Aabir_AK_HW03

February 6, 2020

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error as MSE
import tabulate as tb

import seaborn as sns

sns.set_style("whitegrid")
```

[2]:
```
np.random.seed(42)
```

0.1 Conceptual exercises

0.1.1 Generating data (X) and computing Y with noise

[3]:
```
X = np.random.normal(loc=np.random.randint(0, 100, 20), scale=np.random.
↪randint(0, 25, 20), size=(1000, 20))
```

[4]:
```
beta = np.random.normal(0, 30, 20)
for i in range(beta.shape[0]):
    if np.random.random() >= 0.70:
        beta[i] = 0

beta = np.array(beta)

assert any([i==0 for i in beta.T]), "beta contains no zeros"
zero_pos = [i for i in range(len(beta)) if beta[i]==0]
print(f"zero values for beta at positions: {zero_pos}")

epsilon = np.random.randn(1000,)

print(X.shape, beta.shape, epsilon.shape)
```

```
zero values for beta at positions: [6, 10, 17, 19]
(1000, 20) (20,) (1000,)
```

[5]:
```
Y = np.matmul(X, beta) + epsilon
print(Y.shape)
```

```
(1000,)
```

0.1.2 Splitting the data

[6]:
```
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, train_size=100)
print(X_train.shape, X_test.shape, Y_train.shape, Y_test.shape)
```

```
(100, 20) (900, 20) (100,) (900,)
```

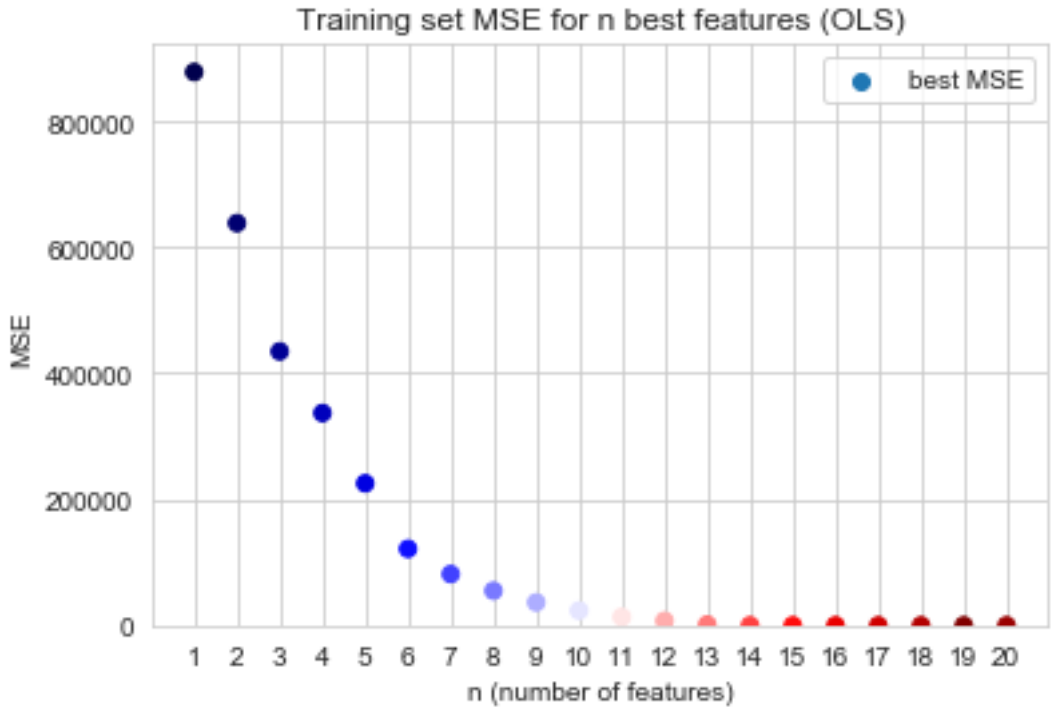
0.1.3 Performing best subset selection and compiling results on *training* set

[7]:
```
# I analyze results of the forward selection algorithm. For backward selection,␣
 ↪use forward = False

forward = True

best_featureset = [] if forward else list(range(20))
best_MSE = []
best_model = []
model_size = []
for select_step in range(1, 21):
    s = select_step if forward else 21-select_step
    model_size.append(s)
    #print(f'best_featureset : {best_featureset}')
    new_mse = []
    all_models = []
    for feature_no in range(20):
        if forward:
            if feature_no in best_featureset:
                new_mse.append(np.inf)
                all_models.append(None)
            else:
                #print(f"{feature_no} not in {best_featureset}")
                f_list = best_featureset+[feature_no]
                mdl = LinearRegression(fit_intercept=True).fit(X_train[:,␣
 ↪f_list], Y_train)
                Y_train_pred = mdl.predict(X_train[:, f_list])
                new_mse.append(MSE(Y_train, Y_train_pred))
                all_models.append(mdl)
        else:
            if feature_no in best_featureset:
                f_list = best_featureset.copy()
```

```
                f_list.remove(feature_no)
                mdl = LinearRegression(fit_intercept=True).fit(X_train[:,␣
↪f_list], Y_train)
                Y_train_pred = mdl.predict(X_train[:, f_list])
                new_mse.append(MSE(Y_train, Y_train_pred))
                all_models.append(mdl)
            else:
                new_mse.append(np.inf)
                all_models.append(None)
    #print(new_mse)
    if forward:
        best_feat = np.argmin(new_mse)
        #print(f'adding new best feature # {best_feat}')
        best_featureset.append(best_feat)
        best_MSE.append(np.min(new_mse))
        best_model.append(all_models[best_feat])
    else:
        worst_feat = np.argmax(new_mse)
        best_featureset.pop(worst_feat)
        best_MSE.append(np.min(new_mse))
    #print(f"best subset of size {model_size}: {best_featureset}\n\tMSE =␣
↪{best_MSE[-1]}")

#assert all([MSE(Y_train, best_model[i].predict(X_train))== best_MSE[i] for i␣
↪in range(20)])
```

[8]:
```
plt.clf()
ranks = np.argsort(best_MSE)
colnums = list(reversed(np.linspace(0.3, 0.99, 20)))
cols = np.zeros(20)
for i, r in enumerate(ranks):
    cols[r] = colnums[i]
plt.scatter(range(1, 21), best_MSE, label='best MSE', c=cols, cmap=plt.cm.
↪seismic)
plt.title('Training set MSE for n best features (OLS)')
plt.xticks(range(1, 21))
plt.xlabel('n (number of features)')
plt.ylabel('MSE')
plt.ylim(0)
plt.legend()
plt.show()
print(f'Train MSE is lowest for the best {ranks[0]+1} features.\n\n')
print(tb.tabulate(dict(zip(['rank', 'feature #', 'beta weight', 'MSE',
                            'L2 of coefs', #'abs-sum of betas',
                            'L2 b/w coefs and betas'
                           ],
```

```
                            [list(range(1, 21)), [f"f{i+1}" for i in␣
↪best_featureset],
                                 beta[best_featureset], best_MSE,
                                 [np.linalg.norm(mdl.coef_) for mdl in␣
↪best_model],
                                 #[np.abs(beta[best_featureset][:i+1]).sum() for␣
↪i in range(len(beta))]
                                 [np.linalg.norm(beta[:i+1] - mdl.coef_) for i,␣
↪mdl in enumerate(best_model)]
                            ]), headers="keys"))
```



Train MSE is lowest for the best 19 features.

rank	feature #	beta weight	MSE	L2 of coefs	L2 b/w coefs and betas
1	f5	24.4879	878367	26.3608	17.9478
2	f16	27.3752	638503	40.9984	24.4321
3	f4	-38.9013	434847	54.946	28.2849

```
     4  f12              -25.2876   336982              58.0339
31.1381
     5  f2                17.5056   225363              63.318
30.9881
     6  f6                20.2686   121443              70.5721
30.8221
     7  f13               11.5036    81299.5            69.2825
31.2721
     8  f8                11.2583    54823              70.4401
31.6916
     9  f19              -35.6605    36339.9            77.4138
39.2698
    10  f9               -11.9184    23392.2            76.9538
38.3716
    11  f17               -3.0314    13116.9            77.584
39.3817
    12  f15               -4.06156    7129.68           77.3908
46.029
    13  f10               -5.3283     1275.6            76.4245
48.296
    14  f14                3.57638      62.4206         77.3582
48.8478
    15  f1                 8.41296       0.622241       77.6478
50.3792
    16  f18                0             0.576682       77.6476
57.2873
    17  f11                0             0.573091       77.6495
57.3696
    18  f20                0             0.569609       77.6504
57.3727
    19  f7                 0             0.569525       77.6504
67.5517
    20  f3               -21.3005        0.569525       77.6504
67.5517
```

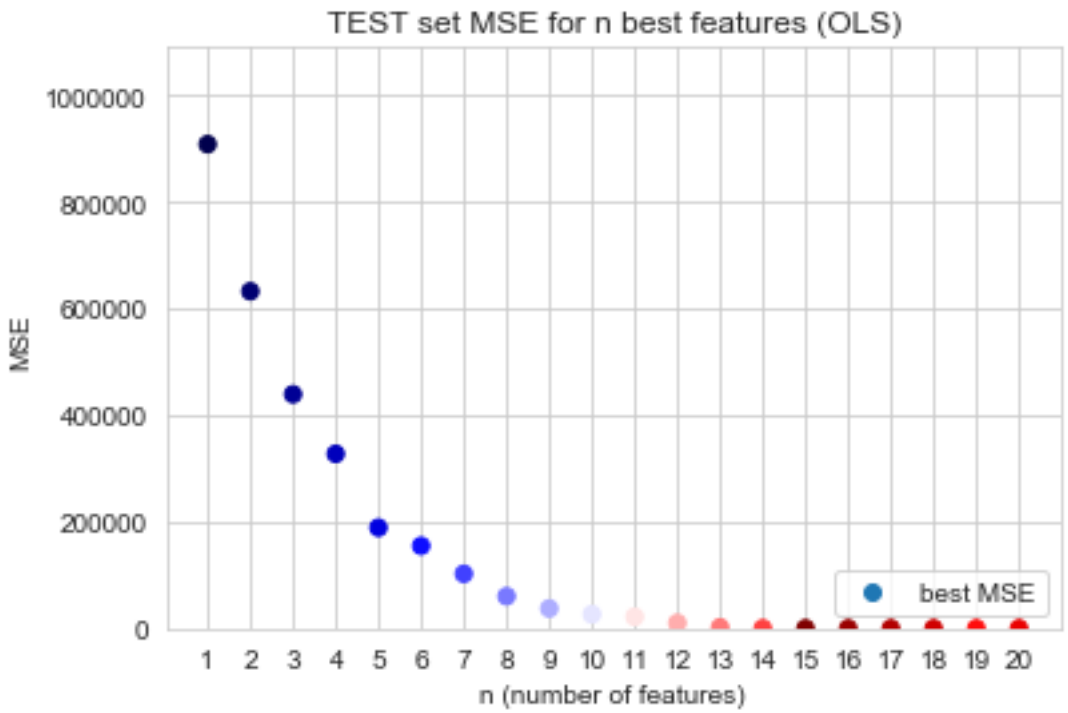
As the variance of each feature is quite large, and the beta values are large too, the MSE decreases rapidly until the 16th feature is added. The f16 through f19 are those for which $\beta = 0$, which intuitively makes sense.

It is a little unusual that f3 (with a relatively large magnitude β is the 'worst' performing feature by this method.

0.1.4 Compiling results on *test* set

[9]:
```
plt.clf()
test_MSEs = [MSE(Y_test, mdl.predict(X_test[:, best_featureset[:i+1]])) for i,␣
 ↪mdl in enumerate(best_model)]
```

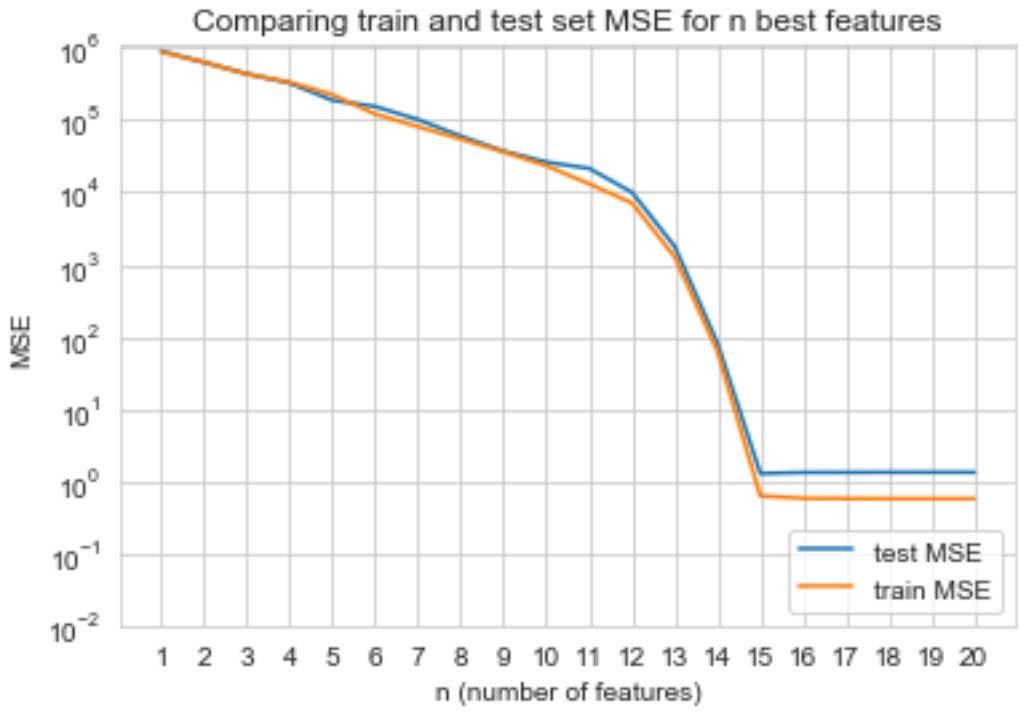
```
ranks = np.argsort(test_MSEs)
colnums = list(reversed(np.linspace(0.3, 0.99, 20)))
cols = np.zeros(20)
for i, r in enumerate(ranks):
    cols[r] = colnums[i]

plt.scatter(range(1, 21), test_MSEs, label='best MSE', c=cols, cmap=plt.cm.
 ↪seismic)
plt.title('TEST set MSE for n best features (OLS)')
plt.xlabel('n (number of features)')
plt.ylabel('MSE')
plt.xticks(range(1, 21))
plt.ylim(0, 1.2*max(test_MSEs))
plt.legend(loc='lower right')
plt.show()

print(f'Test MSE is lowest for the best {ranks[0]+1} features.\n\n')

print(tb.tabulate(dict(zip(['rank', 'feature #','train MSE', 'test MSE'
                           ],
                           [list(range(1, 21)), [f"f{i+1}" for i in␣
 ↪best_featureset],
                            best_MSE, test_MSEs
                           ])), headers="keys"))
```



```
Test MSE is lowest for the best 15 features.


  rank  feature #        train MSE      test MSE
------  -----------  -------------  ------------
     1  f5           878367         907799
     2  f16          638503         631939
     3  f4           434847         438833
     4  f12          336982         327119
     5  f2           225363         188702
     6  f6           121443         154619
     7  f13           81299.5       102197
     8  f8            54823          60055.1
     9  f19           36339.9        37190.1
    10  f9            23392.2        26241.2
    11  f17           13116.9        21473.9
    12  f15            7129.68       10034
    13  f10            1275.6         1779.84
    14  f14              62.4206        81.0563
    15  f1                0.622241       1.2726
    16  f18               0.576682       1.31484
    17  f11               0.573091       1.31802
    18  f20               0.569609       1.32218
    19  f7                0.569525       1.32324
    20  f3                0.569525       1.32324
```

[10]:
```
# plotting train and test set MSEs

plt.plot(range(1, 21), test_MSEs, label='test MSE')
plt.plot(range(1, 21), best_MSE, label='train MSE')
plt.title('Comparing train and test set MSE for n best features')
plt.xlabel('n (number of features)')
plt.ylabel('MSE')
plt.xticks(range(1, 21))
plt.ylim(0.01, 1.2*max(test_MSEs))
plt.legend(loc='lower right')
plt.yscale('log')
plt.show()

print('Based on test MSE, it would appear that the best featureset is the top␣
 ↪15 features')
f_string15 = 'f'+', f'.join([str(x) for x in best_featureset[:15]])
print(f'{f_string15}')

print("\n\n\nThe 5 worst performing features are excluded:")
f_string5 = 'f'+', f'.join([str(x) for x in best_featureset[15:]])
```

```
print(f'{f_string5}')
```



Based on test MSE, it would appear that the best featureset is the top 15 features
f4, f15, f3, f11, f1, f5, f12, f7, f18, f8, f16, f14, f9, f13, f0

The 5 worst performing features are excluded:
f17, f10, f19, f6, f2

[11]:
```
print('Comparing betas and trained weights for top 15\n\n')

sgn = np.sign(np.multiply(beta[:15], best_model[14].coef_))[:15]
print(tb.tabulate({'feature':[f'f{i}' for i in range(15)],
                   'beta': beta[:15],'coef':best_model[14].coef_,
                   'signum(beta x coef)':sgn},
                    headers='keys'))

print(f'\n\nmodel intercept: {best_model[-1].intercept_}')

print(f'\n\nY_train mean: {np.mean(Y_train)}\nY_train std: {np.std(Y_train)}')
print(f'\n\nY_test mean: {np.mean(Y_test)}\nY_test std: {np.std(Y_test)}')
```

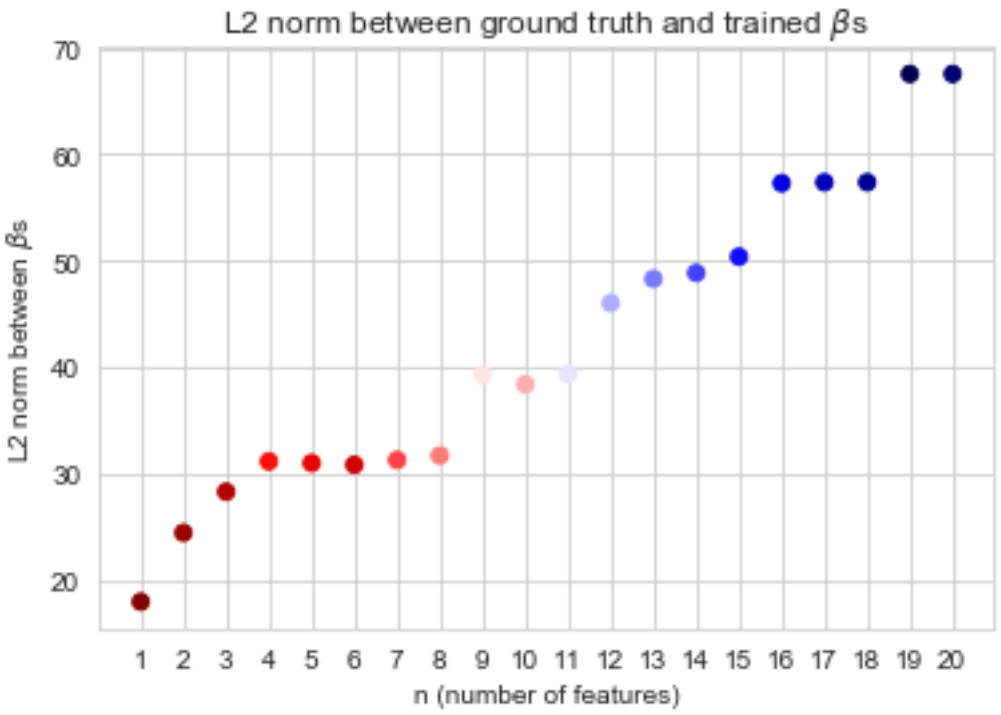
```
Comparing betas and trained weights for top 15


feature         beta       coef    signum(beta x coef)
---------  ---------  ---------  --------------------
f0           8.41296   24.4866                       1
f1          17.5056    27.3841                       1
f2         -21.3005   -38.9028                       1
f3         -38.9013   -25.2945                       1
f4          24.4879    17.4989                       1
f5          20.2686    20.2472                       1
f6           0         11.5112                       0
f7          11.2583    11.2474                       1
f8         -11.9184   -35.664                        1
f9          -5.3283   -11.9061                       1
f10          0         -3.03784                      0
f11        -25.2876    -4.05922                      1
f12         11.5036    -5.3328                      -1
f13          3.57638    3.58911                      1
f14         -4.06156    8.40132                     -1


model intercept: -305.3161570324494


Y_train mean: -1422.4022211681897
Y_train std: 1099.459363803579


Y_test mean: -1358.61291988624
Y_test std: 1078.7174916080571
```

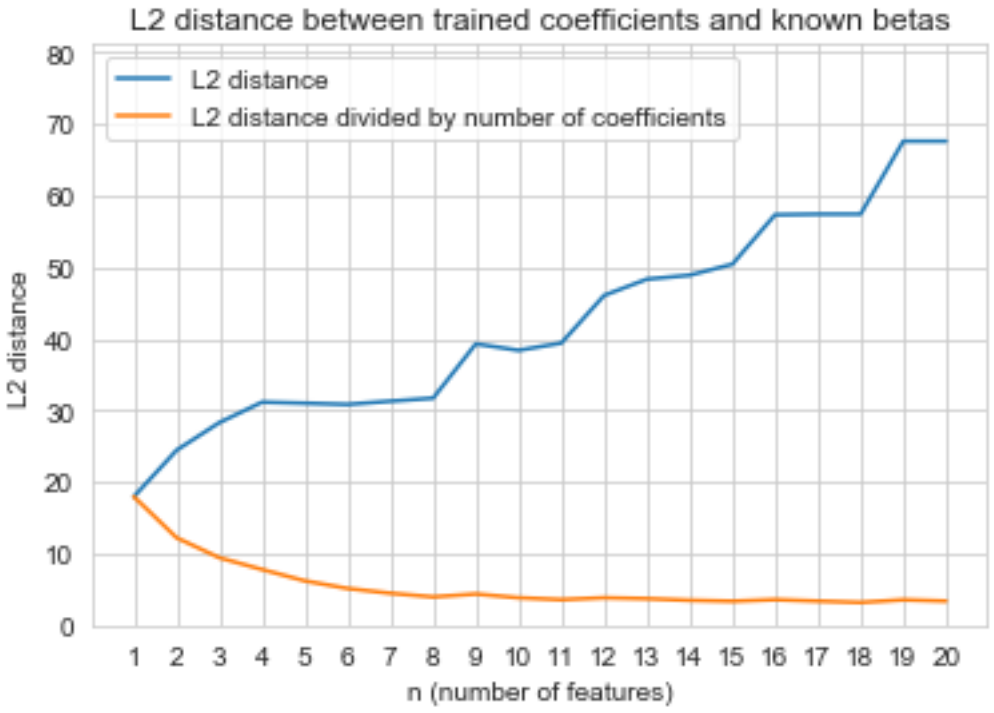
Clearly the coefficients are vastly different between the true generating process and the best fit model. We notice that for most of the βs, the trained coefficient has the same sign.

It is worth noting some of the factors that may be leading to this mismatch: - Each of the features in X has a large mean and a relatively large variance. They are all normally distributed with different means and variances - A few of the betas are set to zero. However, these could still have predictive value for the linear regression algorithm in terms of reducing the error. For example, their coefficients may be set so as to allow minor corrections to predictions made using all the previous features. - The generating process has no intercept, but the linear regression does. If the generated data clusters heavily, the intercept could explain a significant degree of the variance, allowing other coefficients. As we can see, the Ys have a mean around -1400 and standard deviation of around 1000. This means an intercept of -300 would do a decent job of beginning to model the data.

0.2 Plot of $\sqrt{\sum_{j=1}^{p}(\beta_j - \hat{\beta^r}_j^2)}$

[12]:
```
plt.clf()
beta_L2 = [np.linalg.norm(beta[:i+1] - mdl.coef_) for i, mdl in␣
 ↪enumerate(best_model)]
ranks = np.argsort(beta_L2)
colnums = list(reversed(np.linspace(0.3, 0.99, 20)))
cols = np.zeros(20)
for i, r in enumerate(ranks):
    cols[r] = colnums[i]
plt.scatter(range(1, 21), beta_L2, c=cols, cmap=plt.cm.seismic)
plt.title(r'L2 norm between ground truth and trained $\beta$s')
plt.xticks(range(1, 21))
plt.xlabel('n (number of features)')
plt.ylabel(r'L2 norm between $\beta$s')
#plt.ylim(0)
plt.show()

# plotting L2 norm between trained weights and known betas

l2_diff = [(np.linalg.norm(beta[:i+1] - mdl.coef_)) for i, mdl in␣
 ↪enumerate(best_model)]
l2_div = [(np.linalg.norm(beta[:i+1] - mdl.coef_))/(i+1) for i, mdl in␣
 ↪enumerate(best_model)]
plt.plot(range(1, 21), l2_diff, label='L2 distance')
plt.plot(range(1, 21), l2_div, label='L2 distance divided by number of␣
 ↪coefficients')
plt.title('L2 distance between trained coefficients and known betas')
plt.xlabel('n (number of features)')
plt.ylabel('L2 distance')
plt.xticks(range(1, 21))
plt.ylim(0, 1.2*max(l2_diff))
plt.legend(loc='upper left')
#plt.yscale('log')
plt.show()
```





As we can see, the L2 distance between trained coefficients and known betas for each best model

of size r increases steadily, though it does flatten out in a few zones. We observe flattening from 4-8, 9-11 and 13-15.

Comparing this with the plot on test MSE error, we observe **no real correlations**. This can be attributed to the huge difference in the trained coefficients that actually end up minimizing the MSE of prediction. Typically, we would hope that we would reach a sweet spot between over-fitting and under-fitting as the learned parameters begin to approximate the known βs. In this case, the L2 distance between learned and known coefficients would hopefully also show some kind of local optima.

However, for reasons discussed above, this does not happen in this example.

0.3 Application exercises with GSS data

Note that as per scikit-learn convention, I treat `alpha` (in sklearn) as λ and `l1_ratio` (in sklearn) as α.

[13]:
```
train_d = pd.read_csv('./data/gss_train.csv')
test_d = pd.read_csv('./data/gss_test.csv')
```

[14]:
```
print(train_d.shape, test_d.shape)
```

```
(1481, 78) (493, 78)
```

[15]:
```
train_d.head()
```

[15]:

	age	attend	authoritarianism	black	born	childs	colath	colrac	colcom	\
0	21	0	4	0	0	0	1	1	0	
1	42	0	4	0	0	2	0	1	1	
2	70	1	1	1	0	3	0	1	1	
3	35	3	2	0	0	2	0	1	0	
4	24	3	6	0	1	3	1	1	0	

	colmil	…	zodiac_GEMINI	zodiac_CANCER	zodiac_LEO	zodiac_VIRGO	\
0	1	…	0	0	0	0	
1	0	…	0	0	0	0	
2	0	…	0	0	0	0	
3	1	…	0	0	0	0	
4	0	…	0	0	0	0	

	zodiac_LIBRA	zodiac_SCORPIO	zodiac_SAGITTARIUS	zodiac_CAPRICORN	\
0	0	0	0	0	
1	0	0	0	0	
2	0	0	0	0	
3	0	1	0	0	
4	0	1	0	0	

	zodiac_AQUARIUS	zodiac_PISCES
0	0	0

```
1                0              0
2                0              0
3                0              0
4                0              0

[5 rows x 78 columns]
```

[16]:
```
Y_train, X_train = train_d['egalit_scale'], train_d[train_d.columns.
 ↪difference(['egalit_scale'])]
Y_test, X_test = test_d['egalit_scale'], test_d[test_d.columns.
 ↪difference(['egalit_scale'])]

print(Y_train.shape, X_train.shape, Y_test.shape, X_test.shape)
```

```
(1481,) (1481, 77) (493,) (493, 77)
```

[17]:
```
from sklearn.linear_model import RidgeCV, LassoCV, ElasticNetCV
from sklearn.model_selection import cross_validate

iters = 10000

ols, rdg = LinearRegression().fit(X_train, Y_train), RidgeCV(alphas=np.
 ↪linspace(0.1,2,10))
las, ela = LassoCV(n_alphas=10, max_iter=iters), ElasticNetCV(alphas=np.
 ↪linspace(0.1, 1, 10), max_iter=iters)

ols_err1, ols_err2 = MSE(Y_train, ols.predict(X_train)), MSE(Y_test, ols.
 ↪predict(X_test))

print(f'For OLS Linear Regression:\nTrain error = {ols_err1}\n\nTest error =␣
 ↪{ols_err2}')
```

```
For OLS Linear Regression:
Train error = 55.12263854924573

Test error = 63.213629623015
```

[20]:
```
def get_alpha_and_MSE(modelCV):
    modelCV.fit(X_train, Y_train)
    al = modelCV.alpha_
    tr_mse = MSE(Y_train, modelCV.predict(X_train))
    te_mse = MSE(Y_test, modelCV.predict(X_test))
    return al, tr_mse, te_mse, modelCV

n = X_train.shape[1]
```

```
print(f"Ordinary Linear Regression: {sum(np.abs(ols.coef_)<0.1)}/{n}␣
 ↪coefficients are in (-0.1, 0.1)\n")

print('Cross validation results\n')
rdg_alpha, rdg_mse1, rdg_mse2, rdgmod = get_alpha_and_MSE(rdg)
print(f"Ridge Regression: {sum(np.abs(rdgmod.coef_)<0.1)}/{n} coefficients are␣
 ↪in (-0.1, 0.1)")

las_alpha, las_mse1, las_mse2, lasmod = get_alpha_and_MSE(las)
print(f"Lasso Regression: {sum(np.abs(lasmod.coef_)<0.1)}/{n} coefficients are␣
 ↪in (-0.1, 0.1)")

ela_alpha, ela_mse1, ela_mse2, elamod = get_alpha_and_MSE(ela)
print(f"ElasticNet1: {sum(np.abs(elamod.coef_)<0.1)}/{n} coefficients are in␣
 ↪(-0.1, 0.1)")


ela2 = ElasticNetCV(alphas=[ela_alpha], l1_ratio = [.1, *list(np.linspace(0.12,␣
 ↪0.45, 5)), .5, .7, .9, .95, .99, 1], max_iter=iters)
ela2_alpha, ela2_mse1, ela2_mse2, elamod2 = get_alpha_and_MSE(ela2)
print(f"ElasticNet2: {sum(np.abs(elamod.coef_)<0.1)}/{n} coefficients are in␣
 ↪(-0.1, 0.1)\n\n")
```

```
Ordinary Linear Regression: 8/77 coefficients are in (-0.1, 0.1)

Cross validation results

Ridge Regression: 9/77 coefficients are in (-0.1, 0.1)
Lasso Regression: 59/77 coefficients are in (-0.1, 0.1)
ElasticNet1: 48/77 coefficients are in (-0.1, 0.1)
ElasticNet2: 48/77 coefficients are in (-0.1, 0.1)
```

[21]:
```
train_errs = [ols_err1, rdg_mse1, las_mse1, ela_mse1, ela2_mse1]
test_errs = [ols_err2, rdg_mse2, las_mse2, ela_mse2, ela2_mse2]

print(tb.tabulate({'CV model':['Linear Regression', f'Ridge Regression␣
 ↪(alpha={rdg_alpha})',
                               f'Lasso Regression (alpha={las_alpha})',
                                   f'ElasticNet (alpha={ela_alpha},␣
 ↪l1_ratio={elamod.l1_ratio_})',
                                  f'ElasticNet (alpha={ela2_alpha},␣
 ↪l1_ratio={elamod2.l1_ratio_})'],
                     'Train Error':train_errs,'Test Error':test_errs,
                   'Notes': ['', '', '', 'optimizing alpha w/ default l1_ratio',
```

```
                            'optimizing l1_ratio w/ optimal alpha']},
                  headers='keys'))

print(f"\n\n\nMean egalit_scale value of {Y_test.shape[0]} samples = {Y_test.
 ↪mean()}")
print(f"Standard deviation = {Y_test.std()}")
```

```
CV model                                          Train Error    Test Error  Notes
------------------------------------------------  -------------  ------------
-----------------------------------
Linear Regression                                       55.1226       63.2136
Ridge Regression (alpha=2.0)                            55.1442       62.9317
Lasso Regression (alpha=0.11488264024514108)            57.3912       62.864
ElasticNet (alpha=0.1, l1_ratio=0.5)                    57.0717       62.5072
optimizing alpha w/ default l1_ratio
ElasticNet (alpha=0.1, l1_ratio=1.0)                    57.2069       62.7786
optimizing l1_ratio w/ optimal alpha
```

```
Mean egalit_scale value of 493 samples = 19.113590263691684
Standard deviation = 9.515673735298671
```

In the ordinary regression model, 8/77 coefficients are within the threshold to be considered small i.e. within (-0.1, 0.1). In these terms, ridge regression only has one more coefficient that drops into this threshold. As expected, we see a gradual increase in the training error with more regularization. However, as the model is able to generalize better, the test error decreases.

Lasso regression seems to be overzealous with the regularization (with nearly 80% of weights being set to very small values), but still performs slightly better than ridge regression. This means that it is likely reducing the effect of most variables on the dataset on the outcome of interest. Given the complexity of social behavior, it is quite likely that there may not be any meaningful linear relationship between egalitarian outlooks and arbitrary GSS variables like `tvhours` (number of hours of TV consumed per week), or the 12 fields `zodaic_*` representing zodiac sign. This could therefore potentially be ignoring confounders in the dataset.

This is worth noting especially because the subsequent regularization methods of Lasso and ElasticNet reduce the test MSE by only around 0.13 and 0.26 respectively. In fact, searching `alpha` for ElasticNet while keeping `l1_ratio` at the default value produces the best performance. Somehow, searching `l1_ratio` over a range of values with this optimal `alpha` actually performs worse on the test set. This could be because of overfitting on the training data, has to balance the two parameters to effectively regularize.

The best performer on the test set is ElasticNet. Note that the marginal improvement is quite small from regularization.

All in all, we do a pretty good job of predicting an individual's self-reported inclination towards egalitarian behvaior. The standard deviation of the test set is around 10, and our test MSE on nearly 500 samples is just about 63. This is a fair level of accuracy.

This can be justified by the fact that the dataset also contains many variables that are known to be predictors of egalitarian attitudes - such as political orientation, confidence in science, opinions on personal liberty (such as homosexuality, marijuana consumption, etc.).

[ ]: