ML-PG Assignment 2

- Divyansh Rastogi (2019464)

Prerequisite

Methods to split the dataset into folds & form training, validation sets:

```
def split(df: pd.DataFrame, folds=5):
    df = df.sample(frac=1, random_state=7).reset_index(drop=True)
    num = len(df) // folds
    left = len(df) % folds
    df_folds = []
    prev = 0
    for i in range(folds):
        length = num + (left > 0)
        df_ = df.iloc[prev: prev + length].copy().reset_index(drop=True)
        df_folds.append(df_)
        prev += length
        left -= 1
    return df_folds

def form_train_val(folds, val_fold):
    train_folds = [folds[i] for i in range(len(folds)) if i ≠ val_fold]
    train_df = pd.concat(train_folds).reset_index(drop=True)
    val_df = folds[val_fold].copy()
    return train_df, val_df
```

Q1.

Dataset: Abalone

Preprocessing:

1. The dataset contained no missing values.
2. The categorical feature 'Sex' was one-hot encoded.
3. Before regression, data is always standardized.

a)

The regression class standardizes data based on the input parameter & uses fit method of sklearn to find the coefficients and intercept. During prediction, the test data is first standardized w.r.t to the training set and then predicted using linear regression equation.

The standardization method is also implemented for all self-implemented classes in the further parts of the assignment.

```python
class Regression:
    """
    Linear Regression
    """
    def __init__(self, normalise):
        self.normalise = normalise # Gaussian Normalisation

    def fit(self, X: pd.DataFrame, y: pd.Series):
        if self.normalise:
            self.mean = X.mean()
            self.std = X.std()
            X = (X - self.mean) / self.std
        lr = LinearRegression()
        lr.fit(X, y)
        self.w = lr.coef_
        self.b = lr.intercept_

    def predict(self, X_test: pd.DataFrame):
        if self.normalise:
            X_test = (X_test - self.mean) / self.std
        y_pred = (X_test * self.w).sum(axis=1) + self.b
        return np.array(y_pred)
```

b)

MSE function:

```python
def MSE(y_actual, y_pred, inbuilt=False):
    if inbuilt:
        return metrics.mean_squared_error(y_actual, y_pred)
    else:
        assert (len(y_actual) == len(y_pred))
        diff = (y_actual - y_pred) ** 2
        return diff.sum() / len(y_actual)
```

Training:

```python
mse_df = {'Validation fold': [], 'Train MSE': [], 'Validation MSE': []}
mse_inbuilt_df = {'Validation fold': [], 'Train MSE': [], 'Validation MSE': []}
for val_fold in range(len(folds)):
    train_df, val_df = form_train_val(folds, val_fold)
    lr = None
    model_filename = f'Weights/1/mse-{val_fold}.sav'
    if os.path.exists(model_filename):
        lr = joblib.load(model_filename)
    else:
        lr = Regression(normalise=True)
        lr.fit(train_df[feature_cols], train_df[out_col])
        joblib.dump(lr, model_filename)

    train_mse = MSE(train_df[out_col], lr.predict(train_df[feature_cols]))
    val_mse = MSE(val_df[out_col], lr.predict(val_df[feature_cols]))
    train_mse_inbuilt = MSE(train_df[out_col], lr.predict(train_df[feature_cols]), inbuilt=True)
    val_mse_inbuilt = MSE(val_df[out_col], lr.predict(val_df[feature_cols]), inbuilt=True)

    mse_df['Validation fold'].append(val_fold)
    mse_df['Train MSE'].append(train_mse)
    mse_df['Validation MSE'].append(val_mse)
    mse_inbuilt_df['Validation fold'].append(val_fold)
    mse_inbuilt_df['Train MSE'].append(train_mse_inbuilt)
    mse_inbuilt_df['Validation MSE'].append(val_mse_inbuilt)
mse_df = pd.DataFrame(mse_df)
mse_inbuilt_df = pd.DataFrame(mse_inbuilt_df)
```

The MSE on train and validation set using self MSE implementation & sklearn MSE implementation is given below: (The results exactly match)

```
mse_df
```

	Validation fold	Train MSE	Validation MSE
0	0	4.733024	5.127119
1	1	4.795933	4.844730
2	2	4.712582	5.505303
3	3	4.870471	4.549547
4	4	4.843510	4.684619

```
mse_inbuilt_df
```

	Validation fold	Train MSE	Validation MSE
0	0	4.733024	5.127119
1	1	4.795933	4.844730
2	2	4.712582	5.505303
3	3	4.870471	4.549547
4	4	4.843510	4.684619

Mean training MSE & mean validation MSE:

```
mse_df.mean()[['Train MSE', 'Validation MSE']]

Train MSE         4.791104
Validation MSE    4.942263
dtype: float64
```

c)

Linear regression using normal equations:

```
class NormalEq:
    """
    Linear Regression using normal equations
    """
    def __init__(self, normalise):
        self.normalise = normalise # Gaussian Normalisation

    def fit(self, X_df: pd.DataFrame, y_df: pd.Series):
        if self.normalise:
            self.mean = X_df.mean()
            self.std = X_df.std()
            X_df = (X_df - self.mean) / self.std
        else:
            X_df = X_df.copy()
        X_df['_bias'] = np.ones(len(X_df))
        X = X_df.to_numpy()
        y = np.array(y_df)
        self.w = np.linalg.pinv(X.T @ X) @ (X.T @ y)

    def predict(self, X_test_df: pd.DataFrame):
        if self.normalise:
            X_test_df = (X_test_df - self.mean) / self.std
        else:
            X_test_df = X_test_df.copy()
        X_test_df['_bias'] = np.ones(len(X_test_df))
        X_test = X_test_df.to_numpy()
        return X_test @ self.w
```

Training:

```
mse_ne_df = {'Validation fold': [], 'Train MSE': [], 'Validation MSE': []}
for val_fold in range(len(folds)):
    train_df, val_df = form_train_val(folds, val_fold)
    lr_ne = None
    model_filename = f'Weights/1/mse-ne-{val_fold}.sav'
    if os.path.exists(model_filename):
        lr_ne = joblib.load(model_filename)
    else:
        lr_ne = NormalEq(normalise=True)
        lr_ne.fit(train_df[feature_cols], train_df[out_col])
        joblib.dump(lr_ne, model_filename)

    train_mse = MSE(train_df[out_col], lr_ne.predict(train_df[feature_cols]))
    val_mse = MSE(val_df[out_col], lr_ne.predict(val_df[feature_cols]))

    mse_ne_df['Validation fold'].append(val_fold)
    mse_ne_df['Train MSE'].append(train_mse)
    mse_ne_df['Validation MSE'].append(val_mse)
mse_ne_df = pd.DataFrame(mse_ne_df)
```

Training and validation MSE: (The results match with the previous results)

```
mse_ne_df
```

	Validation fold	Train MSE	Validation MSE
0	0	4.733024	5.127119
1	1	4.795933	4.844730
2	2	4.712582	5.505303
3	3	4.870471	4.549547
4	4	4.843510	4.684619

```
mse_ne_df.mean()[['Train MSE', 'Validation MSE']]
```

```
Train MSE         4.791104
Validation MSE    4.942263
dtype: float64
```

d)

Training using sklearn linear regression: (explicit standardization is done)

```
mse_sklearn_df = {'Validation fold': [], 'Train MSE': [], 'Validation MSE': []}
for val_fold in range(len(folds)):
    train_df, val_df = form_train_val(folds, val_fold)
    # explicit standardization
    mean_train_df = train_df[feature_cols].mean()
    std_train_df = train_df[feature_cols].std()
    train_df[feature_cols] = (train_df[feature_cols] - mean_train_df) / std_train_df
    val_df[feature_cols] = (val_df[feature_cols] - mean_train_df) / std_train_df

    lr_sklearn = None
    model_filename = f'Weights/1/mse-sklearn-{val_fold}.sav'
    if os.path.exists(model_filename):
        lr_sklearn = joblib.load(model_filename)
    else:
        lr_sklearn = LinearRegression()
        lr_sklearn.fit(train_df[feature_cols], train_df[out_col])
        joblib.dump(lr_sklearn, model_filename)

    train_mse = MSE(train_df[out_col], lr_sklearn.predict(train_df[feature_cols]), inbuilt=True)
    val_mse = MSE(val_df[out_col], lr_sklearn.predict(val_df[feature_cols]), inbuilt=True)

    mse_sklearn_df['Validation fold'].append(val_fold)
    mse_sklearn_df['Train MSE'].append(train_mse)
    mse_sklearn_df['Validation MSE'].append(val_mse)
mse_sklearn_df = pd.DataFrame(mse_sklearn_df)
```

MSE using sklearn linear regression.

```
mse_sklearn_df
```

	Validation fold	Train MSE	Validation MSE
0	0	4.733024	5.127119
1	1	4.795933	4.844730
2	2	4.712582	5.505303
3	3	4.870471	4.549547
4	4	4.843510	4.684619

```
mse_sklearn_df.mean()[['Train MSE', 'Validation MSE']]
```

```
Train MSE         4.791104
Validation MSE    4.942263
dtype: float64
```

There is no deviation observed from self implemented linear regression methods.

Q2.

Dataset: Diabetes

Preprocessing:

1. No null values are observed.
2. Data is always standardized before regression.

a) Analysis:

1. No null values are observed.

```
df.isna().any()
```

```
Pregnancies                 False
Glucose                     False
BloodPressure               False
SkinThickness               False
Insulin                     False
BMI                         False
DiabetesPedigreeFunction    False
Age                         False
Outcome                     False
dtype: bool
```

2. No categorical data is observed.

```
df.dtypes

Pregnancies                   int64
Glucose                       int64
BloodPressure                 int64
SkinThickness                 int64
Insulin                       int64
BMI                         float64
DiabetesPedigreeFunction    float64
Age                           int64
Outcome                       int64
dtype: object
```

3. There is a 2:1 ratio class imbalance.

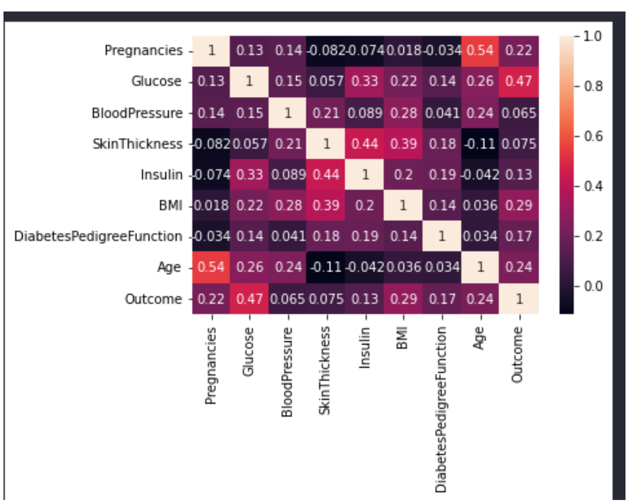
```
df['Outcome'].value_counts()

0    500
1    268
Name: Outcome, dtype: int64
```

4. Using describe() method of pandas, the following statistics are observed:

	Pregnancies	Glucose	BloodPressure	SkinThickness	Insulin	BMI	DiabetesPedigreeFunction	Age	Outcome
count	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000	768.000000
mean	3.845052	120.894531	69.105469	20.536458	79.799479	31.992578	0.471876	33.240885	0.348958
std	3.369578	31.972618	19.355807	15.952218	115.244002	7.884160	0.331329	11.760232	0.476951
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.078000	21.000000	0.000000
25%	1.000000	99.000000	62.000000	0.000000	0.000000	27.300000	0.243750	24.000000	0.000000
50%	3.000000	117.000000	72.000000	23.000000	30.500000	32.000000	0.372500	29.000000	0.000000
75%	6.000000	140.250000	80.000000	32.000000	127.250000	36.600000	0.626250	41.000000	1.000000
max	17.000000	199.000000	122.000000	99.000000	846.000000	67.100000	2.420000	81.000000	1.000000

5. Correlation heatmap is plotted using seaborn. Maximum correlation of 0.54 and minimum correlation of -0.11 is observed. Overall there is no significantly high correlation observed between the features.



6. A pairplot is also plotted with hue=Outcome using seaborn. A subsection of the plot is shown:

```
sns.pairplot(df, hue='Outcome')
plt.show()
```

b)

Utility functions: (BCE is binary cross entropy loss)

```python
def BCE(y_actual, y_pred):
    return -np.mean(y_actual * np.log(y_pred) + (1 - y_actual) * np.log(1 - y_pred))

def accuracy(y_actual, y_pred):
    return sum([y_pred[i] == y_actual[i] for i in range(len(y_actual))]) / len(y_actual)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

Class LogRegression:

1. Training data and validating data is standardized with training mean and standard deviation.
2. Uses Batch Gradient Descent for weight updates
3. Utilises Binary Cross Entropy loss as loss criterion (with L2 regularization)

4. Loss function:

```
def loss(self, y_actual, y_pred):
    return BCE(y_actual, y_pred) + self.reg_lambda * np.sum(self.w ** 2)
```

5. Probability function:

```
def prob(self, X):
    return sigmoid(X @ self.w)
```

6. Gradient function:

```
def grad(self, X, y_actual):
    y_pred = self.prob(X)
    gradient = ((y_pred - y_actual) @ X) / self.num_samples + 2 * self.reg_lambda * self.w
    return gradient
```

7. Predict function:

```
def predict(self, X_test):
    if self.normalise:
        X_test = (X_test - self.mean) / self.std
    else:
        X_test = X_test.copy()
    X_test['_bias'] = np.ones(len(X_test))
    y_pred = np.array([int((self.w.T @ x) >= 0) for x in X_test.to_numpy()])
    return y_pred
```

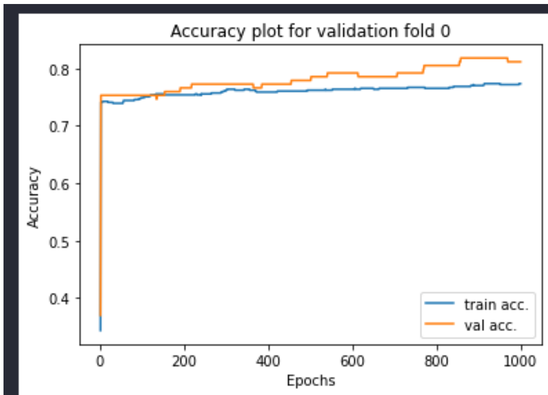
c)

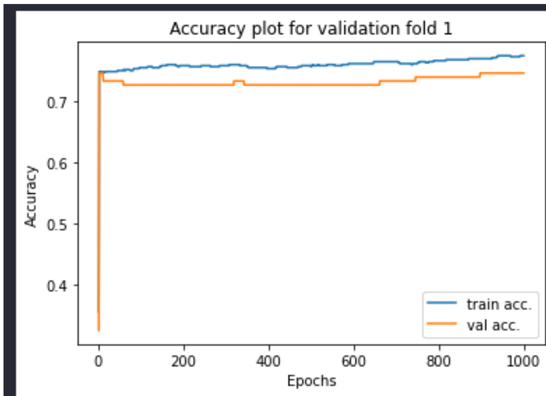
Each fold is run for 1000 epochs with learning rate: 0.01, and lambda: 0.

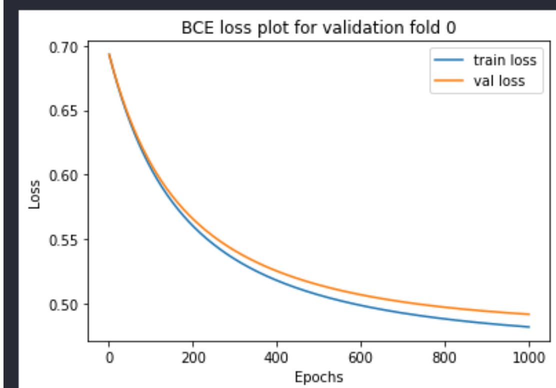
Table representing fold-wise accuracy:

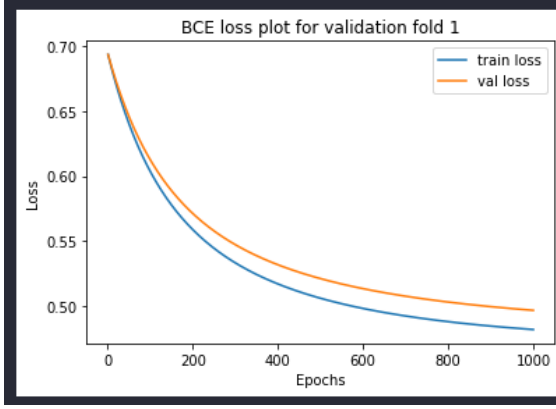
Validation fold	Train Acc.	Validation Acc.
0	0.773616	0.811688
1	0.775244	0.746753
2	0.768730	0.779221
3	0.786992	0.718954
4	0.760976	0.771242

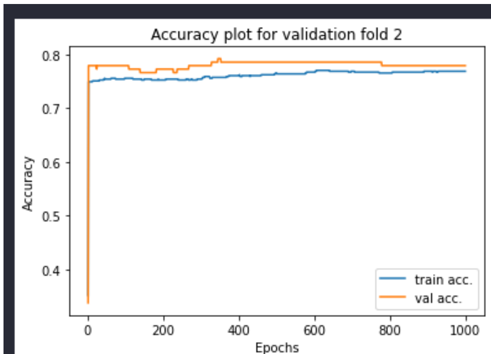
The plots for each fold are given below. It's mainly observed that the training loss starts decreasing with epochs while the validation loss starts gradually decreasing (decreasing with a smaller gradient) with epochs indicating the (slight) loss of generalizability on validation set. W.r.t accuracy, the model has a steep ascent in accuracy in the few first epochs, and then slowly seems to improve both on the test and validation set. No evident overfitting is observed and the model is able to generalize well.

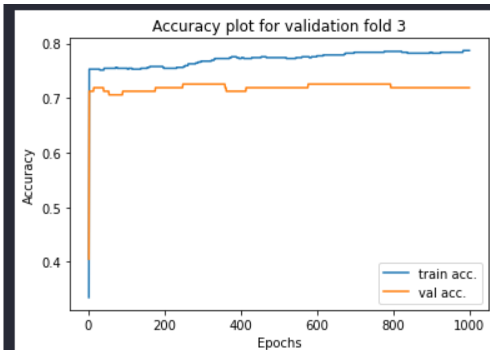

Accuracy plot for validation fold 0
Accuracy
Epochs
train acc.
val acc.

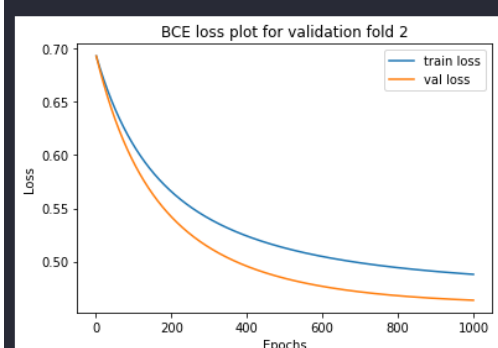

Accuracy plot for validation fold 1
Accuracy
Epochs
train acc.
val acc.

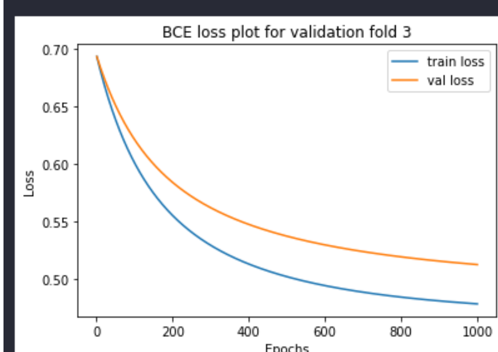

BCE loss plot for validation fold 0
Loss
Epochs
train loss
val loss

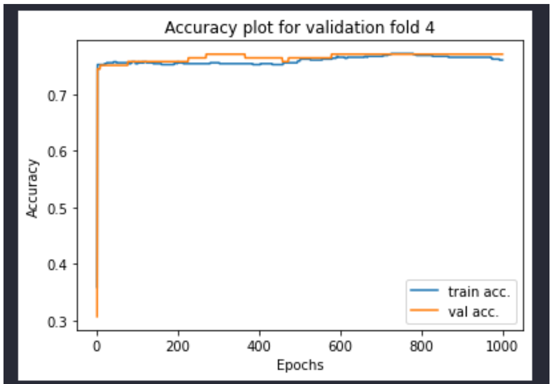

BCE loss plot for validation fold 1
Loss
Epochs
train loss
val loss

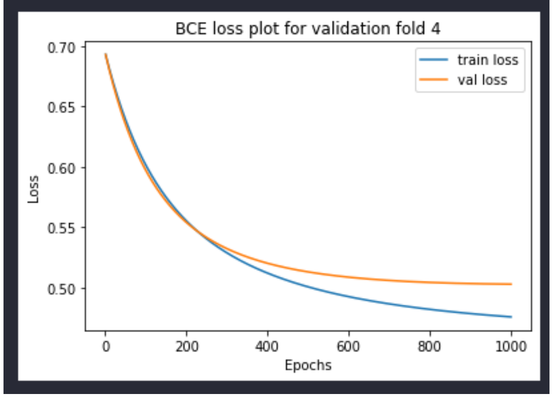

Accuracy plot for validation fold 2
Accuracy
Epochs
train acc.
val acc.


Accuracy plot for validation fold 3
Accuracy
Epochs
train acc.
val acc.


BCE loss plot for validation fold 2
Loss
Epochs
train loss
val loss


BCE loss plot for validation fold 3
Loss
Epochs
train loss
val loss





d)

The loss in L2 regularisation takes the form:

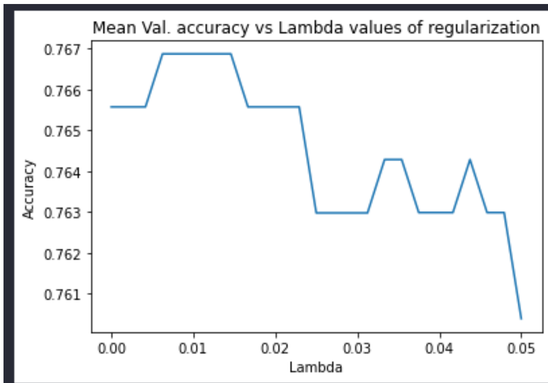
```
def loss(self, y_actual, y_pred):
    return BCE(y_actual, y_pred) + self.reg_lambda * np.sum(self.w ** 2)
```

The gradient in L2 regularisation take the form:

```
def grad(self, X, y_actual):
    y_pred = self.prob(X)
    gradient = ((y_pred - y_actual) @ X) / self.num_samples + 2 * self.reg_lambda * self.w
    return gradient
```

For finding optimal lambda, a grid search is performed over 25 values in linspace (0 → 0.05). The metric to decide the optimality of lambda is the highest mean validation accuracy over all folds. The plot for mean validation accuracy vs lambdas is given below along with the choice of optimal lambda:

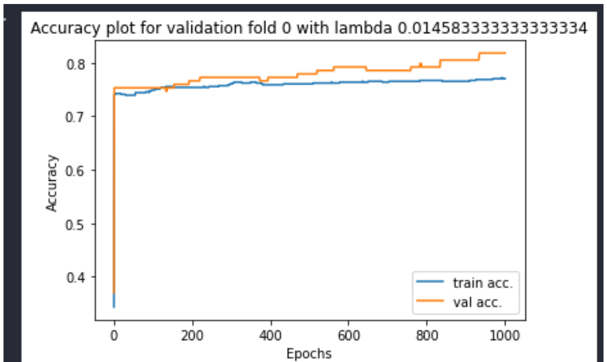
There are multiple values (specifically 0.006 → 0.014) where highest mean validation accuracy is observed. The lambda with the highest value among the highest mean validation accuracy is chosen to demonstrate the effect of choice of lambda on the model in later experiments.

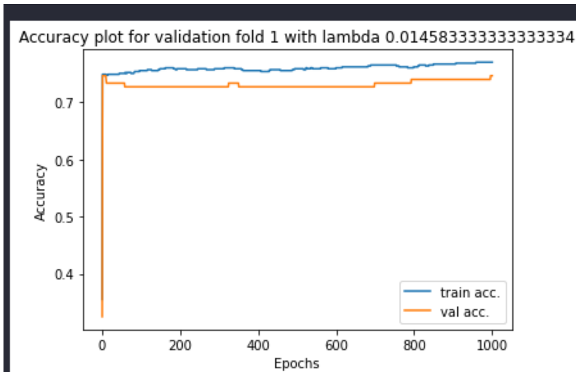


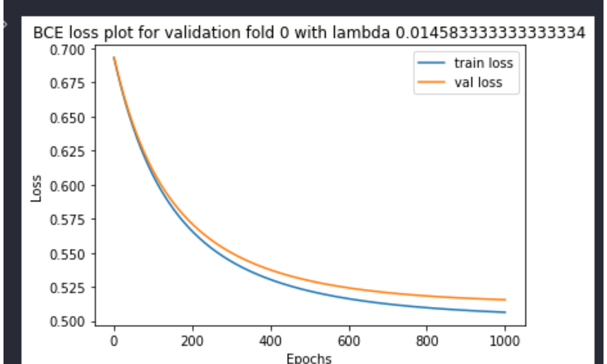

```
optimal_reg_lambda
```

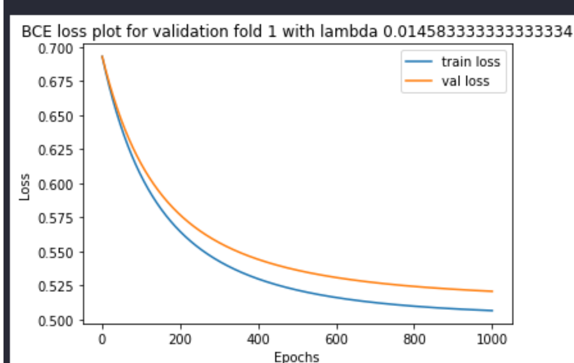
```
0.014583333333333334
```

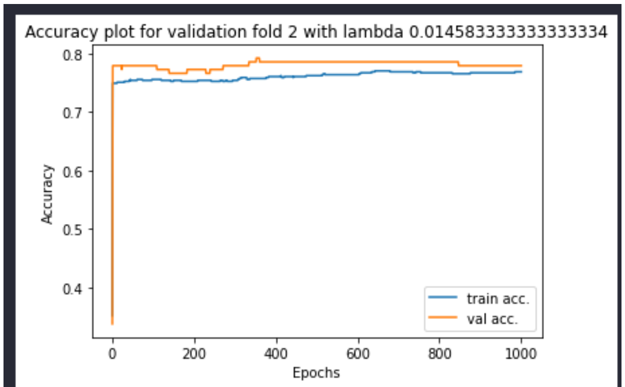
Using lambda=0.014583, the above tables and plots are again plotted.

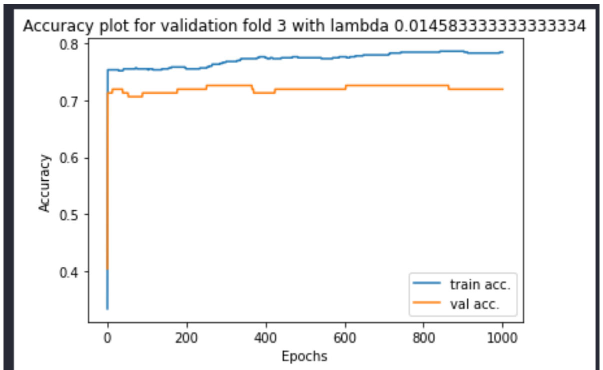


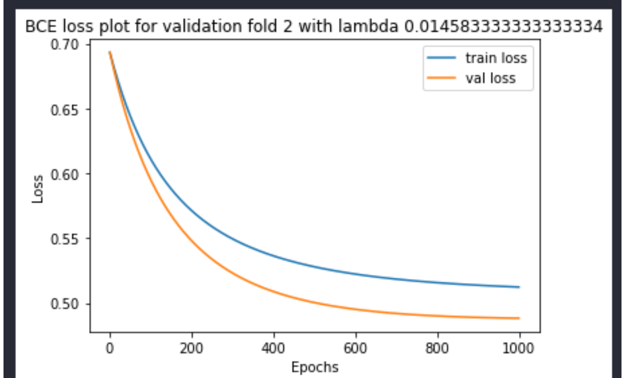


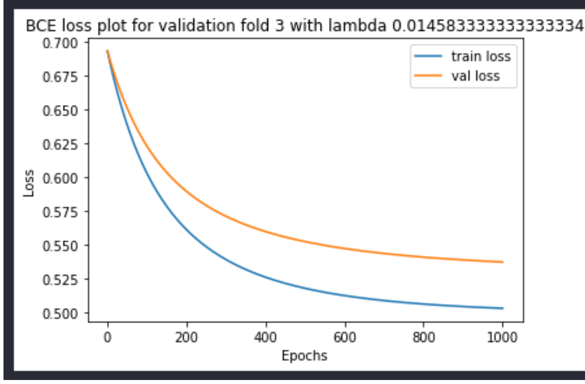


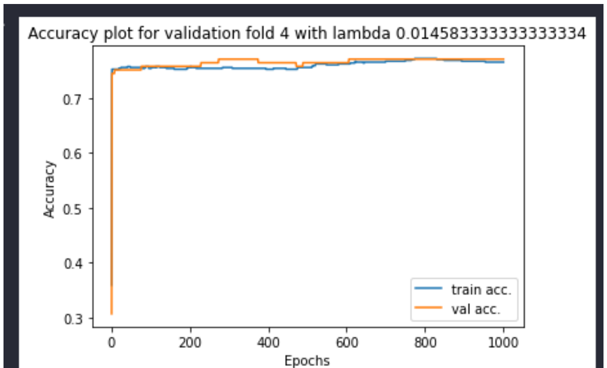


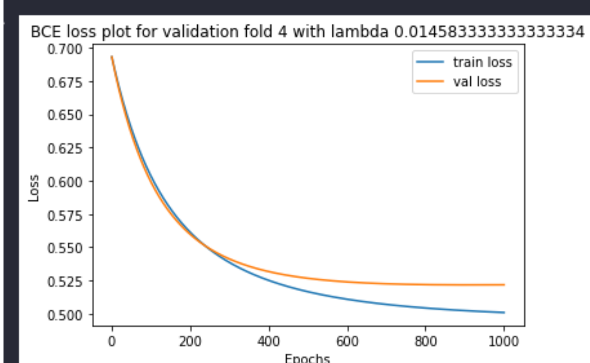













```
log_reg_lambda_df
```

	Validation fold	Train Acc.	Validation Acc.
0	0	0.770358	0.818182
1	1	0.770358	0.746753
2	2	0.768730	0.779221
3	3	0.783740	0.718954
4	4	0.765854	0.771242

.

Performance comparison:

Comparing the mean training and validation accuracy of regularised and unregularised models.

Unregularised:

```
log_reg_df.mean()[['Train Acc.', 'Validation Acc.']]
```

```
Train Acc.         0.773111
Validation Acc.    0.765572
dtype: float64
```

Regularised:

```
log_reg_lambda_df.mean()[['Train Acc.', 'Validation Acc.']]
```

```
Train Acc.         0.771808
Validation Acc.    0.766870
dtype: float64
```

In the unregularised model, a slightly higher training accuracy is observed w.r.t to regularised model although a slightly lower validation accuracy is observed w.r.t to the regularised model. This indicates a higher extent of generalizability in the regularised model.

Comparing the mean distance of weights from origin in both the models:

```
mean_dist_unregularized = np.mean([np.sum(x.w ** 2) for x in log_reg_models])
mean_dist_regularized = np.mean([np.sum(x.w ** 2) for x in log_reg_olambda_models])
mean_dist_unregularized, mean_dist_regularized
```

```
(1.4866829173322713, 1.1995173286141736)
```

The mean weight distance in the unregularised model is higher than the mean weight distance of the regularised model indicating lower weight values in the regularised model.

e)

Sklearn results with unregularised model:

Model used:

```
log_reg = LogisticRegression(penalty='none', random_state=7)
```

```
log_reg_sklearn_df.mean()[['Train Acc.', 'Validation Acc.']]
0.7s
```

```
Train Acc.         0.722323
Validation Acc.    0.712215
dtype: float64
```

Validation fold	Train Acc.	Validation Acc.
0	0.734528	0.720779
1	0.701954	0.701299
2	0.705212	0.733766
3	0.744715	0.679739
4	0.725203	0.725490

Sklearn results with regularised model:

Model used:

```
log_reg = LogisticRegression(penalty='l2', random_state=7)
```

```
log_reg_l2_sklearn_df.mean()[['Train Acc.', 'Validation Acc.']]
✓ 0.3s

Train Acc.         0.723299
Validation Acc.    0.713522
```

```
log_reg_l2_sklearn_df
✓ 0.7s
```

Validation fold	Train Acc.	Validation Acc.
0	0.734528	0.720779
1	0.705212	0.701299
2	0.705212	0.733766
3	0.746341	0.679739
4	0.725203	0.732026

Self-implemented logistic regression outperforms sklearn's implementation. This can arise due to multiple reasons, primarily hyperparameter tuning (self implemented model runs for 1000 epochs while sklearn's logistic model runs for maximum 100 epochs) & difference in convergence algorithms.

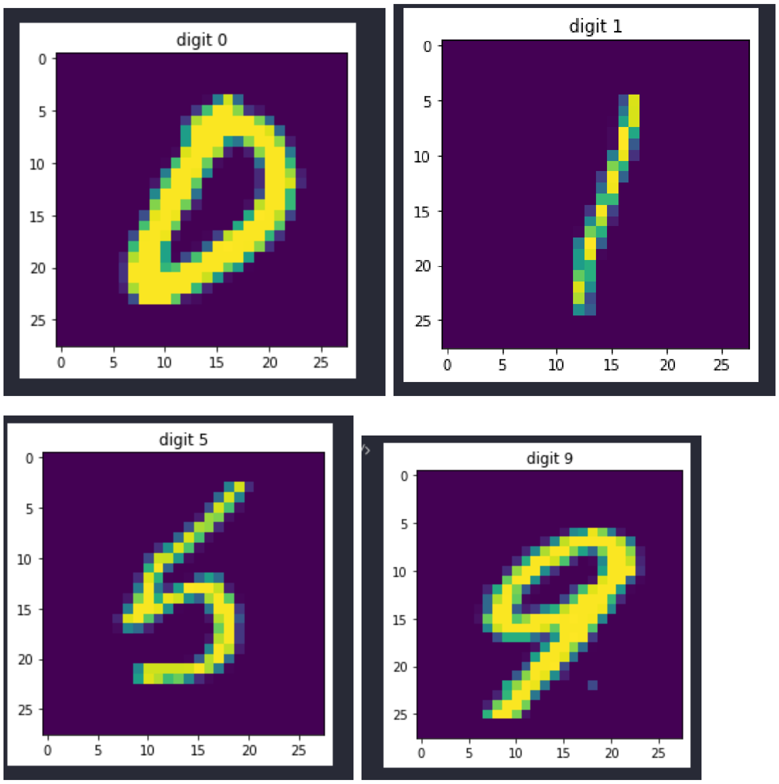
Q3.

Dataset: MNIST (loaded train set using idx2numpy)

```
orig_X = idx2numpy.convert_from_file('/content/drive/My Drive/Data Colab/ml-pg-assignment-3/train-images.idx3-ubyte')
y = idx2numpy.convert_from_file('/content/drive/My Drive/Data Colab/ml-pg-assignment-3/train-labels.idx1-ubyte')
```

a)

5 instances of each class are visualised using imshow() method of matplotlib. Here are some samples visualisations for reference:



b)

Logistic regression class is extended to One-vs-One approach. For n classes, n*(n-1)/2 classifiers are needed where each classifier classifies only between a single pair of classes. At last, a majority vote is taken to determine the output class label. OVO requires dataset bifurcation and is computationally expensive. Each sub logistic classifier standardizes the data & is run for 2000 epochs with lr=0.01 & lambda=0.01. Fit method of OVO logistic classifier:

```python
def fit_ovo(self, _X, y, _X_val, y_val, epochs, display):
    if display:
        print('* Fitting OVO models')
    for c1 in range(self.num_classes):
        for c2 in range(c1 + 1, self.num_classes):
            idx = (y == c1) | (y == c2)
            idx_val = (y_val == c1) | (y_val == c2)
            X_c1_c2 = _X[idx]
            X_val_c1_c2 = _X_val[idx_val]
            y_c1_c2 = np.array([int(u == c1) for u in y[idx]])
            y_val_c1_c2 = np.array([int(u == c1) for u in y_val[idx_val]])
            self.models[(c1, c2)] = LogRegression_Base()
            self.models[(c1, c2)].fit(X_c1_c2, y_c1_c2, X_val_c1_c2, y_val_c1_c2, epochs)
            if display:
                print(f'> Fitted model {c1}, {c2}')
                print(f'> Acc. train: {accuracy(y_c1_c2, self.models[(c1, c2)].predict(X_c1_c2))}')
                if _X_val is not None:
                    print(f'> Acc. val: {accuracy(y_val_c1_c2, self.models[(c1, c2)].predict(X_val_c1_c2))}')
                print()
    if display:
        print('=========')
```

Predict method:

```
def predict_ovo(self, X):
    predictions = {}
    for c1 in range(self.num_classes):
        for c2 in range(c1 + 1, self.num_classes):
            p = self.models[(c1, c2)].predict(X)
            predictions[(c1, c2)] = [c1 if u else c2 for u in p]
    y_ = np.array(list(predictions.values()))
    y = np.array([mode(y_[:, i]) for i in range(X.shape[0])])
    return y
```

Accuracy table:

ovo_df

Validation fold	Train Acc.	Validation Acc.
0	0.916687	0.911667
1	0.915875	0.916083
2	0.917542	0.909167
3	0.916542	0.915333
4	0.916562	0.913417

Class-wise train accuracy for each fold:

ovo_classwise_train_df

	0	1	2	3	4	5	6	7	8	9
0	0.959798	0.980933	0.881579	0.872821	0.931604	0.899816	0.958017	0.915965	0.851305	0.905621
1	0.959179	0.981381	0.883784	0.869794	0.934118	0.900676	0.957161	0.912281	0.849033	0.903442
2	0.959525	0.981441	0.886274	0.874143	0.932305	0.896283	0.956301	0.915431	0.851788	0.910680
3	0.959141	0.979951	0.884672	0.870156	0.934866	0.900874	0.955635	0.914354	0.851059	0.906316
4	0.957845	0.981537	0.888002	0.872353	0.933734	0.897856	0.959553	0.912818	0.850800	0.903293

Class-wise validation accuracy for each fold:

ovo_classwise_val_df

	0	1	2	3	4	5	6	7	8	9
0	0.947099	0.980597	0.887179	0.859873	0.938879	0.898036	0.950764	0.906959	0.839422	0.900931
1	0.962478	0.979577	0.877414	0.874262	0.923736	0.890459	0.955954	0.917534	0.852787	0.913924
2	0.950997	0.977528	0.871219	0.867781	0.939523	0.881468	0.957663	0.915294	0.849408	0.877704
3	0.959149	0.985765	0.887188	0.876040	0.920455	0.890028	0.965577	0.908272	0.844652	0.904087
4	0.967374	0.978261	0.874062	0.854096	0.934690	0.904982	0.944492	0.908944	0.843911	0.909475

c)

Logistic regression class is extended to the One-vs-Rest approach. For n classes, n classifiers are needed where each classifier classifies between a class and all the rest classes. At last, the class with the highest positive probability is taken. OVR is relatively computationally efficient but suffers from class imbalance. Each sub logistic classifier standardizes that data & is run for 2000 epochs with lr=0.01 & lambda=0.01. Fit method of OVR logistic classifier:

```
def fit_ovr(self, _X, y, _X_val, y_val, epochs, display):
    if display:
        print('* Fitting OVR models')
    for c in range(self.num_classes):
        y_c = np.array([int(u) for u in (y == c)])
        y_val_c = np.array([int(u) for u in (y_val == c)])
        self.models[c] = LogRegression_Base()
        self.models[c].fit(_X, y_c, _X_val, y_val_c, epochs)
        if display:
            print(f'> Fitted model {c}')
            print(f'> Acc. train: {accuracy(y_c, self.models[c].predict(_X))}')
            if _X_val is not None:
                print(f'> Acc. val: {accuracy(y_val_c, self.models[c].predict(_X_val))}')
            print()
    if display:
        print('=========')
```

Predict method of OVR logistic classifier:

```
def predict_ovr(self, X):
    prob = []
    for c in range(self.num_classes):
        prob.append(self.models[c].predict_prob(X))
    prob_ = np.array(prob)
    y = np.array([np.argmax(prob_[:, i]) for i in range(X.shape[0])])
    return y
```

Accuracy table:

ovr_df

Validation fold	Train Acc.	Validation Acc.
0	0.860979	0.862250
1	0.861437	0.858583
2	0.862688	0.852833
3	0.861417	0.858417
4	0.860833	0.860333

Class-wise training accuracy for each fold:

ovr_classwise_train_df

	0	1	2	3	4	5	6	7	8	9
0	0.964008	0.972603	0.818922	0.836718	0.896012	0.743796	0.927004	0.884276	0.755028	0.785445
1	0.964203	0.972258	0.818334	0.840679	0.902032	0.744463	0.926257	0.885766	0.756113	0.778128
2	0.965035	0.973981	0.826911	0.838006	0.897815	0.749885	0.924214	0.886974	0.747483	0.788709
3	0.963142	0.971520	0.816833	0.839927	0.903576	0.748850	0.923886	0.886201	0.754547	0.780211
4	0.962529	0.973144	0.819710	0.842105	0.896848	0.743832	0.927744	0.886744	0.755390	0.775750

Class-wise validation accuracy for each fold:

ovr_classwise_val_df

	0	1	2	3	4	5	6	7	8	9
0	0.956485	0.974627	0.825641	0.839968	0.910866	0.756782	0.919355	0.881157	0.748513	0.784081
1	0.966841	0.972283	0.818640	0.851477	0.878320	0.723498	0.925775	0.887110	0.747387	0.787342
2	0.956811	0.971108	0.803803	0.827508	0.898637	0.734713	0.923793	0.883137	0.758037	0.752080
3	0.967660	0.975801	0.817889	0.838602	0.882867	0.739049	0.929432	0.881245	0.744482	0.778982
4	0.965742	0.968841	0.819850	0.837031	0.910941	0.740775	0.913749	0.884770	0.742710	0.785110

There are some classes with lower accuracies such as 5, 8, 3 (in both OVO and OVR) indicating the difficulty in classification for these classes.

d)

Comparison with sklearn implementation. Dataset is explicitly standardized before classification.

Using OneVsOneClassifier:

Model used: (Max iters=750 to ensure convergence)

```
ot ovo_sklearn_models_exists:
lr = OneVsOneClassifier(LogisticRegression(max_iter=750, random_state=7))
```

Accuracy table:

ovo_sklearn_df

	Validation fold	Train Acc.	Validation Acc.
0	0	0.980083	0.929667
1	1	0.979896	0.929667
2	2	0.980125	0.927000
3	3	0.978771	0.931167
4	4	0.980396	0.932000

Class Wise training accuracies for each fold:

ovo_sklearn_classwise_train_df

0	1	2	3	4	5	6	7	8	9
0.999369	0.997964	0.976608	0.957128	0.984991	0.970588	0.997679	0.980947	0.966196	0.966653
0.998953	0.998138	0.977764	0.961383	0.987380	0.965027	0.997229	0.979665	0.966192	0.964316
0.999788	0.998544	0.979588	0.959917	0.985433	0.966957	0.997678	0.980561	0.964018	0.965031
0.998526	0.996815	0.979012	0.960235	0.984674	0.965271	0.997267	0.977201	0.961481	0.964421
0.998723	0.998508	0.980878	0.961283	0.987776	0.967950	0.997261	0.981489	0.963074	0.964548

Class Wise validation accuracies for each fold:

ovo_sklearn_classwise_val_df

0	1	2	3	4	5	6	7	8	9
0.962457	0.973881	0.917949	0.897293	0.953311	0.900842	0.955008	0.935106	0.883602	0.910246
0.977312	0.975201	0.903442	0.904641	0.921165	0.887809	0.963295	0.940753	0.901568	0.911392
0.965947	0.975120	0.894555	0.909574	0.938671	0.888053	0.959356	0.933333	0.911168	0.888519
0.959149	0.986477	0.909750	0.915141	0.934441	0.900280	0.956971	0.938575	0.893039	0.906589
0.970636	0.976087	0.921601	0.889078	0.940628	0.901292	0.951324	0.941982	0.903945	0.911168

Sklearn's OVO implementation outperforms self implementation in training accuracy showcasing better convergence, although slightly better performance in validation showcasing similar generalizability of models.

Using OneVsRestClassifier:

Model used: (Max iters=750 to ensure convergence)

```
lr = OneVsRestClassifier(LogisticRegression(max_iter=750, random_state=7))
```

Accuracy table:

ovr_sklearn_df

Validation fold	Train Acc.	Validation Acc.
0	0.933729	0.913417
1	0.933208	0.915583
2	0.935458	0.907167
3	0.933417	0.912917
4	0.932979	0.917250

Class Wise training accuracies for each fold:

```
ovr_sklearn_classwise_train_df
```

0	1	2	3	4	5	6	7	8	9
0.984003	0.980933	0.917711	0.902974	0.941681	0.895450	0.966667	0.949057	0.885537	0.903314
0.981578	0.980823	0.919236	0.905580	0.948449	0.896246	0.963555	0.946172	0.883904	0.897145
0.984107	0.980167	0.920642	0.905088	0.946230	0.900872	0.966434	0.949098	0.884344	0.907310
0.983361	0.978640	0.917320	0.901400	0.947850	0.898114	0.965307	0.948255	0.883159	0.901684
0.980626	0.979299	0.919101	0.902400	0.945529	0.899470	0.967558	0.945860	0.883244	0.898678

Class Wise validation accuracies for each fold:

```
ovr_sklearn_classwise_val_df
```

0	1	2	3	4	5	6	7	8	9
0.966724	0.973881	0.892308	0.872611	0.940577	0.884004	0.950764	0.921032	0.846219	0.877223
0.974695	0.964989	0.883291	0.895359	0.916881	0.857774	0.957586	0.928743	0.866725	0.897890
0.964286	0.967095	0.876404	0.869301	0.928450	0.863594	0.947502	0.928627	0.862098	0.858569
0.963404	0.978648	0.893634	0.894343	0.913462	0.870457	0.956110	0.922195	0.855688	0.867389
0.977162	0.971739	0.893244	0.884812	0.932994	0.872694	0.946200	0.928284	0.858491	0.890863

Sklearn’s OVR implementation outperforms self implementation both in training and validation accuracies indicating better convergence and generalizability. This difference arises due to differences in hyperparameters, convergence algorithms, etc.