

INTRODUCTION

In this project I:

- Implemented multiclass logistic regression with regularization,
- Used Principal Component Analysis or PCA to reduce the feature dimensions,
- Used feature transform.
- Used digits dataset from sklearn.

PRINCIPAL COMPONENT ANALYSIS

Principal component analysis (PCA) is the process of computing the principal components and using them to perform a change of basis on the data, sometimes using only the first few principal components and ignoring the rest. As we know when we do SVD analysis of matrices we can see that some of the eigen values have higher importance, in control theory they call them dominant poles. Dominant poles are those poles that are closer to the imaginary axis. They describe most of the dynamics of the system. PCA basically finds the axis' that describe the system the best, or the eigen vectors in which most of the information lies. An example taken from Wikipedia is shown in figure 1. We can see the eigen vectors that describe the data the best. There are few steps to calculate PCA in data. First we compute the mean in every dimension, then compute the covariance matrix.

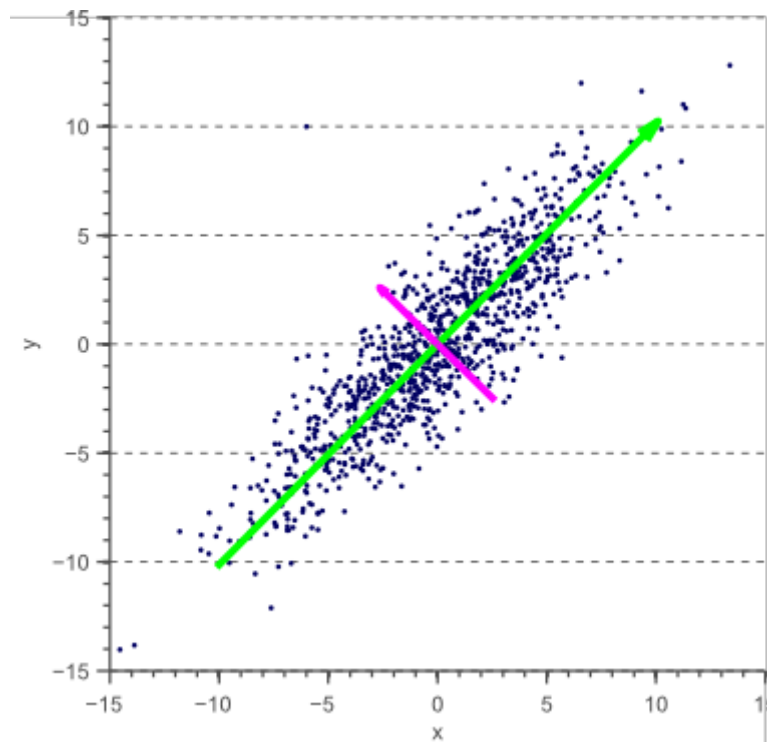


Figure 1. Two eigen vectors scaled and mean-centered that are good representation for the data shown.

After we compute covariance matrix, then we compute eigenvectors and corresponding eigen values. Next step is sorting the eigen vectors by corresponding eigen values. Finally, we can take as much components as we want and we can form a matrix out of those eigen vectors and transform those samples onto a new subspace. All these sentences are represented in the Choosing K Algorithm shown in figure 2. Where we pick the smallest value of K for which 99% of the variance is retained.

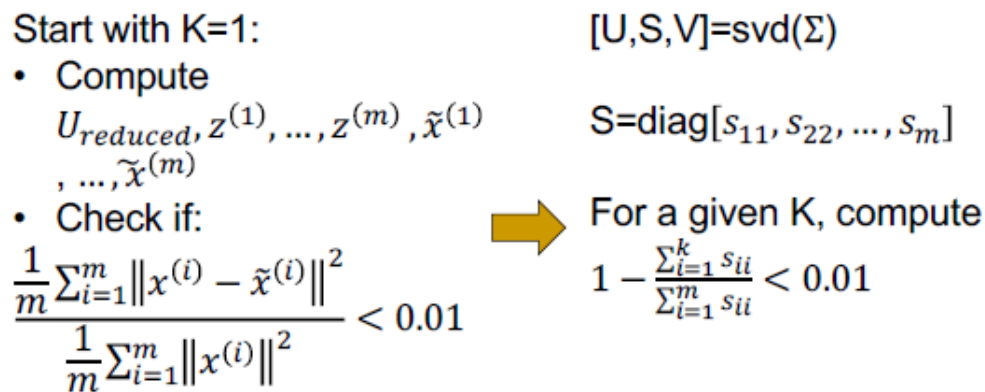


Figure 2. Choosing K: Algorithm, from Prof. Tsai's slides.

In our case it was required to compute three features, but when I computed PCA and selected corresponding eigen vectors they represented only 40.3% of the digits' data. And that was not nearly enough to have some good results on classification. So I increased the number of components to 10, so they represent the majority information of the dataset (over 75%), and that was enough to achieve pretty good accuracy on classification.

All this is achieved with few lines of code using sklearn built-in functions for calculating PCA:

```
pca = decomposition.PCA(n_components=10)
pca.fit(X_digits)
X_pca = pca.transform(X_digits)
print("Preserved amount of the information [{0}%]".format(np.sum(pca.explained_variance_ratio_)))
```

FEATURE TRANSFORM

In order to transform from a space to another we use transformation. After discussion with Prof. Tsai I decided to use 5th order transformation. It is impossible to use the 20th order. The reason is that if you transform 10 features using 20th order polynomial transform then you would need around 430Gb of memory to store all that information, if each feature is of type float. To do the feature transformation, same as PCA I used sklearn built-in functions:

```
phi_20 = PolynomialFeatures(5)
X = phi_20.fit_transform(X_pca)
```

LOGISTIC REGRESSION WITH REGULARIZATION

This algorithm is a soft binary classification algorithm, which will return probability instead 0/1 classes. This makes this algorithm very useful for multi-class classification. When we use this algorithm for prediction we look for the highest probability instead the class, that's how we are able to classify an input. Logistic regression model is given by:

$$h(x) = \theta(w^T x)$$

where:

$$\theta(x) = \frac{e^x}{1 + e^x}$$

which gives an output between 0 and 1. This logistic function θ is a soft threshold, that's why it is very helpful because it gives probabilities.

Formally, we are trying to learn the target function $f(x) = P[y = +1|x]$. The data we have gives us samples of this f function. The data is generated by a noisy target $P(y|x)$:

$$P(y|x) = \begin{cases} f(x) & \text{for } y = +1 \\ 1 - f(x) & \text{for } y = -1 \end{cases}$$

To learn from data we need to define an error function. The standard error measure used in logistic regression is based on the notion of likelihood:

$$P(y|x) = \theta(y w^T x)$$

In this work we decided to use **cross-entropy** with **L2 regularization** loss function which is given with:

$$J(\theta) = \frac{1}{N} \sum_{i=1}^N \left[-y^i \log(h_{\theta}(x^i)) - (1 - y^i) \log(1 - h_{\theta}(x^i)) \right] + \frac{\lambda}{2N} \sum_{j=1}^N \omega_j^2$$

To find the optimal solution I used the gradient descent. From loss function we calculate the gradient which is given with:

$$\frac{J(\theta)}{\partial \theta} = \frac{1}{N} \sum_{i=1}^N [(h_{\theta}(x^i) - y^i) x^i + \lambda \omega_i]$$

and then weight update is given with:

$$\theta \leftarrow \theta - \alpha \frac{J(\theta)}{\partial \theta}$$

where α is the learning rate, and λ regularization coefficient. Larger values of λ will cause under fitting, and if value of $\lambda=0$ we will have no regularization at all.

To classify multi-classes there are few strategies. I used One Vs All strategy. This strategy will take a label at the time, and make the other labels as one label, this helps because after you separate the data in that form, you can simply use binary classification. When we want to predict, we will have a set of weights, in order to know the output we need to use all generated weights and choose the output with the highest probability. The implementation of Logistic Regression is shown below:

```
class LogRgrsReg(object):

    def __init__(self):
        None

    def sigmoid(self, z):
        return 1/(1+np.exp(-z))

    def hypothesis(self, X, W):
        z = np.dot(X, W)
        return self.sigmoid(z)

    def cost_function(self, X, W, Y, lambda):
        predictions = self.sigmoid(X @ W)
        predictions[predictions == 1] = 0.9999999 # log(1) = 0
        gradient = (X.T @ (predictions - Y)) / len(Y) + lambda*np.sum(W[1:-
1]) / len(Y)
        return gradient

    def fit_model(self, X, W, Y, learn_rate=0.05, lambda=0.001, epochs=10):
        W_model = [] # save the weights for each class in this array
        classes = np.unique(Y)
        for c_class in classes: # iterate through all unique classes in Y
            Y_c = np.where(Y == c_class, 1, 0) #
            W_i= copy.deepcopy(W) # Always use this as init weight
            for epoch in range(epochs):
                gradient = self.cost_function(X, W_i, Y_c, lambda)
                W_i = W_i - learn_rate*gradient
            W_model.append(W_i)
        return W_model

    def predict_one_vs_all(self, X, W):
        max_probability = 0.0
        c_class = 99999
        for i in range(len(W)):
```

```

        curr_prob = self.hypothesis(X, W[i])
        if max_probability < curr_prob:
            max_probability = curr_prob
            c_class = i
    return c_class

def check_accuracy(self, X, W, Y):
    accuracy = 0.0
    tot_hits = 0
    for i in range(len(X)):
        if self.predict_one_vs_all(X[i], W) == Y[i]:
            tot_hits += 1
    accuracy = tot_hits/len(Y)
    # print("Accuracy: {0}%".format(accuracy*100.0))
    return accuracy

```

EXPERIMENTS AND RESULTS

The requirement of this work was doing 5-fold cross-validation. The V-fold cross validation is a validation technique.

After I did PCA and transformation, then I combined labels and feature data in one matrix. I did that because V-fold cross-validation requires that you first shuffle the data. I did that in these lines of code:

```

# Add label with the corresponding row for shuffling
X = np.hstack((X, Y))
np.random.shuffle(X)    # Shuffle data

```

In this work we were asked to do 5-fold cross-validation. That's why I separated the data in 5 equal parts:

```

# Generate 5-Fold training and testing...
X_p1 = copy.deepcopy(X[0:359, :-1])
X_p2 = copy.deepcopy(X[359:718, :-1])
X_p3 = copy.deepcopy(X[718:1077, :-1])
X_p4 = copy.deepcopy(X[1077:1436, :-1])
X_p5 = copy.deepcopy(X[1436:1795, :-1])

Y_p1 = copy.deepcopy(X[0:359, -1])
Y_p2 = copy.deepcopy(X[359:718, -1])
Y_p3 = copy.deepcopy(X[718:1077, -1])
Y_p4 = copy.deepcopy(X[1077:1436, -1])
Y_p5 = copy.deepcopy(X[1436:1795, -1])

```

Then combined the data for all possible cases:

```
# leave 1st out
X_train1 = np.concatenate((X_p2, X_p3, X_p4, X_p5), axis=0)
Y_train1 = np.concatenate((Y_p2, Y_p3, Y_p4, Y_p5), axis=None)
X_test1 = copy.deepcopy(X_p1)
Y_test1 = copy.deepcopy(Y_p1)

# leave 2nd out
X_train2 = np.concatenate((X_p1, X_p3, X_p4, X_p5), axis=0)
Y_train2 = np.concatenate((Y_p1, Y_p3, Y_p4, Y_p5), axis=None)
X_test2 = copy.deepcopy(X_p2)
Y_test2 = copy.deepcopy(Y_p2)

# leave 3rd out
X_train3 = np.concatenate((X_p1, X_p2, X_p4, X_p5), axis=0)
Y_train3 = np.concatenate((Y_p1, Y_p2, Y_p4, Y_p5), axis=None)
X_test3 = copy.deepcopy(X_p3)
Y_test3 = copy.deepcopy(Y_p3)

# leave 4th out
X_train4 = np.concatenate((X_p1, X_p2, X_p3, X_p5), axis=0)
Y_train4 = np.concatenate((Y_p1, Y_p2, Y_p3, Y_p5), axis=None)
X_test4 = copy.deepcopy(X_p4)
Y_test4 = copy.deepcopy(Y_p4)

# leave 5th out
X_train5 = np.concatenate((X_p1, X_p2, X_p3, X_p4), axis=0)
Y_train5 = np.concatenate((Y_p1, Y_p2, Y_p3, Y_p4), axis=None)
X_test5 = copy.deepcopy(X_p5)
Y_test5 = copy.deepcopy(Y_p5)
```

Finally, I used a for loop to train data for different regularization coefficients and also calculate:

$$E_{cv}(H, A) = \frac{1}{V} \sum_{v=1}^V E_{val}^{(v)}(g_v^-)$$

I also keep track of the E_{in} and E_{out} :

```
for i in range(experiments):
    w_init = X_train1[0]
    W_models_1 = logRegression.fit_model(X_train1, w_init, Y_train1, learn_rate=1
, lambda=lambda_x, epochs=100)
    w_init = X_train2[0]
    W_models_2 = logRegression.fit_model(X_train2, w_init, Y_train2, learn_rate=1
, lambda=lambda_x, epochs=100)
    w_init = X_train3[0]
```

```

W_models_3 = logRegression.fit_model(X_train3, w_init, Y_train3, learn_rate=1
, lambda=lambda_x, epochs=100)
w_init = X_train4[0]
W_models_4 = logRegression.fit_model(X_train4, w_init, Y_train4, learn_rate=1
, lambda=lambda_x, epochs=100)
w_init = X_train5[0]
W_models_5 = logRegression.fit_model(X_train5, w_init, Y_train5, learn_rate=1
, lambda=lambda_x, epochs=100)

E_cv_min = 1.0
max_index = 1

### 5 fold validation:
#####
tr_1 = 1.0-logRegression.check_accuracy(X_train1, W_models_1, Y_train1)
ts_1 = 1.0-logRegression.check_accuracy(X_test1, W_models_1, Y_test1)
# E_cv 1:
eval1_1 = 1.0-logRegression.check_accuracy(X_p1, W_models_1, Y_p1)
eval1_2 = 1.0-logRegression.check_accuracy(X_p2, W_models_1, Y_p2)
eval1_3 = 1.0-logRegression.check_accuracy(X_p3, W_models_1, Y_p3)
eval1_4 = 1.0-logRegression.check_accuracy(X_p4, W_models_1, Y_p4)
eval1_5 = 1.0-logRegression.check_accuracy(X_p5, W_models_1, Y_p5)
E_cv1 = float(1.0/5.0)*(eval1_1+eval1_2+eval1_3+eval1_4+eval1_5)
E_cv_min = E_cv1
#####
tr_2 = 1.0-logRegression.check_accuracy(X_train2, W_models_2, Y_train2)
ts_2 = 1.0-logRegression.check_accuracy(X_test2, W_models_2, Y_test2)
# E_cv 2:
eval2_1 = 1.0-logRegression.check_accuracy(X_p1, W_models_2, Y_p1)
eval2_2 = 1.0-logRegression.check_accuracy(X_p2, W_models_2, Y_p2)
eval2_3 = 1.0-logRegression.check_accuracy(X_p3, W_models_2, Y_p3)
eval2_4 = 1.0-logRegression.check_accuracy(X_p4, W_models_2, Y_p4)
eval2_5 = 1.0-logRegression.check_accuracy(X_p5, W_models_2, Y_p5)
E_cv2 = float(1.0/5.0)*(eval2_1+eval2_2+eval2_3+eval2_4+eval2_5)
if E_cv_min > E_cv2:
    E_cv_min = E_cv2
    max_index = 2
#####
tr_3 = 1.0-logRegression.check_accuracy(X_train3, W_models_3, Y_train3)
ts_3 = 1.0-logRegression.check_accuracy(X_test3, W_models_3, Y_test3)
# E_cv 3:
eval3_1 = 1.0-logRegression.check_accuracy(X_p1, W_models_3, Y_p1)
eval3_2 = 1.0-logRegression.check_accuracy(X_p2, W_models_3, Y_p2)
eval3_3 = 1.0-logRegression.check_accuracy(X_p3, W_models_3, Y_p3)
eval3_4 = 1.0-logRegression.check_accuracy(X_p4, W_models_3, Y_p4)

```

```

eval3_5 = 1.0-logRegression.check_accuracy(X_p5, W_models_3, Y_p5)
E_cv3 = float(1.0/5.0)*(eval3_1+eval3_2+eval3_3+eval3_4+eval3_5)
if E_cv_min > E_cv3:
    E_cv_min = E_cv3
    max_index = 3

#####
tr_4 = 1.0-logRegression.check_accuracy(X_train4, W_models_4, Y_train4)
ts_4 = 1.0-logRegression.check_accuracy(X_test4, W_models_4, Y_test4)
# E_cv 4:
eval4_1 = 1.0-logRegression.check_accuracy(X_p1, W_models_4, Y_p1)
eval4_2 = 1.0-logRegression.check_accuracy(X_p2, W_models_4, Y_p2)
eval4_3 = 1.0-logRegression.check_accuracy(X_p3, W_models_4, Y_p3)
eval4_4 = 1.0-logRegression.check_accuracy(X_p4, W_models_4, Y_p4)
eval4_5 = 1.0-logRegression.check_accuracy(X_p5, W_models_4, Y_p5)
E_cv4 = float(1.0/5.0)*(eval4_1+eval4_2+eval4_3+eval4_4+eval4_5)
if E_cv_min > E_cv4:
    E_cv_min = E_cv4
    max_index = 4

#####
tr_5 = 1.0-logRegression.check_accuracy(X_train5, W_models_5, Y_train5)
ts_5 = 1.0-logRegression.check_accuracy(X_test5, W_models_5, Y_test5)
# E_cv 5:
eval5_1 = 1.0-logRegression.check_accuracy(X_p1, W_models_5, Y_p1)
eval5_2 = 1.0-logRegression.check_accuracy(X_p2, W_models_5, Y_p2)
eval5_3 = 1.0-logRegression.check_accuracy(X_p3, W_models_5, Y_p3)
eval5_4 = 1.0-logRegression.check_accuracy(X_p4, W_models_5, Y_p4)
eval5_5 = 1.0-logRegression.check_accuracy(X_p5, W_models_5, Y_p5)
E_cv5 = float(1.0/5.0)*(eval5_1+eval5_2+eval5_3+eval5_4+eval5_5)
if E_cv_min > E_cv5:
    E_cv_min = E_cv5
    max_index = 5

##### Print best E_cv #####
print("Lambda: {0}; E_cv: {1}; fold-
out: {2}".format(lambda_x, E_cv_min, max_index))

E_in[i] = np.array([lambda_x, tr_1, tr_2, tr_3, tr_4, tr_5])
E_out[i] = np.array([lambda_x, ts_1, ts_2, ts_3, ts_4, ts_5])
E_cv[i] = np.array([lambda_x, E_cv1, E_cv2, E_cv3, E_cv4, E_cv5])
lambda_x += 0.01

```

EXPERIMENT 1: iterations =100, learning rate = 1

After training and testing for each lambda these are the best E_{cv} :


```

Lambda: 0.9; E_cv: 0.0668523676880223; fold-out: 1
Lambda: 0.91; E_cv: 0.07130919220055712; fold-out: 1
Lambda: 0.92; E_cv: 0.07298050139275768; fold-out: 1
Lambda: 0.93; E_cv: 0.0668523676880223; fold-out: 1
Lambda: 0.9400000000000001; E_cv: 0.07019498607242339; fold-out: 1
Lambda: 0.9500000000000001; E_cv: 0.06852367688022283; fold-out: 1
Lambda: 0.9600000000000001; E_cv: 0.10362116991643454; fold-out: 1
Lambda: 0.9700000000000001; E_cv: 0.23231197771587742; fold-out: 3

```

It seems that leaving 1st part out gives the best E_{cv} for $\lambda=0.9$ and learning rate 1. As we can see here with the increasing of λ we are going towards under-fitting.

In the figures below we can see plots of E_{val} , E_{cv} and E_{in} .

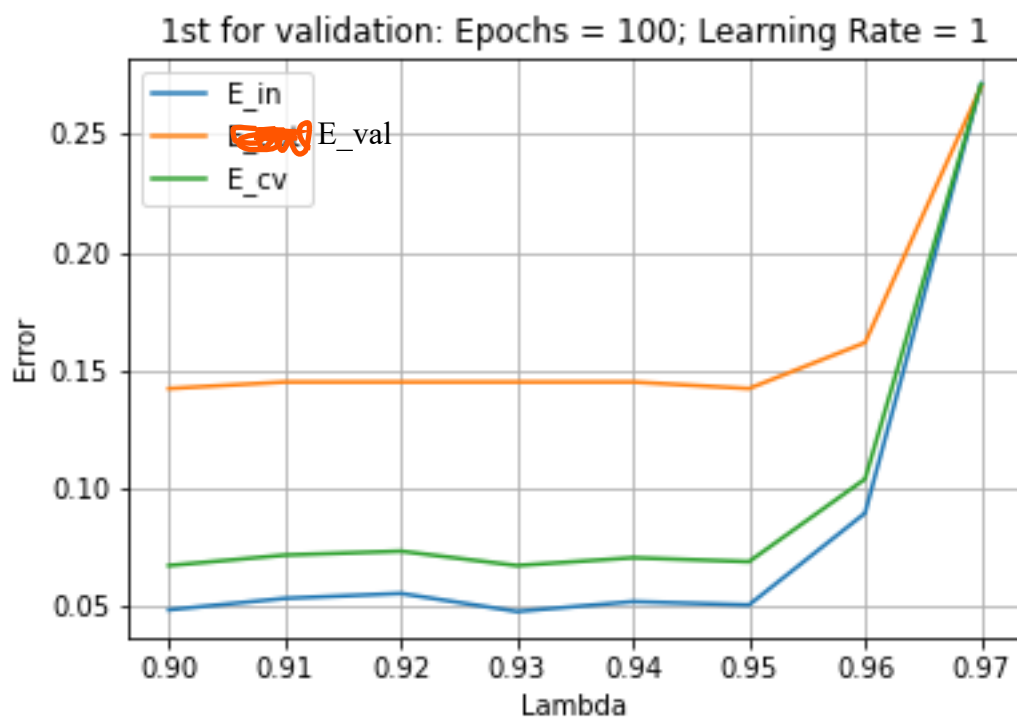


Figure 3. Errors for the 1st portion of data left out.

In figure 3, we can see that E_{cv} and E_{in} are pretty close to each other with the current settings, but we can see that with the increasing of regularization factor we cause under-fitting. We have to mention that learning rate is high.

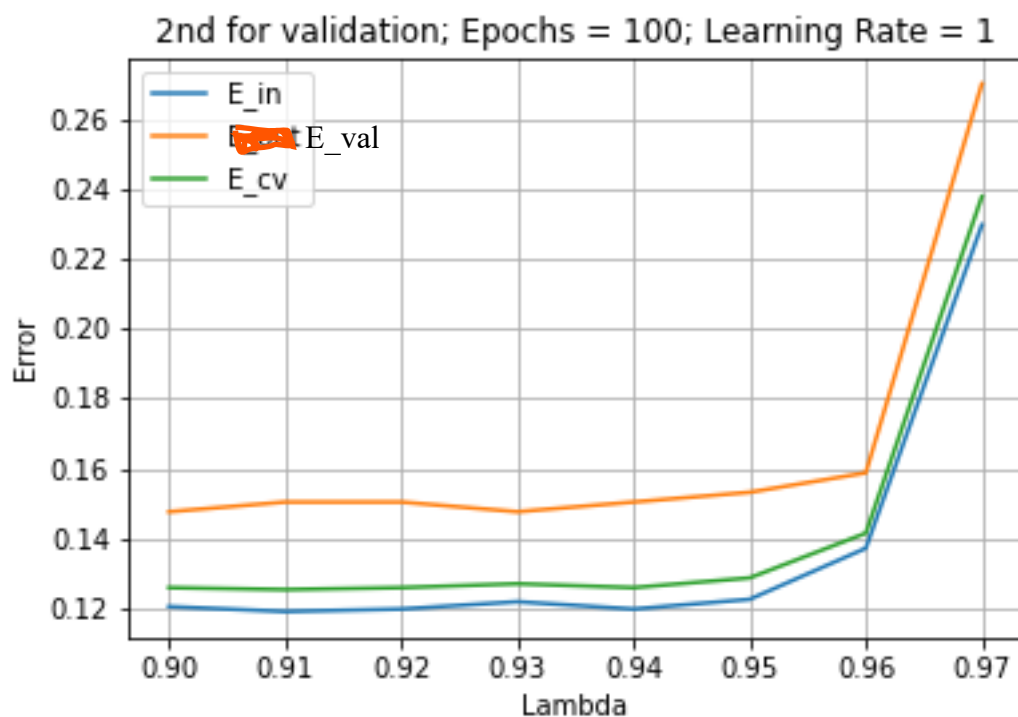


Figure 4. Error plots for the 2nd portion of data left out.

As we can see we have the same case as when we left 1st portion out. Accuracy is lower than the first case. We caused under-fitting with the increased value of regularization factor.

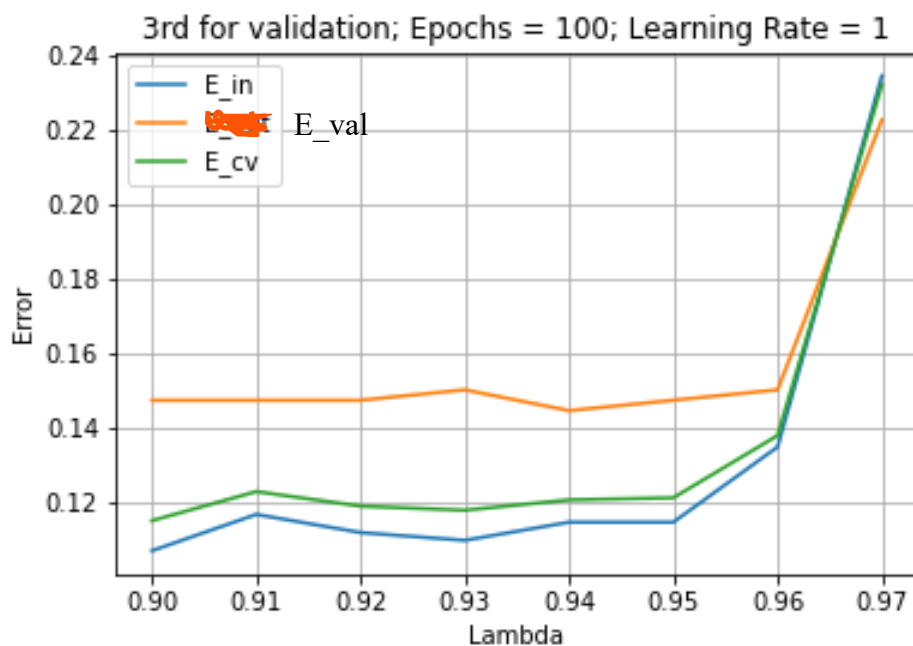


Figure 5. Error plots for the 3rd portion of data left out.

Leaving 3rd portion of data out for validation caused a smaller E_{cv} and E_{in} than the previous case. As we can see as soon as we increase lambda to 0.96 we cause the E_{cv} and E_{in} to increase but E_{val} is still not effected much. We can see that when we set lambda to 0.97 we already have E_{cv} and E_{in} error rates over 22%.

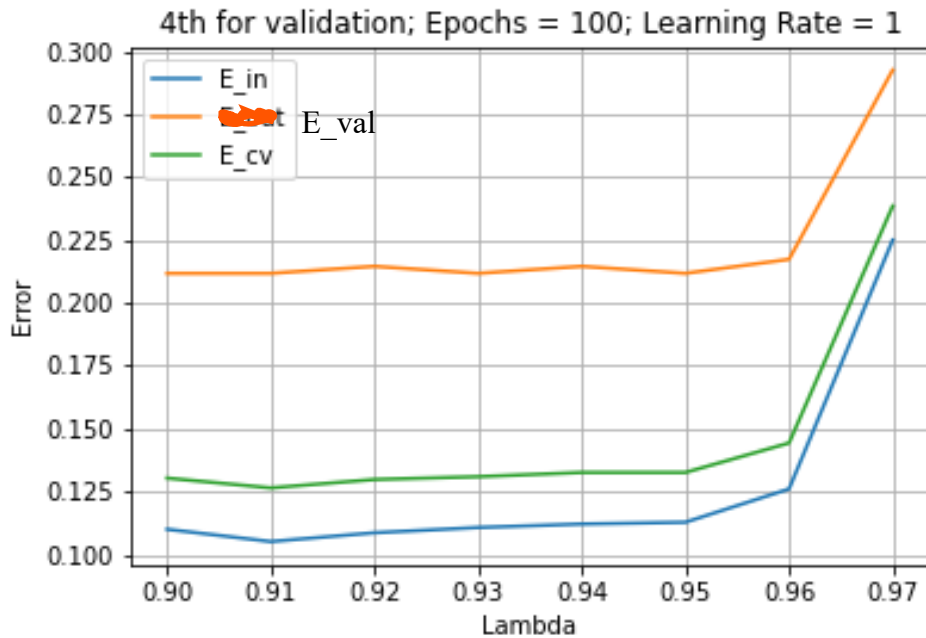


Figure 6. Error plots for the 4th portion of data left out for validation.

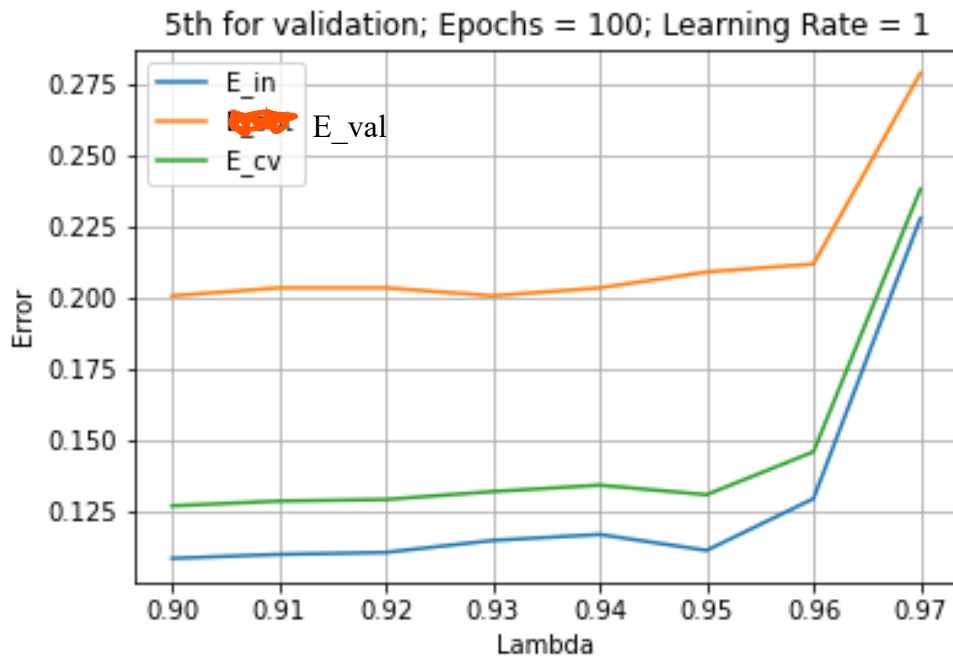


Figure 7. Error plots for the 5th portion of data left out for validation.

As we can see from figure 6 and 7, leaving 4th and 5th portions out for validation is not a good idea, they contain a lot of information and as we can see E_{val} is very high, over 20%. Also the effect of regularization factor is the same as in the other cases.

EXPERIMENT 2: iterations =500, learning rate = 0.5

In this experiment I re-shuffled the data. I use different learning rate and more iterations. For each λ I calculated E_{cv} and the lowest E_{cv} is achieved with $\lambda=1.912$:

```
Lambda: 1.91; E_cv: 0.03899721448467968; fold-out: 2
Lambda: 1.912; E_cv: 0.03788300835654599; fold-out: 2
Lambda: 1.914; E_cv: 0.04178272980501392; fold-out: 2
Lambda: 1.916; E_cv: 0.06629526462395544; fold-out: 2
Lambda: 1.918; E_cv: 0.12534818941504178; fold-out: 2
Lambda: 1.92; E_cv: 0.21838440111420612; fold-out: 3
Lambda: 1.922; E_cv: 0.32367688022284125; fold-out: 3
Lambda: 1.924; E_cv: 0.4378830083565459; fold-out: 3
```

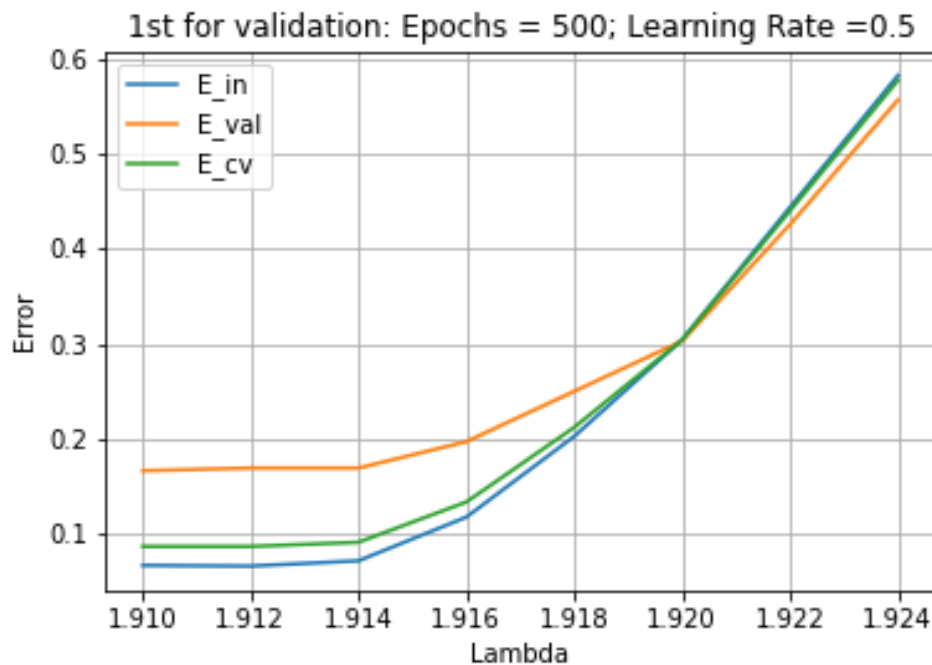


Figure 8. Error plots for the 1th portion of data left out for validation.

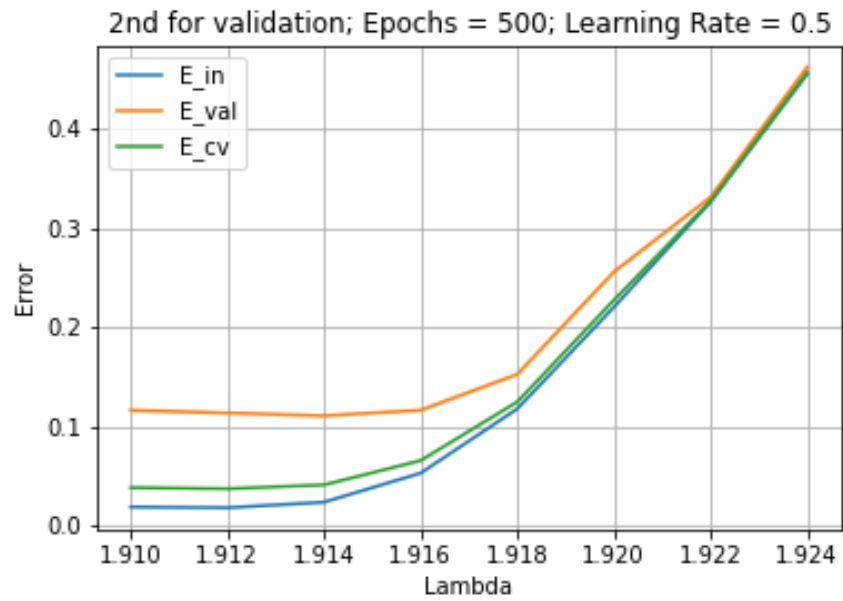


Figure 9. Error plots for the 2th portion of data left out for validation.

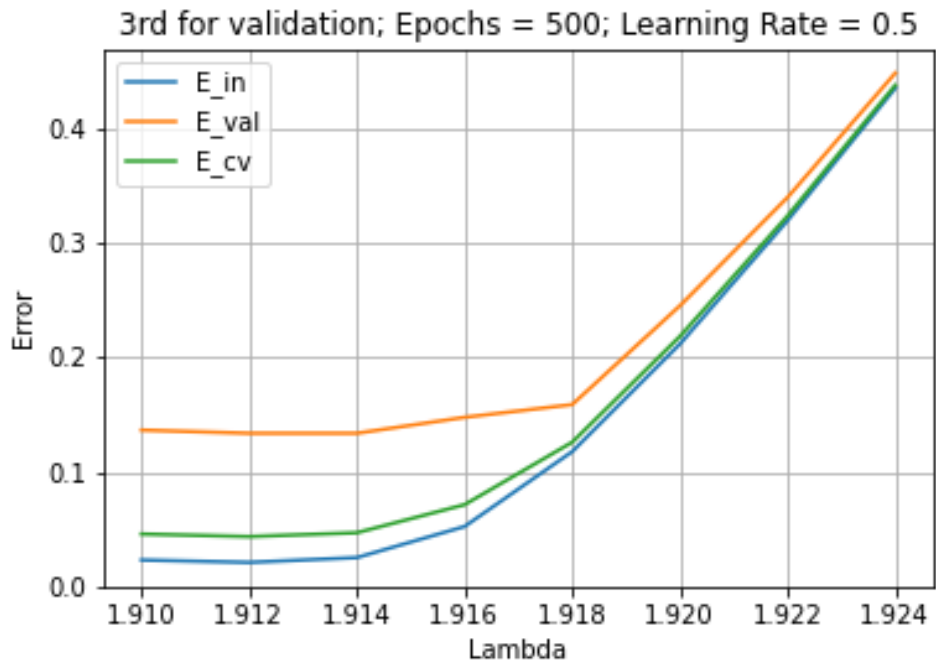


Figure 10. Error plots for the 3th portion of data left out for validation.

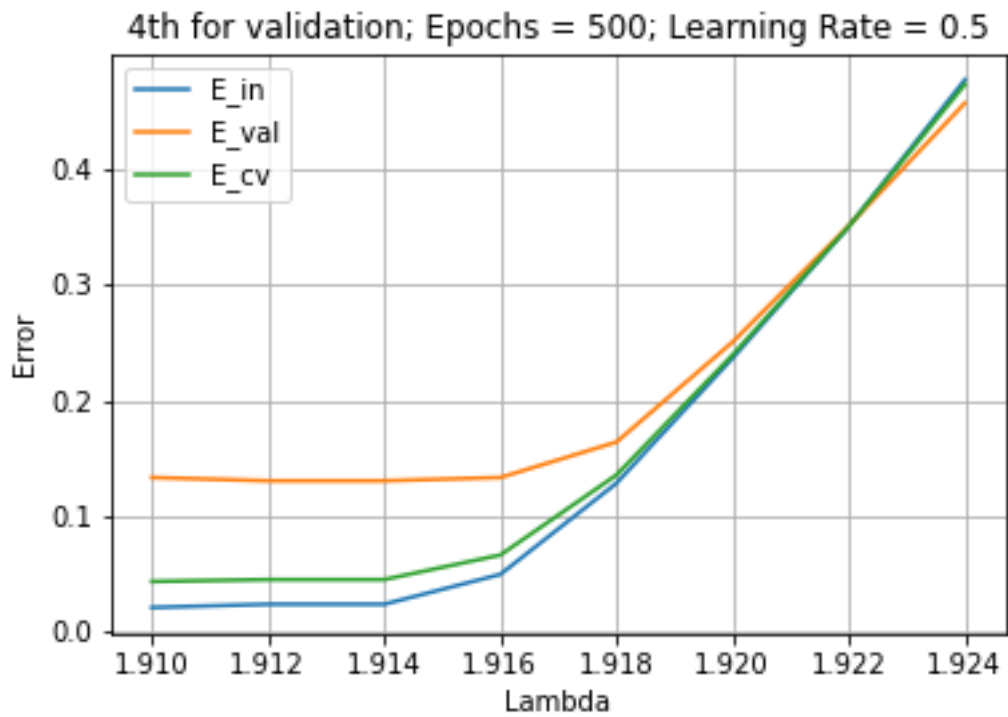


Figure 11. Error plots for the 4th portion of data left out for validation.

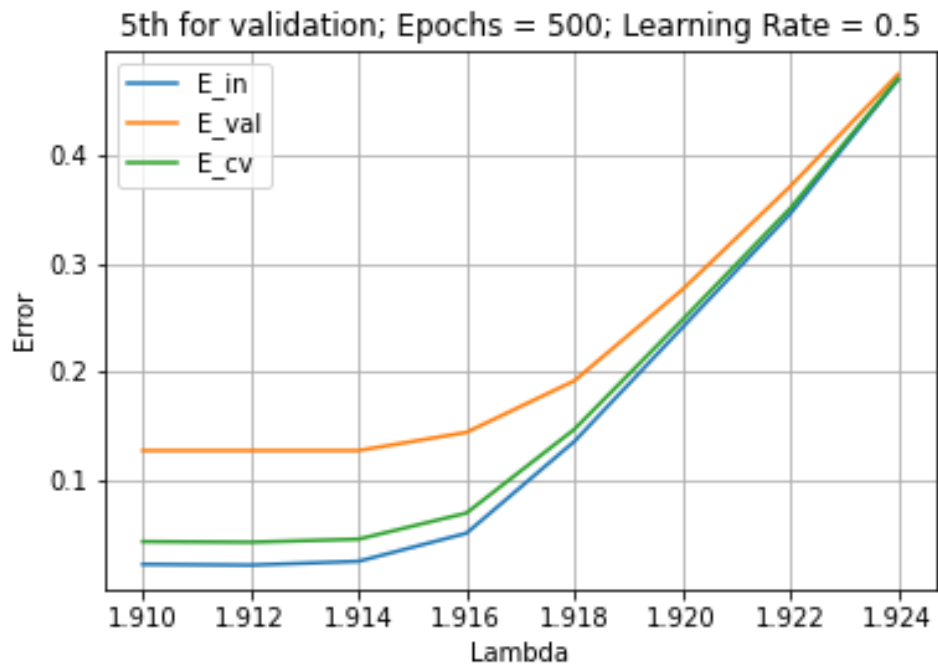



Figure 12. Error plots for the 5th portion of data left out for validation.

As we can see from the figures above, we did not encounter over-fitting and the reason is for using only 5th degree of polynomial feature transformation. However, the effect of regularization can be seen here too. As we can see when we increase λ we cause under-fitting, because we 'lower' the degree of the model. The accuracy according to 5-fold cross-validation for this setup is very high, around 3% E_{cv} . As expected when we suppress high order terms of the model, using regularization with a high value of λ all errors will increase, but they also tend to become very close to each other. Regularization tries to keep the weight w as small as possible, this directly causes 'dimension reduction', and this means that even though the model is a high order model, we force many coefficients to be small and this leads to a behaviour of a lower polynomial degree.

 11/11/2020