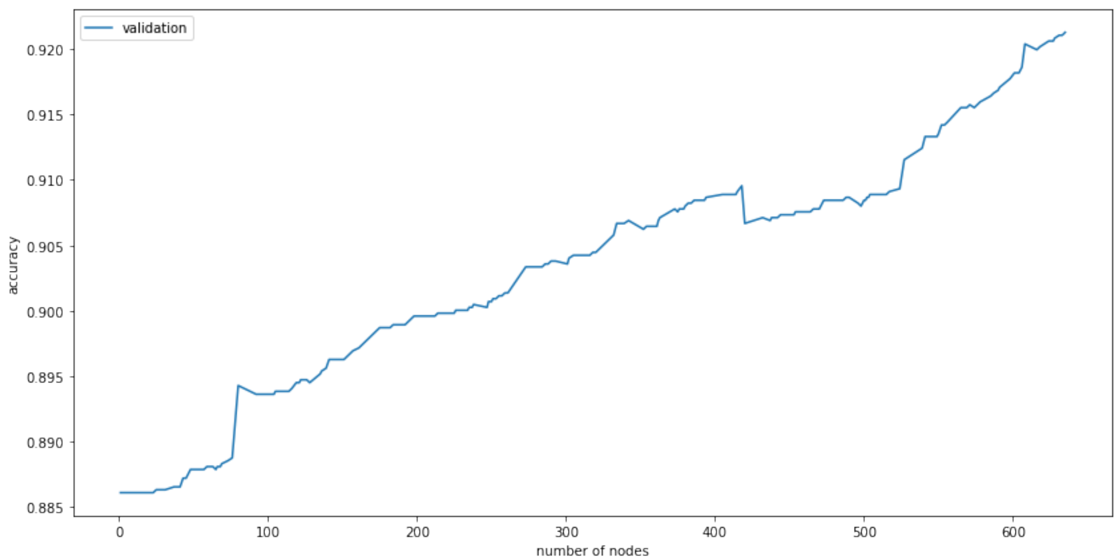
[207]:
```
pruned_tree = tree.post_pruning(X_valid, y_valid)
```

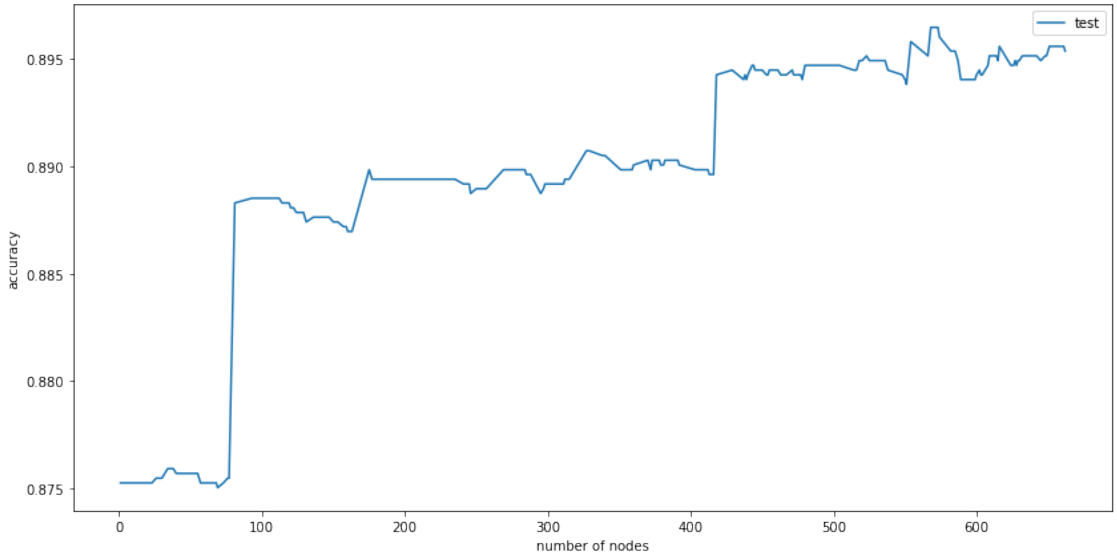
[209]:

	train	validation	test
Tree	0.959909	0.882795	0.876355
Pruned tree	0.917966	0.920831	0.896483

As we can see here after pruning the tree the resultant model has generalized well to the validation and the test set.







1.3.1 Conclusion

As we can see from the above graphs that there is a clear trend in training, validation and test data that, as the nodes increases accuracy increases.

Comparing the pruned tree with the one without pruning we can see that pruned tree has generalized better.

1.4 Random Forest

To avoid treating categorical variable as oridinal categorical variable, I have converted them to one hot encoding. Below is the code for the same.

[ ]:
```
def convert_numerical(df, min_n_ord=0):
    for n, c in df.items():
        if is_categorical_dtype(c) and len(df[n].cat.categories) > min_n_ord:
            df[n] = c.cat.codes

def proc_df(df, y_label, min_n_ord=0):
    X, y = None, None
    df = df.copy()

    y = df[y_label]
    if is_categorical_dtype(y): y = (y.cat.codes).values
    df.drop(y_label, axis=1, inplace=True)

    convert_numerical(df, min_n_ord)
    df = pd.get_dummies(df)
    X = df.values
```

```
    return X, y
```

Here I have converted a categoical variable with less than `min_n_ord` attributes to its one hot encoding with `pd.get_dummies(df)`.

[215]:
```
X_train, y_train, is_cat_col = proc_df(train_raw, y_label='y', min_n_ord=6)
X_valid, y_valid, _ = proc_df(valid_raw, y_label='y', min_n_ord=6)
X_test, y_test, _ = proc_df(test_raw, y_label='y', min_n_ord=6)
```

Used sklearn to create a Random Forest model

[216]:
```
model = RandomForestClassifier(n_jobs=-1, n_estimators=40,max_features=0.5,
                      min_samples_leaf=5,
                      oob_score=True).fit(X_train, y_train)
```

Accuracy with the default parameters

[217]:

	Train	Validation	Test	OOB
0	0.9572	0.903804	0.901349	0.904418

1.4.1 Optimal parameters using Grid search

Code for grid search through parameters

[29]:
```
accuracies = {}

n_estimators = np.arange(50, 451, 50)
max_features = np.arange(0.1, 1, 0.1)
min_samples_split = np.arange(2, 10, 2)

best_model, best_score, best_param = None, None, None
for ne in n_estimators:
    for mf in max_features:
        for mss in min_samples_split:
            model = RandomForestClassifier(n_jobs=-1, n_estimators=ne,
                                           max_features=mf,
                                           min_samples_leaf=5,
                                           min_samples_split=mss,
                                           oob_score=True,
                                           random_state=125).fit(X_train,
                                                                 y_train)
            accuracies[(ne, mf, mss)] = (model.score(X_valid, y_valid),
                                         model.oob_score_)

            if best_score is None or model.oob_score_ > best_score:
                best_score = model.oob_score_
                best_param = (ne, mf, mss)
                best_model = model
```

The following are the best parameters obtained

[30]:

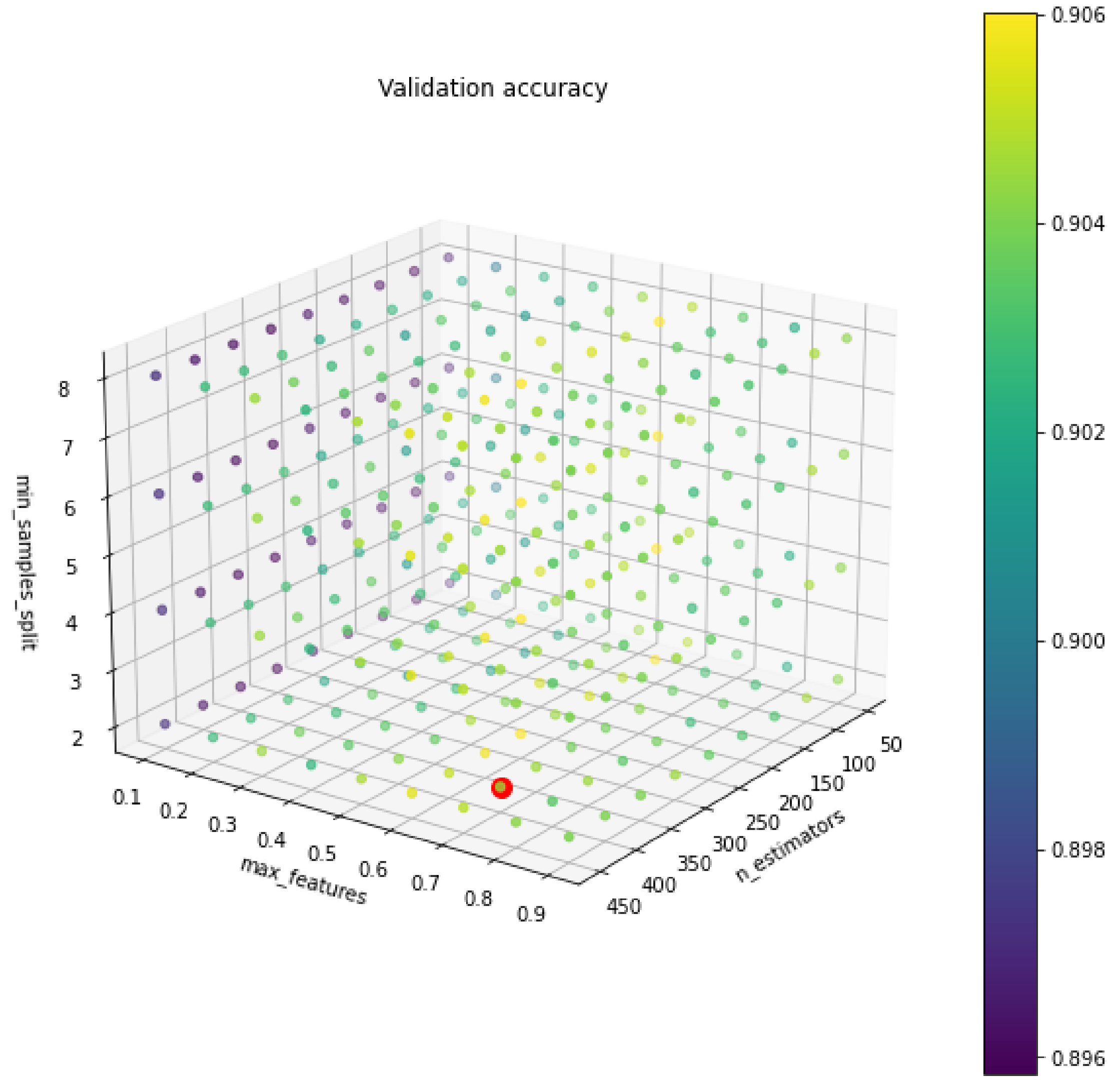
	n_estimators	max_features	min_samples_split
0	400	0.7	2

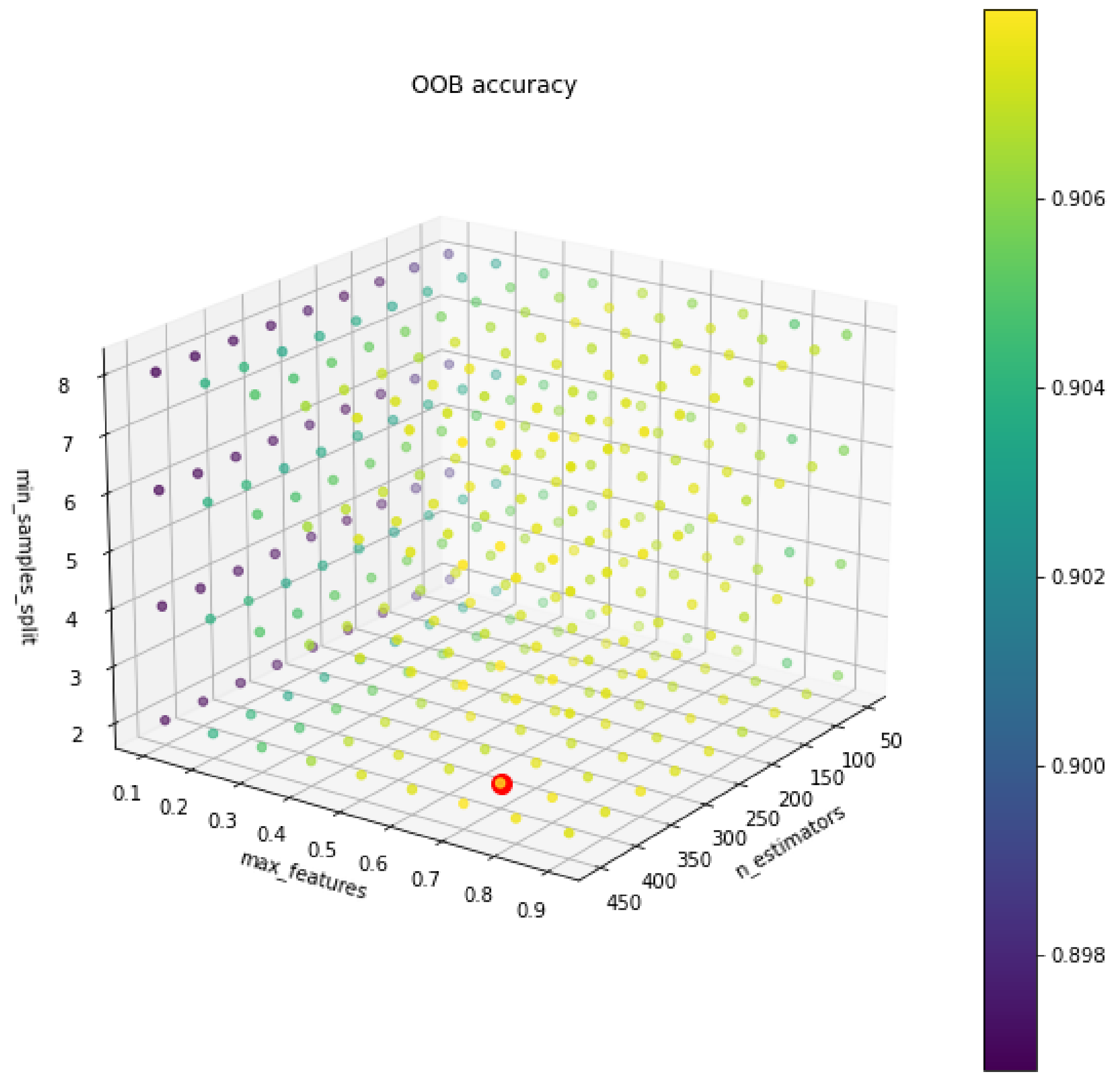
Accuracy with the best parameters,

[31]:

	Train	Validation	Test	OOB
0	0.9632	0.904025	0.903119	0.907432

Viewing the error in 3D space for different parameter values





1.4.2 Comparision

[248]:

	Train	Validation	Test	OOB
Decision Tree	0.917966	0.920831	0.896483	NaN
Random Forest	0.963200	0.904025	0.903119	0.907432

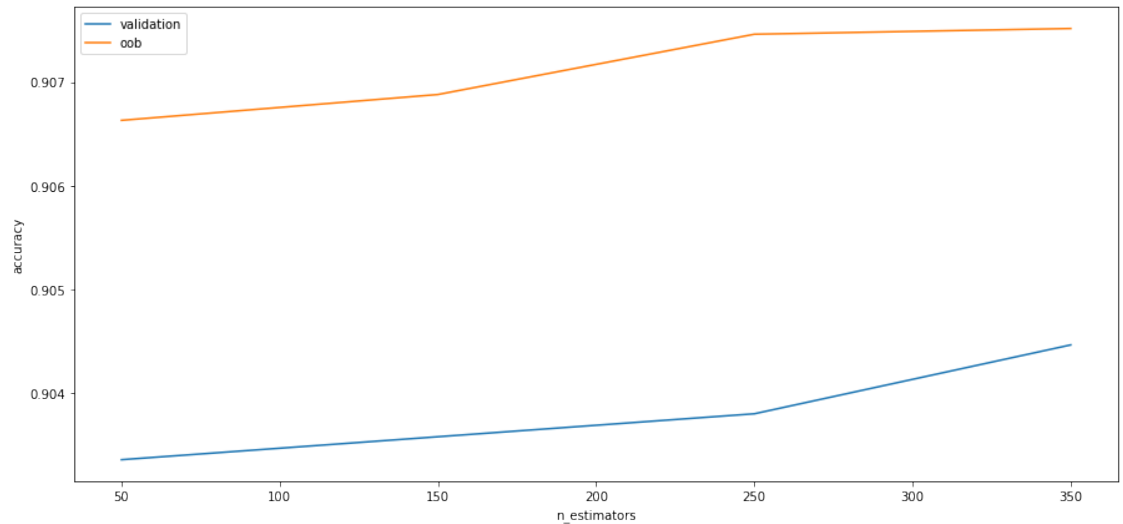
As we can see the numbers are very close, `random forest` is performing better on test and train data, whereas `Decision tree` on the validation set.

1.5 Random forest parameter sensitivity

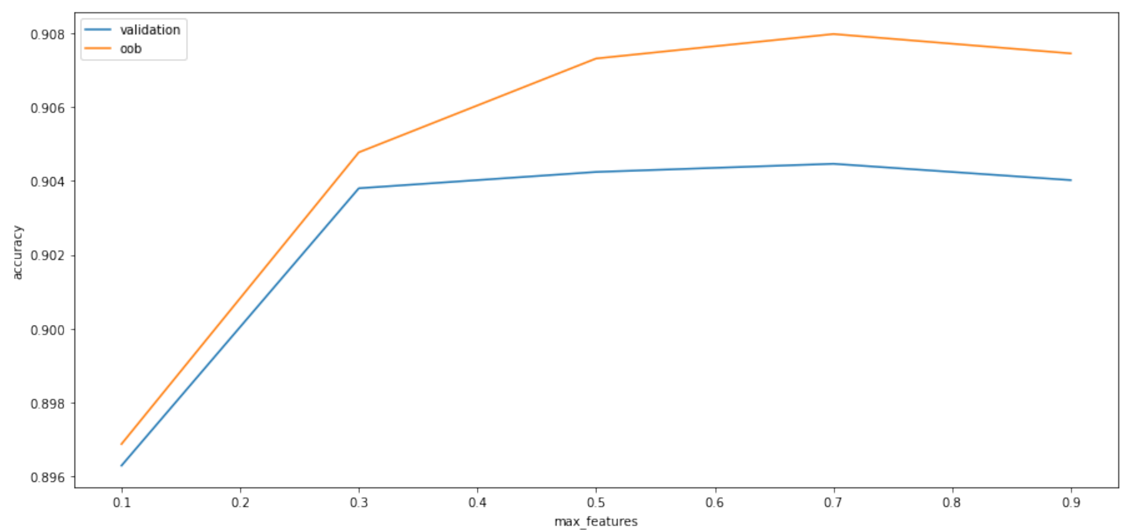
Train the model on the optimal parameters

[91]:
```python
model = RandomForestClassifier(n_jobs=-1, n_estimators=400,
                                          max_features=0.7,
                                          min_samples_leaf=5,
                                          min_samples_split=2,
                                          oob_score=True,
                                          random_state=125).fit(X_train,
                                                                 y_train)
```

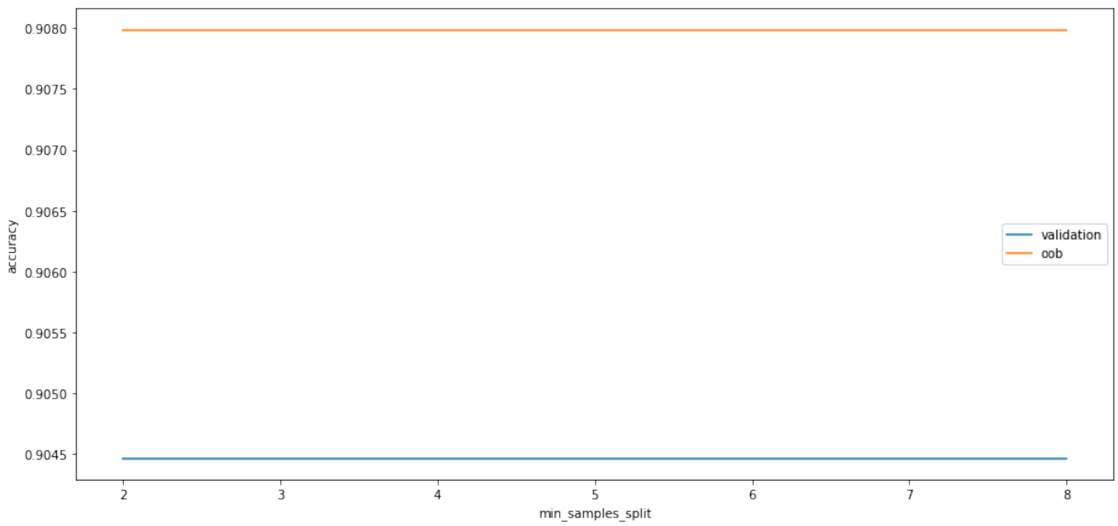
Plot of the validation and OOB accuracy



As we can see that as we increase the number of estimator (numer of tree) the accuracy increases.



As we increase the max features the accuracy increases



The min sample split parameter is not affecting the accuracy much

1.5.1 Conclusion

We can see that as we increase the n_estimators and max_features the accuracy increases, and min_samples_split has got not much affect.

And out of n_estimators and max_features, max_features is more sensitive of accuracy.

2 Neural Networks

2.1 One Hot Encoding

2.1.1 Reading data

Reading csv file

[4]:
```
train_raw = pd.read_csv(train_path, low_memory=False, header=None)
test_raw = pd.read_csv(test_path, low_memory=False, header=None)
```

converting into categorical variable

[5]:
```
convert_category(train_raw)
apply_category(test_raw, train_raw)
```

2.1.2 Visualizing the data

Again we will be working with tabular data

[6]:

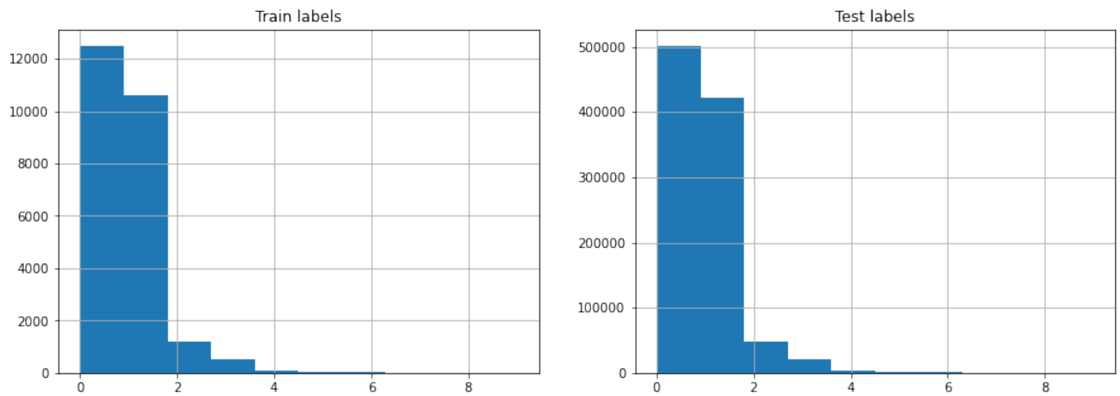
	0	1	2	3	4	5	6	7	8	9	10
0	1	10	1	11	1	13	1	12	1	1	9
1	2	11	2	13	2	10	2	12	2	1	9

2	3	12	3	11	3	13	3	10	3	1	9
3	4	10	4	11	4	1	4	13	4	12	9
4	4	1	4	13	4	12	4	11	4	10	9

Label distribution

[7]:

```
Text(0.5, 1.0, 'Test labels')
```



The class distributions are very skewed

2.1.3 Encoding

[8]:

```python
X_train, y_train, _ = proc_df(train_raw, 10, min_n_ord=100)
X_test, y_test, _ = proc_df(test_raw, 10, min_n_ord=100)
```

Size of the dataset

[9]:

	train	test
rows	25010	1000000
columns	85	85

saving data

[10]:

```python
save_path = f'{data_dir}/q2_data.pickle'

with open(save_path, 'wb') as file:
    pickle.dump((X_train, X_test, y_train, y_test), file)
```

2.2 Implementation

Below is the code for implementation of the Linear layer

[ ]:

```python
class Linear():
    @staticmethod
    def forward(x, w, b):
        num_train = x.shape[0]
```

```
        z = x.reshape(num_train, -1)@w + b
        cache = (x, w, b)
        return z, cache

    @staticmethod
    def backward(dz, cache):
        x, w, b = cache
        num_train = x.shape[0]
        dw = x.reshape(num_train, -1).T@dz
        dx = (dz@w.T).reshape(x.shape)
        db = dz.sum(axis=0)
        return dx, dw, db
```

Class for Sigmoid layer

[ ]:
```
class Sigmoid():

    @staticmethod
    def forward(x):
        a = 1 + np.exp(-x)
        a = np.reciprocal(a)
        cache = a
        return a, cache

    @staticmethod
    def backward(da, cache):
        a = cache
        dx = da*a*(1 - a)
        return dx
```

Class for ReLU layer

[ ]:
```
class ReLU():

    @staticmethod
    def forward(x):
        a = np.maximum(x, 0)
        cache = x
        return a, cache

    @staticmethod
    def backward(da, cache):
        x = cache
        dx = np.where( x > 0, da, 0)
        return dx
```

Loss function used

[ ]:
```
def squared_loss(scores, y):
    num_train = len(y)
    diff = scores.copy()
    diff[range(num_train), y] -= 1

    loss = np.sum(diff**2)
    loss /= 2*num_train

    dscores = diff/num_train
    return loss, dscores
```

These function are some of the basic building blocks of neural network, combination of these will generate the network.

2.3 Validation data creation

[3]:
```
data_dir = "./data/"

q2_data = f'{data_dir}/q2_data.pickle'

with open(q2_data, 'rb') as file:
    X_train, X_test, y_train, y_test = pickle.load(file)
```

Creating validation set

[4]:
```
data_dict = split_skewed(X_train, y_train)

data_dict['X_test'] = X_test
data_dict['y_test'] = y_test
```

Dataset size

[5]:

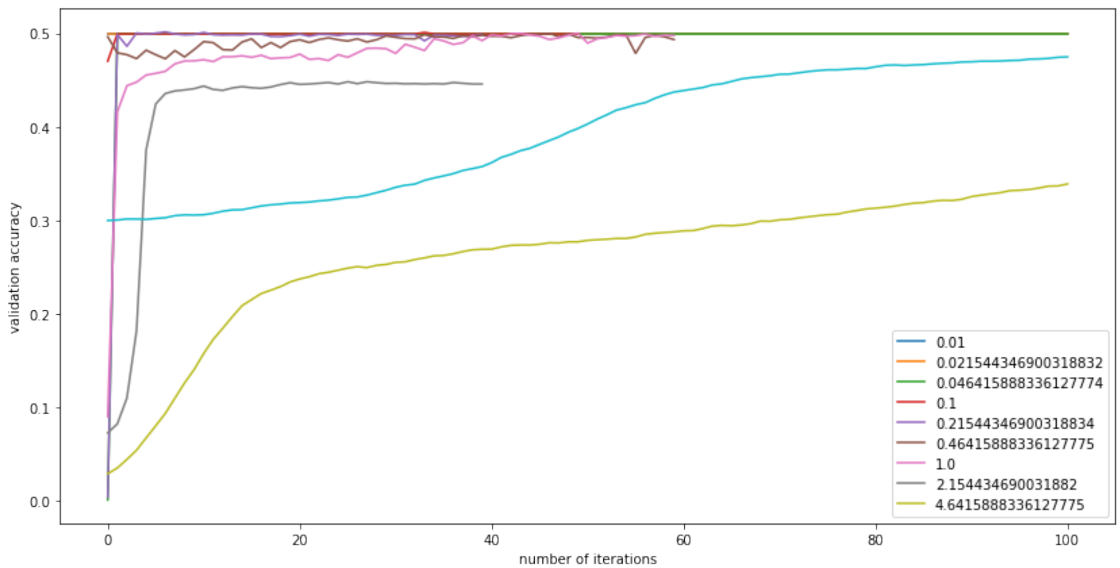
	Train	Validation	Test
X	(20004, 85)	(5006, 85)	(1000000, 85)
y	20004	5006	1000000

2.4 Single hidden layer

Stopping criteria: The stopping criteria I have used is a combination of `number of epochs` and `early stopping`. In early stopping, I stop the training process if average validation accuracy over certain `epoch_per_stop` epochs decrease, `epoch_per_stop`=10.

2.4.1 Search Weight Scale

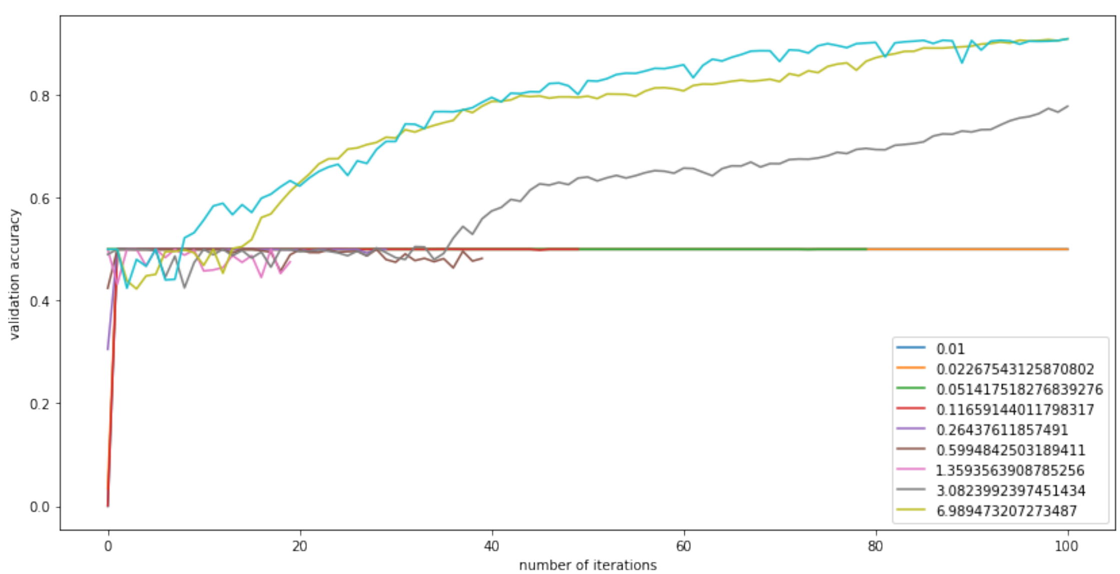
Here I have tried to find the hyperparameter, `weight scale`. Below I have plotted validation accuraccy of different weight scales over the training process.



We can see here that different weight scales are all converging to 0.5 accuracy.

2.4.2 Learning rate

Here I have tried to find the hyperparameter, `learning rate`. Below I have plotted validation accuraccy of different learning rate over the training proces.



from the above plot we can see the with learning rate a between 6-15 is giving us good results.

2.4.3 Hidden nodes

Now observing the behaviour for the number of hidden nodes.

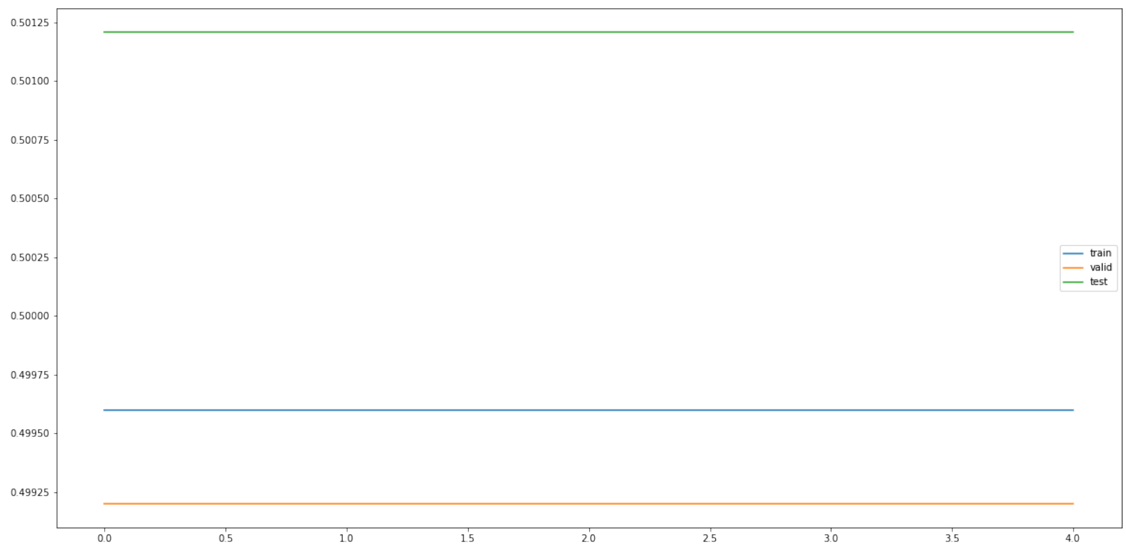
learning rate 0.1 **Accuracy and train time** for various hidden layer size

[92]:

	5	10	15	20	25
Train	0.499600	0.499600	0.499600	0.499600	0.499600
Validation	0.499201	0.499201	0.499201	0.499201	0.499201
Test	0.501209	0.501209	0.501209	0.501209	0.501209
Time	3.263922	3.361467	3.599032	3.729842	3.821096

Plot of the above table

[93]: <matplotlib.legend.Legend at 0x7fad1be709d0>



confusion matrix for each parameter

Hidden size : 5

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	501209	0	0	0	0	0	0	0	0	0
1	422498	0	0	0	0	0	0	0	0	0
2	47622	0	0	0	0	0	0	0	0	0
3	21121	0	0	0	0	0	0	0	0	0
4	3885	0	0	0	0	0	0	0	0	0
5	1996	0	0	0	0	0	0	0	0	0
6	1424	0	0	0	0	0	0	0	0	0
7	230	0	0	0	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

Hidden size : 10

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	501209	0	0	0	0	0	0	0	0	0
1	422498	0	0	0	0	0	0	0	0	0
2	47622	0	0	0	0	0	0	0	0	0
3	21121	0	0	0	0	0	0	0	0	0
4	3885	0	0	0	0	0	0	0	0	0
5	1996	0	0	0	0	0	0	0	0	0
6	1424	0	0	0	0	0	0	0	0	0
7	230	0	0	0	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

Hidden size : 15

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	501209	0	0	0	0	0	0	0	0	0
1	422498	0	0	0	0	0	0	0	0	0
2	47622	0	0	0	0	0	0	0	0	0
3	21121	0	0	0	0	0	0	0	0	0
4	3885	0	0	0	0	0	0	0	0	0
5	1996	0	0	0	0	0	0	0	0	0
6	1424	0	0	0	0	0	0	0	0	0
7	230	0	0	0	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

Hidden size : 20

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	501209	0	0	0	0	0	0	0	0	0
1	422498	0	0	0	0	0	0	0	0	0
2	47622	0	0	0	0	0	0	0	0	0
3	21121	0	0	0	0	0	0	0	0	0
4	3885	0	0	0	0	0	0	0	0	0
5	1996	0	0	0	0	0	0	0	0	0
6	1424	0	0	0	0	0	0	0	0	0
7	230	0	0	0	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

Hidden size : 25

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	501209	0	0	0	0	0	0	0	0	0
1	422498	0	0	0	0	0	0	0	0	0

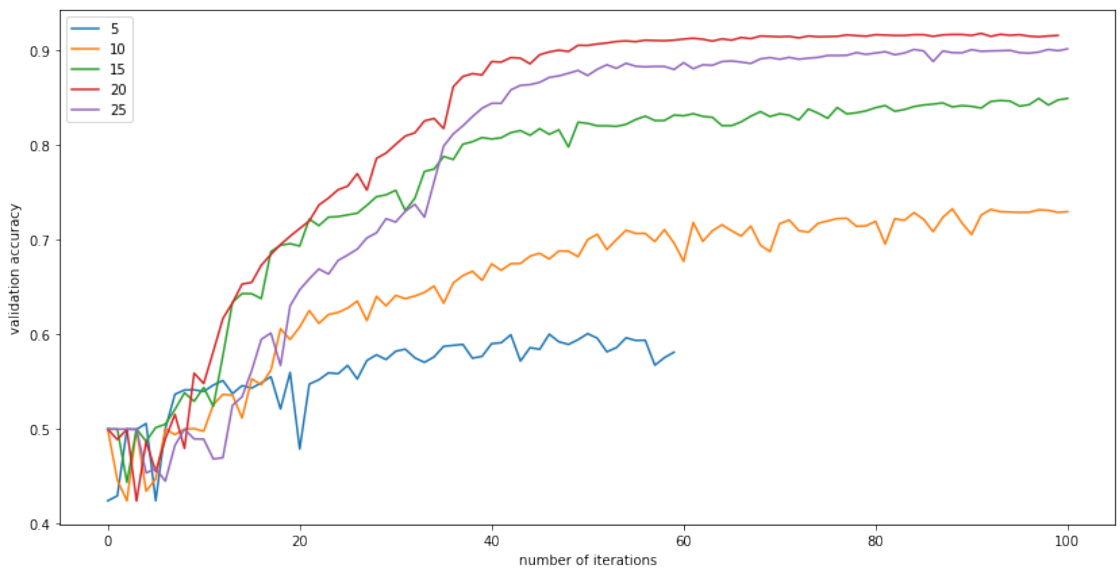
2	47622	0	0	0	0	0	0	0	0	0
3	21121	0	0	0	0	0	0	0	0	0
4	3885	0	0	0	0	0	0	0	0	0
5	1996	0	0	0	0	0	0	0	0	0
6	1424	0	0	0	0	0	0	0	0	0
7	230	0	0	0	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

as we can see with different hidden sizes it is just predicting the label 0 as it forms the large fraction in the train dataset.

The above learning rate is not prefect and we are not able to see the trend as we increase the number of hidden nodes. So, now experimenting with a learning rate of 20.

learning rate 20 **Validation accuracy** over the train process

[124]: <matplotlib.legend.Legend at 0x7fad4add53d0>

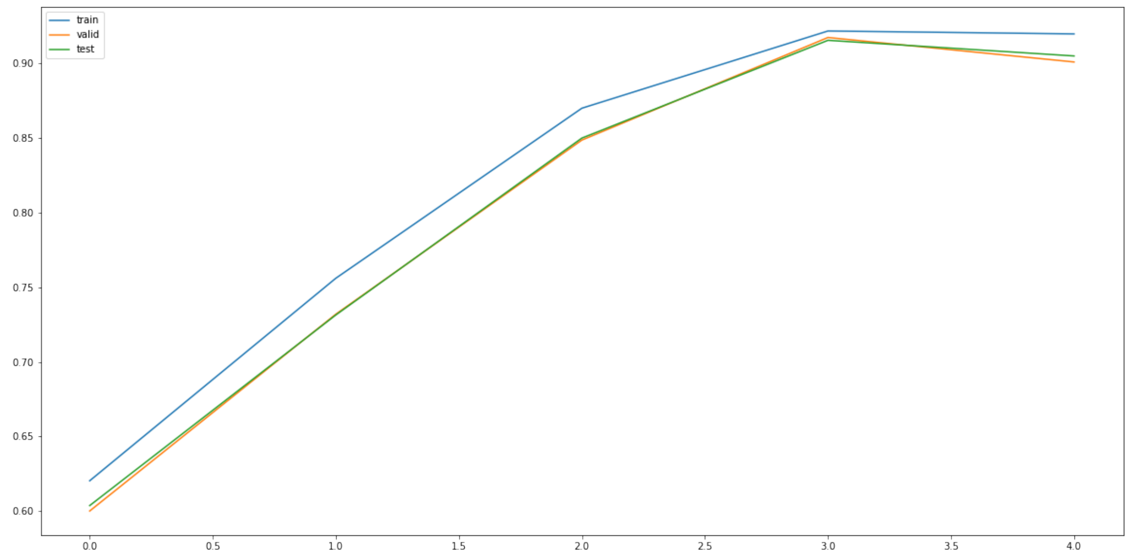


Accuracies and train time for each parameters

[125]:

	5	10	15	20	25
Train	0.620326	0.755999	0.869926	0.921666	0.919716
Validation	0.600080	0.731922	0.848582	0.917299	0.900919
Test	0.603712	0.731391	0.849917	0.915414	0.904944
Time	6.455063	11.242596	11.641337	11.974978	12.534626

[126]: <matplotlib.legend.Legend at 0x7fad19627fd0>



here the trend is clearly visible and we can see as the number of nodes increases the accuracy also increases and time taken by the model to train also increases.

confusion matrix

```
Hidden size : 5

Predicted :       0       1  2  3  4  5  6  7  8  9
Actual :
0            371170  130039  0  0  0  0  0  0  0  0
1            189956  232542  0  0  0  0  0  0  0  0
2             10692   36930  0  0  0  0  0  0  0  0
3              7177   13944  0  0  0  0  0  0  0  0
4              2749    1136  0  0  0  0  0  0  0  0
5              1627     369  0  0  0  0  0  0  0  0
6               143    1281  0  0  0  0  0  0  0  0
7                19     211  0  0  0  0  0  0  0  0
8                10       2  0  0  0  0  0  0  0  0
9                 2       1  0  0  0  0  0  0  0  0

Hidden size : 10

Predicted :       0       1  2  3  4  5  6  7  8  9
Actual :
0            447872   53337  0  0  0  0  0  0  0  0
1            138979  283519  0  0  0  0  0  0  0  0
2              4618   43004  0  0  0  0  0  0  0  0
3              3499   17622  0  0  0  0  0  0  0  0
4              3157     728  0  0  0  0  0  0  0  0
5              1768     228  0  0  0  0  0  0  0  0
6                55    1369  0  0  0  0  0  0  0  0
```

7	4	226	0	0	0	0	0	0	0	0
8	9	3	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

Hidden size : 15

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	481584	19625	0	0	0	0	0	0	0	0
1	54165	368333	0	0	0	0	0	0	0	0
2	440	47182	0	0	0	0	0	0	0	0
3	978	20143	0	0	0	0	0	0	0	0
4	3805	80	0	0	0	0	0	0	0	0
5	1910	86	0	0	0	0	0	0	0	0
6	5	1419	0	0	0	0	0	0	0	0
7	0	230	0	0	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	2	1	0	0	0	0	0	0	0	0

Hidden size : 20

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	498999	2210	0	0	0	0	0	0	0	0
1	6083	416415	0	0	0	0	0	0	0	0
2	48	47574	0	0	0	0	0	0	0	0
3	184	20937	0	0	0	0	0	0	0	0
4	3793	92	0	0	0	0	0	0	0	0
5	1988	8	0	0	0	0	0	0	0	0
6	0	1424	0	0	0	0	0	0	0	0
7	20	210	0	0	0	0	0	0	0	0
8	11	1	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

Hidden size : 25

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	492175	9034	0	0	0	0	0	0	0	0
1	9729	412769	0	0	0	0	0	0	0	0
2	5	47617	0	0	0	0	0	0	0	0
3	434	20687	0	0	0	0	0	0	0	0
4	3537	348	0	0	0	0	0	0	0	0
5	1959	37	0	0	0	0	0	0	0	0
6	0	1424	0	0	0	0	0	0	0	0
7	36	194	0	0	0	0	0	0	0	0
8	10	2	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

As we can see it is mostly prediction the data points as class 0 or 1, as they form large fraction of the dataset, the data set is very skewed so we could use macro f1 as the measure of accuracy.

2.5 Adaptive learning rate

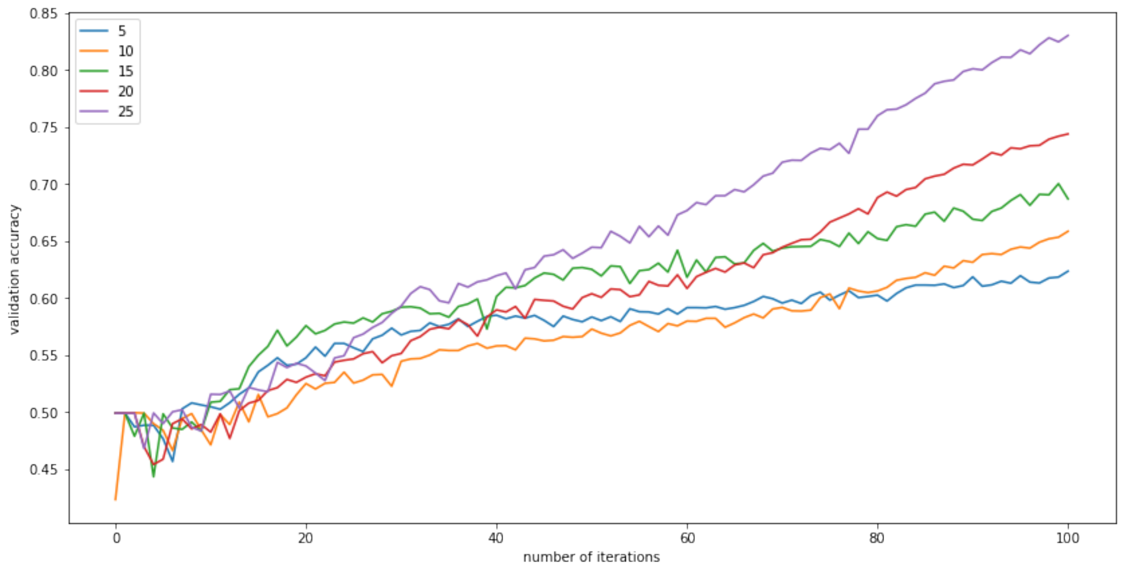
Continuing our experiments with hidden layer size, but with addition of adaptive learning rate.

Stopping criteria: In the stopping criteria, I had to increase value of `epoch_per_stop` to 30 to get better accuracy.

2.5.1 Hidden sizes

Validation accuracy over the train process

[241]: <matplotlib.legend.Legend at 0x7fad3ed3e510>

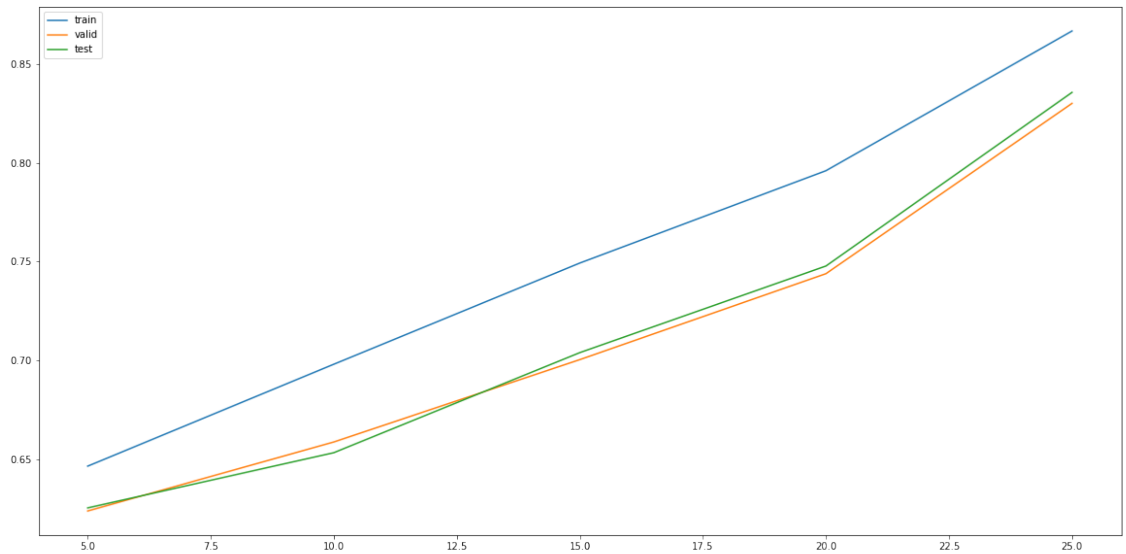


Accuracies and training time

[242]:

	5	10	15	20	25
Train	0.646371	0.698010	0.749300	0.796091	0.866877
Validation	0.623652	0.658610	0.700360	0.743907	0.830204
Test	0.625218	0.653189	0.703913	0.747786	0.835773
Time	10.790469	10.734794	11.779319	12.707102	13.536842

[243]: <matplotlib.legend.Legend at 0x7fad4ad43f50>



Confusion matrix

Hidden size : 5

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	420532	80677	0	0	0	0	0	0	0	0
1	217812	204686	0	0	0	0	0	0	0	0
2	12696	34926	0	0	0	0	0	0	0	0
3	8601	12520	0	0	0	0	0	0	0	0
4	3247	638	0	0	0	0	0	0	0	0
5	1685	311	0	0	0	0	0	0	0	0
6	212	1212	0	0	0	0	0	0	0	0
7	45	185	0	0	0	0	0	0	0	0
8	9	3	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

Hidden size : 10

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	407278	93931	0	0	0	0	0	0	0	0
1	176587	245911	0	0	0	0	0	0	0	0
2	8472	39150	0	0	0	0	0	0	0	0
3	3248	17873	0	0	0	0	0	0	0	0
4	3462	423	0	0	0	0	0	0	0	0
5	1660	336	0	0	0	0	0	0	0	0
6	59	1365	0	0	0	0	0	0	0	0
7	3	227	0	0	0	0	0	0	0	0
8	11	1	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

Hidden size : 15

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	432129	69080	0	0	0	0	0	0	0	0
1	150714	271784	0	0	0	0	0	0	0	0
2	4677	42945	0	0	0	0	0	0	0	0
3	2154	18967	0	0	0	0	0	0	0	0
4	2253	1632	0	0	0	0	0	0	0	0
5	1790	206	0	0	0	0	0	0	0	0
6	19	1405	0	0	0	0	0	0	0	0
7	1	229	0	0	0	0	0	0	0	0
8	7	5	0	0	0	0	0	0	0	0
9	2	1	0	0	0	0	0	0	0	0

Hidden size : 20

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	444830	56379	0	0	0	0	0	0	0	0
1	119542	302956	0	0	0	0	0	0	0	0
2	2461	45161	0	0	0	0	0	0	0	0
3	1566	19555	0	0	0	0	0	0	0	0
4	3317	568	0	0	0	0	0	0	0	0
5	1792	204	0	0	0	0	0	0	0	0
6	7	1417	0	0	0	0	0	0	0	0
7	9	221	0	0	0	0	0	0	0	0
8	8	4	0	0	0	0	0	0	0	0
9	2	1	0	0	0	0	0	0	0	0

Hidden size : 25

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	475753	25456	0	0	0	0	0	0	0	0
1	62478	360020	0	0	0	0	0	0	0	0
2	822	46800	0	0	0	0	0	0	0	0
3	397	20724	0	0	0	0	0	0	0	0
4	3657	228	0	0	0	0	0	0	0	0
5	1912	84	0	0	0	0	0	0	0	0
6	2	1422	0	0	0	0	0	0	0	0
7	0	230	0	0	0	0	0	0	0	0
8	10	2	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

As we can see the adaptive learning rate is not helping us much here, the training time has actually increased as compared to previous section, it is taking much longer to converge in this case.

2.6 ReLU activation

This section is about comparing ReLU and sigmoid activation function

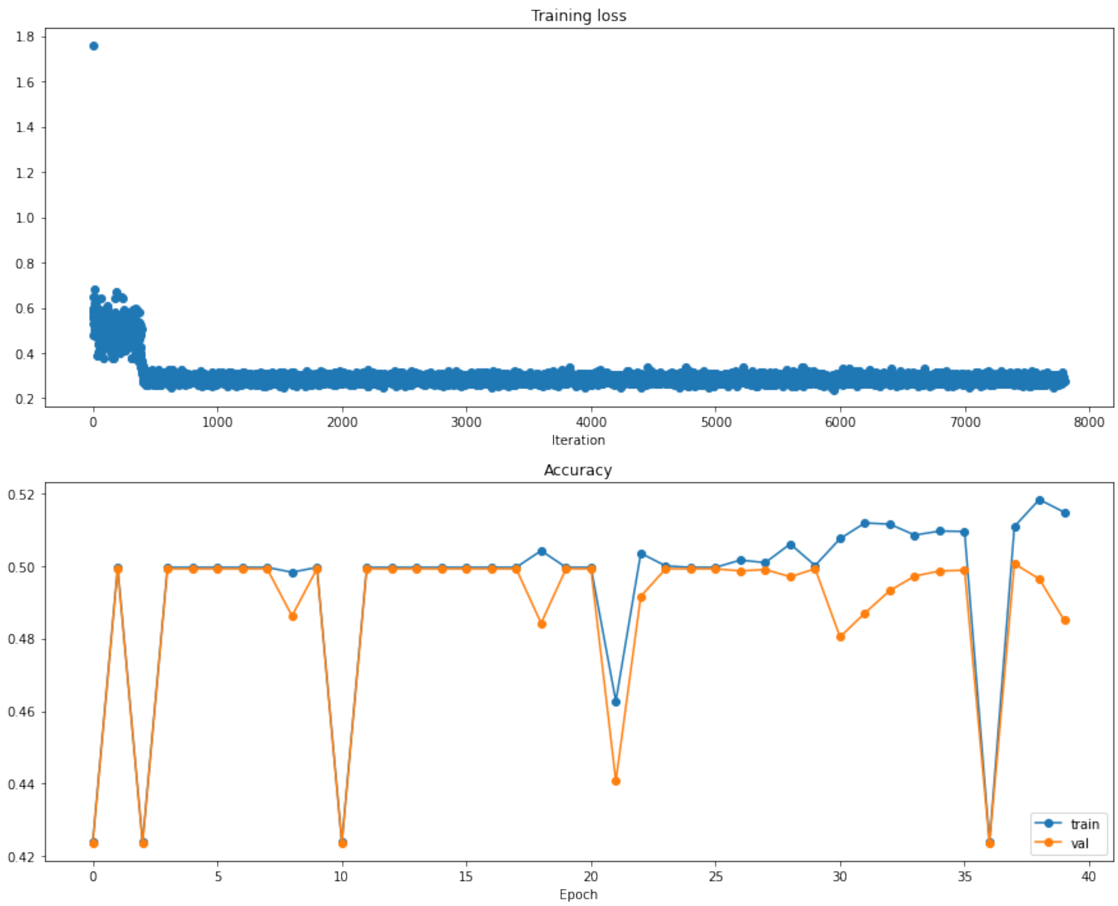
2.6.1 Sigmoid

code for training model

[266]:
```
weight_scale = 1e-1
learning_rate = 10
batch_size = 100

acc = []
model = FullyConnectedNet(hidden_dims=[100, 100], input_dim=85,
                          num_classes=10, weight_scale=weight_scale,
                          activation='sigmoid', sampling='random')
output = model.train(data_dict, batch_size=batch_size, num_epochs=100,
                     epoch_per_stop=10, lr_init=learning_rate,
                     use_adaptive_lr=False, verbose=False)

loss_history, train_acc_history, val_acc_history, best_params = output
model.params = best_params
```



[171]:

	Train	Validation	Test
sigmoid	0.4996	0.499201	0.501209

[172]:

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	501209	0	0	0	0	0	0	0	0	0
1	422498	0	0	0	0	0	0	0	0	0
2	47622	0	0	0	0	0	0	0	0	0
3	21121	0	0	0	0	0	0	0	0	0
4	3885	0	0	0	0	0	0	0	0	0
5	1996	0	0	0	0	0	0	0	0	0
6	1424	0	0	0	0	0	0	0	0	0
7	230	0	0	0	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

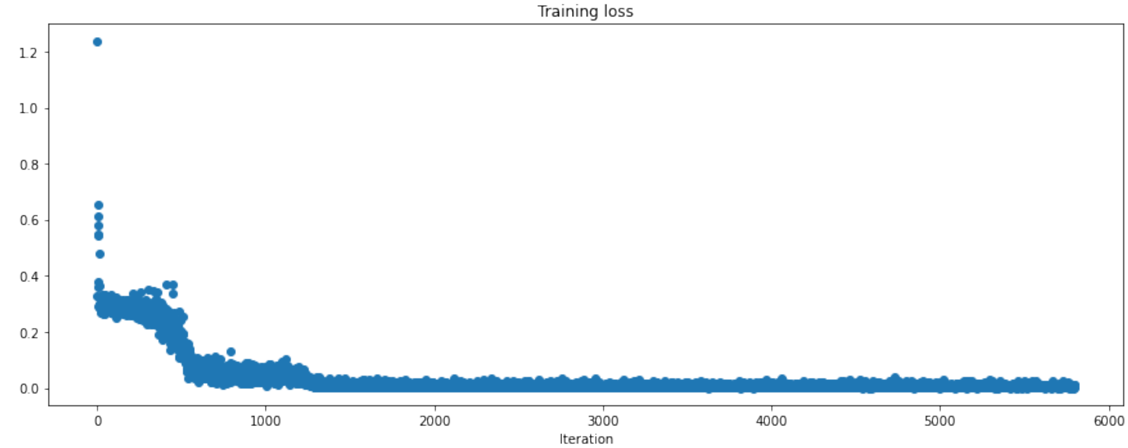
2.6.2 ReLU

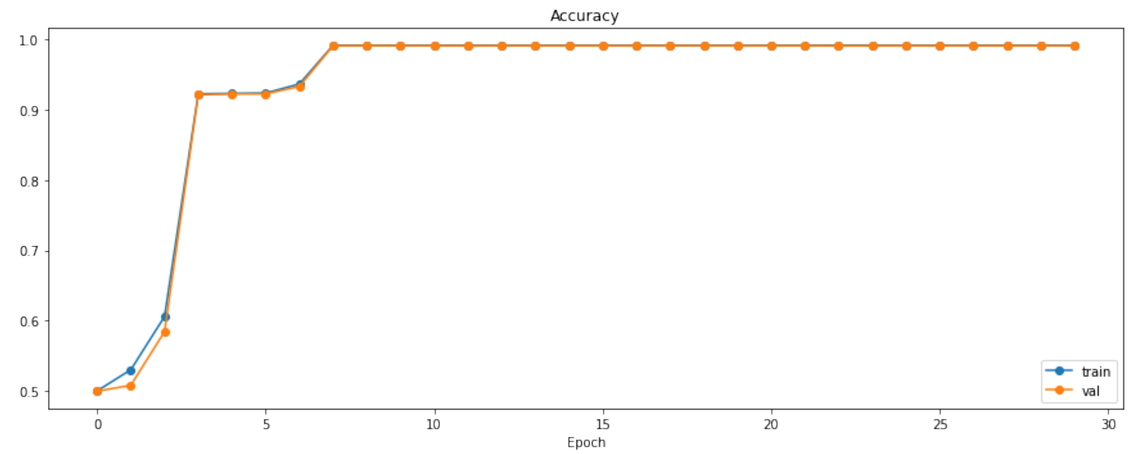
Code for training reLU model

[281]:
```
weight_scale = 1e-1
learning_rate = 10
batch_size = 100

model = FullyConnectedNet(hidden_dims=[100, 100], input_dim=85,
                          num_classes=10, weight_scale=weight_scale,
                          activation='relu', sampling='random')
output = model.train(data_dict, batch_size=batch_size, num_epochs=100,
                 epoch_per_stop=10, lr_init=learning_rate,
                 use_adaptive_lr=True, verbose=False)

loss_history, train_acc_history, val_acc_history, best_params = output
model.params = best_params
```





[284]:

	Train	Validation	Test
relu	0.992152	0.99161	0.99239

[285]:

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	501177	32	0	0	0	0	0	0	0	0
1	0	422490	8	0	0	0	0	0	0	0
2	0	17	47605	0	0	0	0	0	0	0
3	0	0	3	21118	0	0	0	0	0	0
4	3819	66	0	0	0	0	0	0	0	0
5	1995	1	0	0	0	0	0	0	0	0
6	0	0	1149	275	0	0	0	0	0	0
7	0	0	0	230	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

2.6.3 Comparision

Accuracy

[179]:

	Train	Validation	Test
sigmoid	0.499600	0.499201	0.501209
sigmoid	0.992152	0.991610	0.992444

As we can see train is far easier in ReLU, as compared to sigmoid function. And we get better accuracy with ReLU.

Confusion matrix

Predicted : Actual :	0	1	2	3	4	5	6	7	8	9
0	501209	0	0	0	0	0	0	0	0	0
1	422498	0	0	0	0	0	0	0	0	0
2	47622	0	0	0	0	0	0	0	0	0
3	21121	0	0	0	0	0	0	0	0	0
4	3885	0	0	0	0	0	0	0	0	0
5	1996	0	0	0	0	0	0	0	0	0
6	1424	0	0	0	0	0	0	0	0	0
7	230	0	0	0	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

Predicted : Actual :	0	1	2	3	4	5	6	7	8	9
0	501209	0	0	0	0	0	0	0	0	0
1	0	422498	0	0	0	0	0	0	0	0
2	0	6	47616	0	0	0	0	0	0	0
3	0	0	0	21121	0	0	0	0	0	0
4	3885	0	0	0	0	0	0	0	0	0
5	1996	0	0	0	0	0	0	0	0	0
6	0	0	1222	202	0	0	0	0	0	0
7	0	0	0	230	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

We can see here that we are getting diagonal entries for other than class 0 and 1

We can see with the increase layer we are getting better results as compared single hidden layer

2.7 MLPClassifier

Code for train with sklearn

[268]:

```
clf = MLPClassifier(random_state=1, solver='sgd',
                    hidden_layer_sizes=(100, 100), alpha=0, batch_size=100,
                   learning_rate_init=1e-1, learning_rate='adaptive')

clf.fit(data_dict['X_train'], data_dict['y_train'])
```

[268]:
```
MLPClassifier(alpha=0, batch_size=100, hidden_layer_sizes=(100, 100),
              learning_rate='adaptive', learning_rate_init=0.1, random_state=1,
              solver='sgd')
```

Accuracy

[278]:

	Train	Validation	Test
sklearn	1.000000	0.990012	0.991111
relu	0.923465	0.922693	0.923707

As we can see the results are better with sklearn library but the results are comparable.

Confusion matrix

Confusion matrix with sklearn

[279]:

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	500666	37	0	0	477	29	0	0	0	0
1	8	422410	80	0	0	0	0	0	0	0
2	0	1065	46364	191	0	0	2	0	0	0
3	0	19	581	20460	0	0	14	47	0	0
4	3374	67	0	0	440	4	0	0	0	0
5	1659	1	0	0	22	314	0	0	0	0
6	0	0	330	631	0	0	457	6	0	0
7	0	0	0	230	0	0	0	0	0	0
8	7	0	0	0	0	5	0	0	0	0
9	0	2	0	0	1	0	0	0	0	0

confusion matrix with our ReLU

[286]:

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	501177	32	0	0	0	0	0	0	0	0
1	0	422490	8	0	0	0	0	0	0	0
2	0	17	47605	0	0	0	0	0	0	0
3	0	0	3	21118	0	0	0	0	0	0
4	3819	66	0	0	0	0	0	0	0	0
5	1995	1	0	0	0	0	0	0	0	0
6	0	0	1149	275	0	0	0	0	0	0
7	0	0	0	230	0	0	0	0	0	0
8	12	0	0	0	0	0	0	0	0	0
9	3	0	0	0	0	0	0	0	0	0

sklearn is performing better

2.8 Skewed sampling

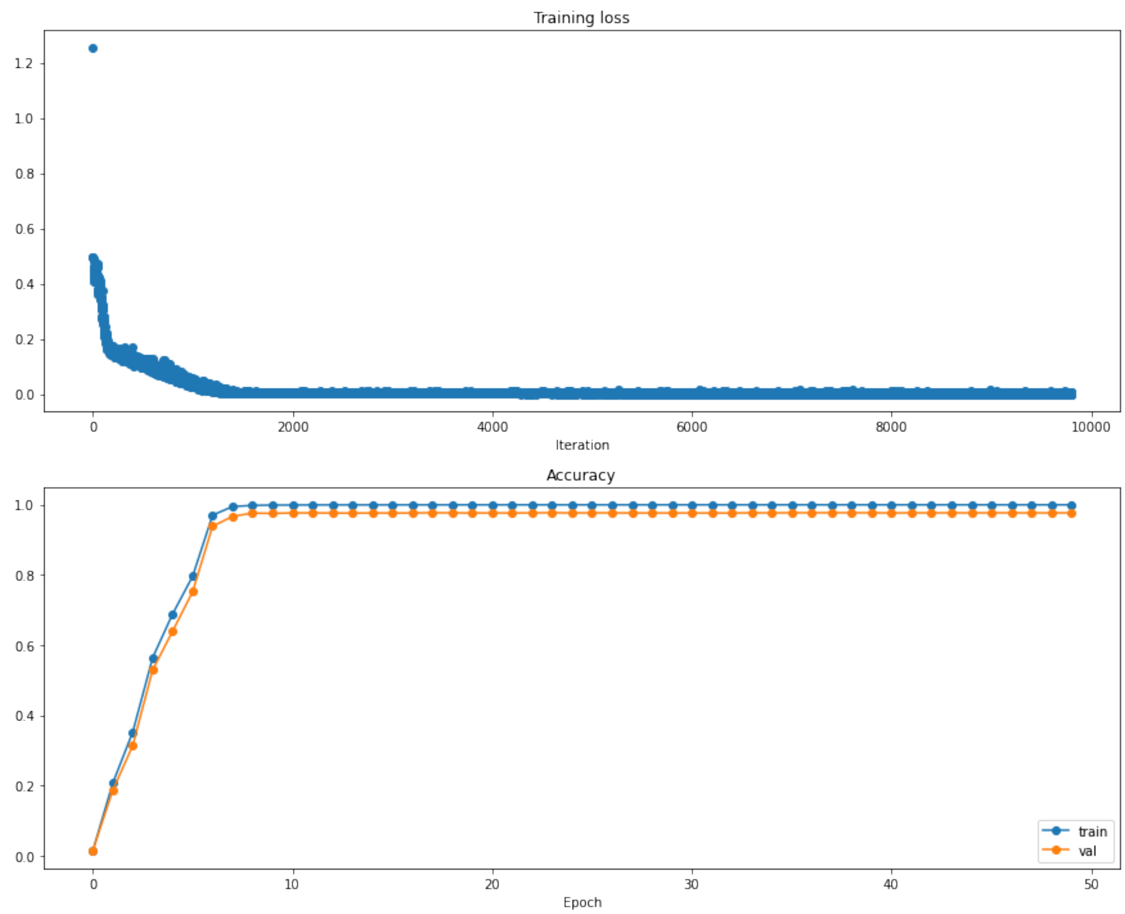
Code for training where for each mini-batch we have equal representation from all the classes.

[291]:
```
weight_scale = 1e-1
learning_rate = 10
batch_size = 100

model = FullyConnectedNet(hidden_dims=[100, 100], input_dim=85,
                          num_classes=10, weight_scale=weight_scale,
                          activation='relu', sampling='skewed')

output = model.train(data_dict, batch_size=batch_size, num_epochs=100,
                 epoch_per_stop=10, lr_init=learning_rate,
                 use_adaptive_lr=True, verbose=False)

loss_history, train_acc_history, val_acc_history, best_params = output
model.params = best_params
```



Accuracy

[294]:

	Train	Validation	Test
skewed	0.99985	0.977427	0.976343

Confusion matrix

[295]:

Predicted :	0	1	2	3	4	5	6	7	8	9
Actual :										
0	500158	8	0	0	675	350	0	0	11	7
1	2	422286	38	21	135	9	6	0	0	1
2	0	305	43014	4089	9	0	192	13	0	0
3	0	9	10916	9802	1	0	337	56	0	0
4	3596	0	0	0	273	6	0	0	2	8
5	1217	0	0	0	0	759	0	0	18	2
6	0	0	533	838	0	0	51	2	0	0
7	0	0	5	219	0	0	6	0	0	0
8	4	0	0	0	0	8	0	0	0	0
9	1	0	0	0	0	2	0	0	0	0

The result obtained here are the best, class 8 and 9 are also being predicted here