Resampling

June 1, 2020

1 Resampling Methods for Model Selection

When it comes to selecting the best model for a given task we need a more robust method of evaluating predictive performance. In this notebook we will look at some techniques for how we choose to sample/resample our data and what effect this has on performance estimates. We will also see how these methods can be used for hyperparameter tuning.

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

from sklearn.linear_model import LinearRegression, Lasso, Ridge
from sklearn.model_selection import cross_validate
from sklearn.datasets import make_regression
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

2 Creating a dataset

Suppose we have a true function that we would like to estimate given by:

$$f_\theta(x) = 1x + 0.1x^3$$

In reality there is an unavoidable error, ϵ, for example it could be error from measurement noise. Obviously as engineers we should be quite familiar with this.

$$y_\theta(x) = f_\theta(x) + \epsilon$$

[2]:
```
# Create arbitrary truth function
theta_true = [0, 1, 0, 0.1]
X = np.linspace(-10, 10, 1000).reshape(-1, 1)
y = theta_true[0] + theta_true[1]*X + theta_true[2]*X**2 + theta_true[3]*X**3

# Real data will have noise
X_set = np.random.uniform(-10, 10, 500)
```

```
y_set = theta_true[0] + theta_true[1]*X_set + theta_true[2]*X_set**2 +␣
 ↪theta_true[3]*X_set**3 + np.random.normal(0, 10, 500)
```

2.1 The Holdout Method

Up until this point we have chosen to split our data into two independant sets. The size of the original dataset guides the proportions of the split. If you holdout a large amount for testing (small amount for training) your test estimates may be *pessimistically biased*, as your model might have performed better if it were trained with more data. Decreasing the test set (increases training set) would help reduce this pessimistic bias, but now your test estimates may have increased variance, as you are using fewer examples to evaluate your model. Having fewer examples for evaluation causes high variance in the estimate because it depends on exaclt which examples ended up in the test split.

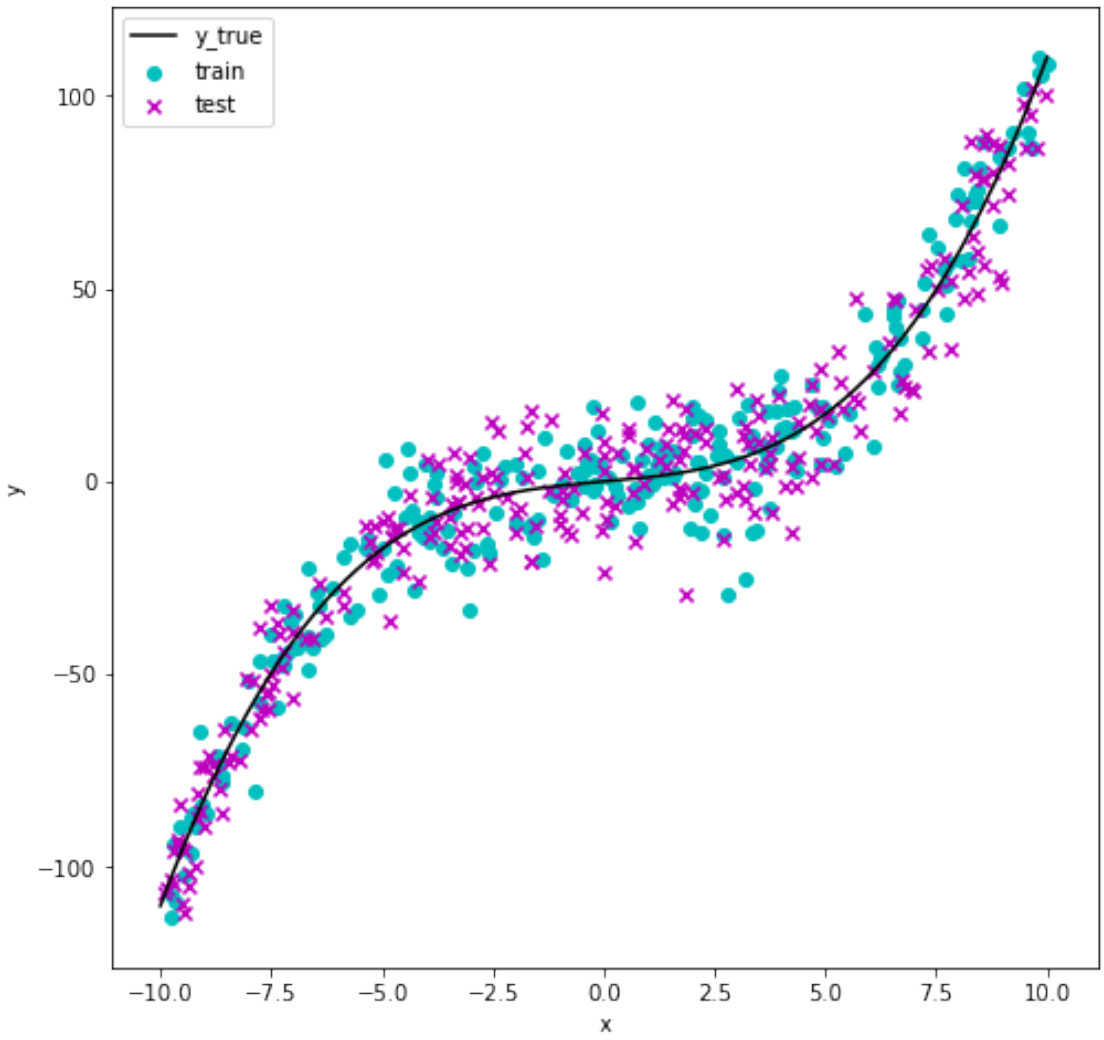
The process would be as follows:

1. Randomly split the data into two sets. Training data to fit model parameters to, and holdout test data to act as unseen data to estimate a models predictive performance.
2. Choose an appropraite learning algorithm and a fixed set of hyperparameters.
3. Fit a model on the training data using the chosen algorithm
4. Evaluate on the test set to estimate generalization performance.
5. If you think the test estimate is good enough you could retrain your model on the entire dataset (training + test) - assuming it has not reached its full capacity with the limited training set.

We saw this in the Features and Regularization notebook, so I won't repeat it here, but we can quickly visualise a single split just to remind ourselves what the data looks like.

[3]:
```
# Basic train, test split to evaluate models.
# I've named the training set "learn" instead of "train" for reasons that will␣
 ↪be clear later.
X_learn, X_test, y_learn, y_test = train_test_split(X_set, y_set, test_size=0.
 ↪5, random_state=1)
```

[4]:
```
fig = plt.figure(figsize=(8, 8))
plt.plot(X, y, color='k', label='y_true')
plt.scatter(X_learn, y_learn, color='c', label='train')
plt.scatter(X_test, y_test, color='m', marker='x', label='test')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



3 Repeated Holdout

The estimate of a models performance might depend greatly on the exact split that was used for the data. By performing the holdout procedure only once we cannot be sure if the model has performed well/badly due to the specific examples that were in each set. We could make this more robust by simply performing multiple splits and performing the training and evaluation procedures on each of the new splits. This might also provide us with an idea of how stable the model is - if the test estimates vary greatly between each split then it may be a sign the model suffers from high variance.

[5]:
```
def repeatedHoldout(regressors, X, y, feature_transform, test_size=0.5, k=10):
    score_dict = {}
    for name, regressor in regressors.items():
        score_k = []
```

```
        for i in range(k):
            X_train, X_test, y_train, y_test = train_test_split(X, y,␣
 ↪test_size=test_size)

            if name != 'simple_reg':
                X_in = feature_transform.fit_transform(X_train.reshape(-1, 1))
                X_t = feature_transform.fit_transform(X_test.reshape(-1, 1))
            else:
                X_in = X_train.reshape(-1, 1)
                X_t = X_test.reshape(-1, 1)

            regressor.fit(X_in, y_train)
            score_k.append(regressor.score(X_t, y_test))

        score_dict[name] = score_k

    return score_dict
```

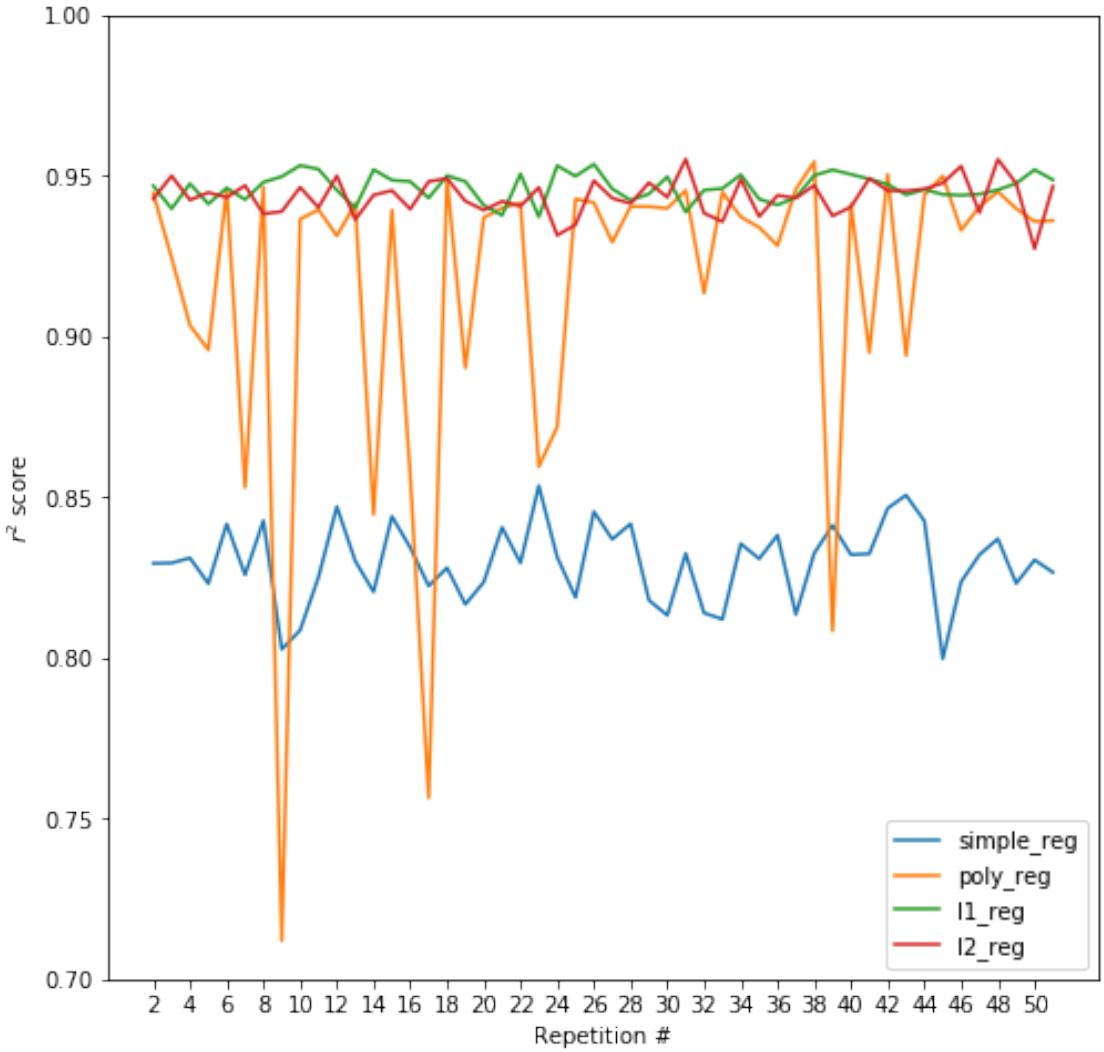
[6]:
```
# Define learning algorithms
simple_reg = LinearRegression(normalize=True)
poly_reg = LinearRegression(normalize=True)
l1_reg = Lasso(alpha=0.1, max_iter=10000, normalize=True)
l2_reg = Ridge(alpha=0.1, max_iter=10000, normalize=True)

regressors = {
    'simple_reg': simple_reg,
    'poly_reg': poly_reg,
    'l1_reg': l1_reg,
    'l2_reg': l2_reg
}

poly = PolynomialFeatures(degree=25, include_bias=False)
```

[7]:
```
k = 50
repeated_holdout_scores = repeatedHoldout(regressors, X_set, y_set, poly,␣
 ↪test_size=0.5, k=k)
```

We can plot these results out to visualise the performance for each regressor across multiple random splits of the data.

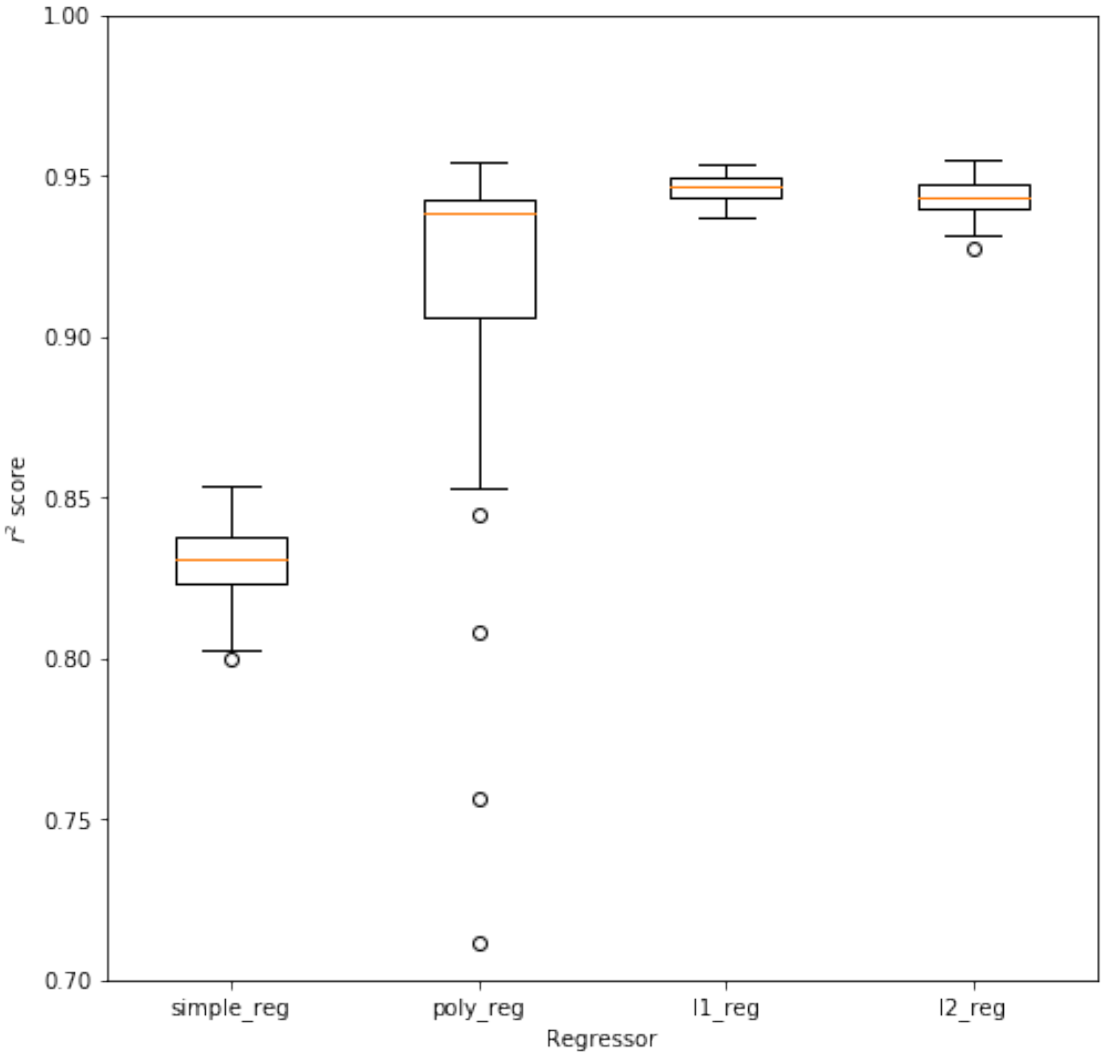
[14]:
```
fig = plt.figure(figsize=(8, 8))
for i, (name, score) in enumerate(repeated_holdout_scores.items()):
    plt.plot(score, label=name)

plt.legend()
plt.xticks(np.arange(k, step=2), (np.arange(k)+1)*2)
plt.xlabel('Repetition #')
```

```
plt.ylabel('$r^2$ score')
plt.ylim(0.7, 1)
plt.show()
```



We can also use box-whisker plots to make the results easier to interpret,

[15]:
```
fig = plt.figure(figsize=(8, 8))
plt.boxplot(repeated_holdout_scores.values(), labels=repeated_holdout_scores.
 ↪keys())
plt.ylabel('$r^2$ score')
plt.xlabel('Regressor')
plt.ylim(0.7, 1)
plt.show()
```



From these results we can see a few things.

The simple linear model fit on a single input feature has high bias (low scores on average) as expected and in this case it also has high variance (scores changed drastically depending on the split).

The results for the standard linear regression model with no regularization, but fit on polynomial input features, stands out. If we only performed a single split we might have ended up with a seemingly good model (many r^2 scores compete with the regularized model scores), but this would be a poor estimate of generalization performance, as we can see there are also many cases where the specific splits caused significantly worse performance.

Both the l1 and l2 regularized linear models show low bias and low variance compared to the other two models, with the l1 model performing slightly better which might also be expected as the sparsity it introduces results in a model that is closer to the true function.

3.1 Information leak

With a repeated holdout the test data is not completely independant between splits because we are sampling randomly with replacement (some examples might appear in the test set multiple times). With this in mind it is possible that the estimates from the repeated holdout method would be overly optimistic because you could be evaluating the same examples at test time across multiple runs. One way we can produce better estimates is to use a three-way holdout method.

4 Three-way Holdout

The three-way holdout is an extension of the holdout methods that introduces an additional split, usually called a validation set. In total you would now have three separate sets: training, validation and test.

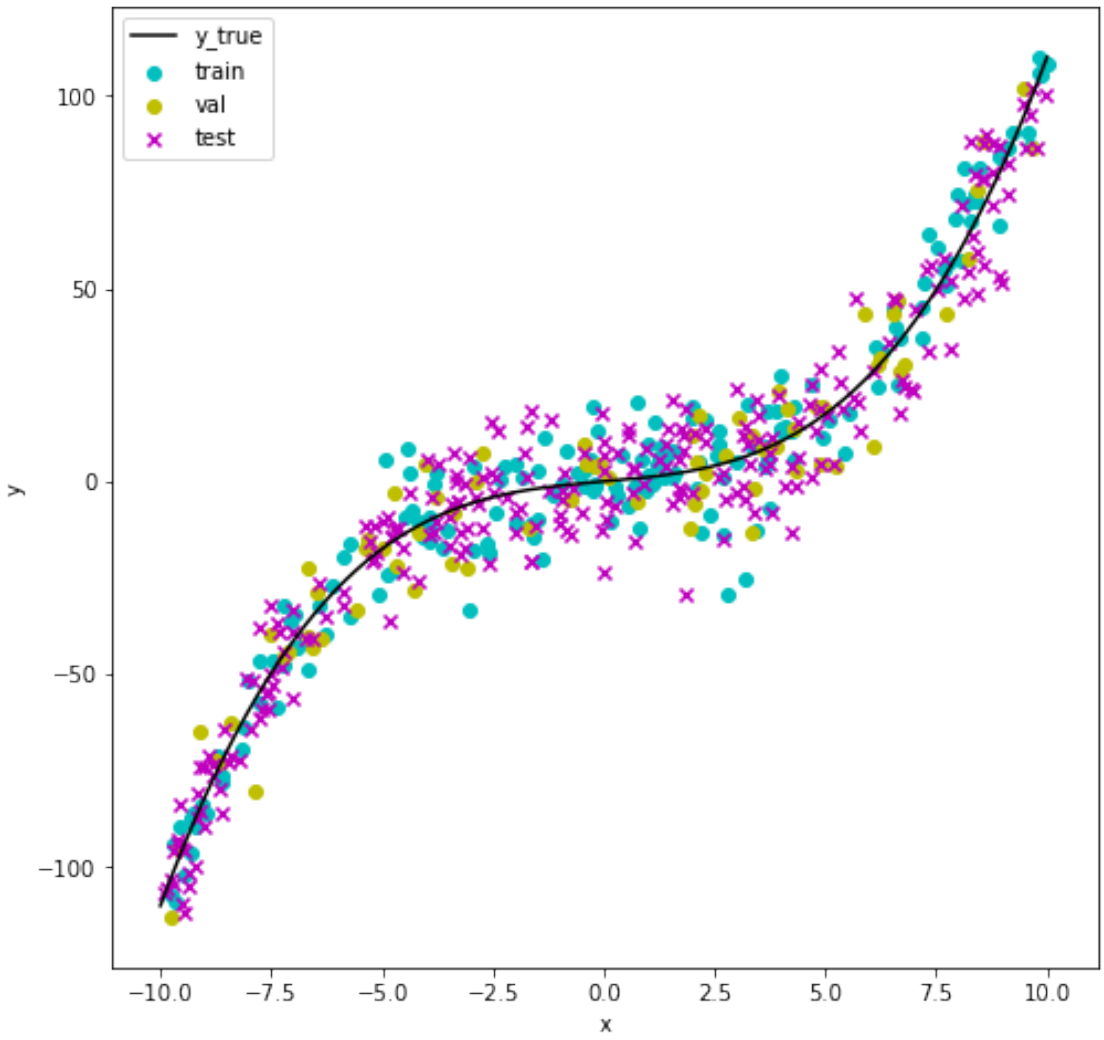
- Training: The set used to determine how to update model parameters
- Validation: The set used to evaluate and compare different models, usually to compare the performance with different hyperparameters
- Test: The set used to evaluate the final model chosen based on the validation set performance. This provides an estimate of the generalization performance.

The process would be as follows:

1. Randomly split dataset into three parts: training, validation and test.
2. Fit model with specified hyperparameter values and evaluate on the validation set.
3. Repeat step 2 with different hyperparameters.
4. Compare the validation performance of the trained models and choose the best performing model and set of hyperparameter values. Optionally train the chosen model with the best performing hyperparameters on the combined training + validation data.
5. Evaluate the generalization performance of the chosen model on the test set.
6. If the test set performance is acceptable you can retrain on the entire dataset with the selected model and hyperparameters from step 4.

[20]:
```
X_train, X_val, y_train, y_val = train_test_split(X_learn, y_learn, test_size=0.
↪3)
```

[21]:
```
fig = plt.figure(figsize=(8,8))
plt.plot(X, y, color='black', linestyle='-', label='y_true')
plt.scatter(X_train, y_train, color='c', label='train')
plt.scatter(X_val, y_val, color='y', label='val')
plt.scatter(X_test, y_test, color='m', marker='x', label='test')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.show()
```



[23]:
```
def threeWayHoldout(X, y, feature_transform, test_size=0.5, val_size=0.1):
    score_dict = {}
    X_learn, X_test, y_learn, y_test = train_test_split(X, y,␣
↪test_size=test_size, random_state=1)
    X_train, X_val, y_train, y_val = train_test_split(X_learn, y_learn,␣
↪test_size=val_size, random_state=1)

    X_train = feature_transform.fit_transform(X_train.reshape(-1, 1))
    X_val = feature_transform.fit_transform(X_val.reshape(-1, 1))

    alphas = np.logspace(-8, 1, num=10)
    for alpha in alphas:
        reg = Ridge(alpha=alpha, max_iter=10000, normalize=True)
        reg.fit(X_train, y_train)
```

```
        score_dict[alpha] = reg.score(X_val, y_val)

#         Or if you wanted to use the mean squared error
#         y_hat = reg.predict(X_val)
#         score_dict[alpha] = mean_squared_error(y_hat, y_val)

    return score_dict, X_learn, X_test, y_learn, y_test
```

[24]:
```
score_dict, X_learn, X_test, y_learn, y_test  = threeWayHoldout(X_set, y_set,␣
↪poly, test_size=4.0/5.0, val_size=0.3)
```

The r^2 scores for the various alphas are shown below

[35]:
```
for alpha, score in score_dict.items():
    print("Alpha: {}\tScore: {}".format(alpha, score))
```

```
Alpha: 1e-08    Score: 0.9510967468961554
Alpha: 1e-07    Score: 0.9317180469594069
Alpha: 1e-06    Score: 0.8952482428210974
Alpha: 1e-05    Score: 0.9227523734361731
Alpha: 0.0001   Score: 0.9519939792346893
Alpha: 0.001    Score: 0.9597622495716588
Alpha: 0.01     Score: 0.9616756335313319
Alpha: 0.1      Score: 0.9557129413113044
Alpha: 1.0      Score: 0.8960266427839704
Alpha: 10.0     Score: 0.6055792348398259
```

Next you could choose the alpha value that produces the highest score and retrain the model using the training and validation sets. This increases the amount of data for training and should therefore improve performance, especially for smaller datasets.

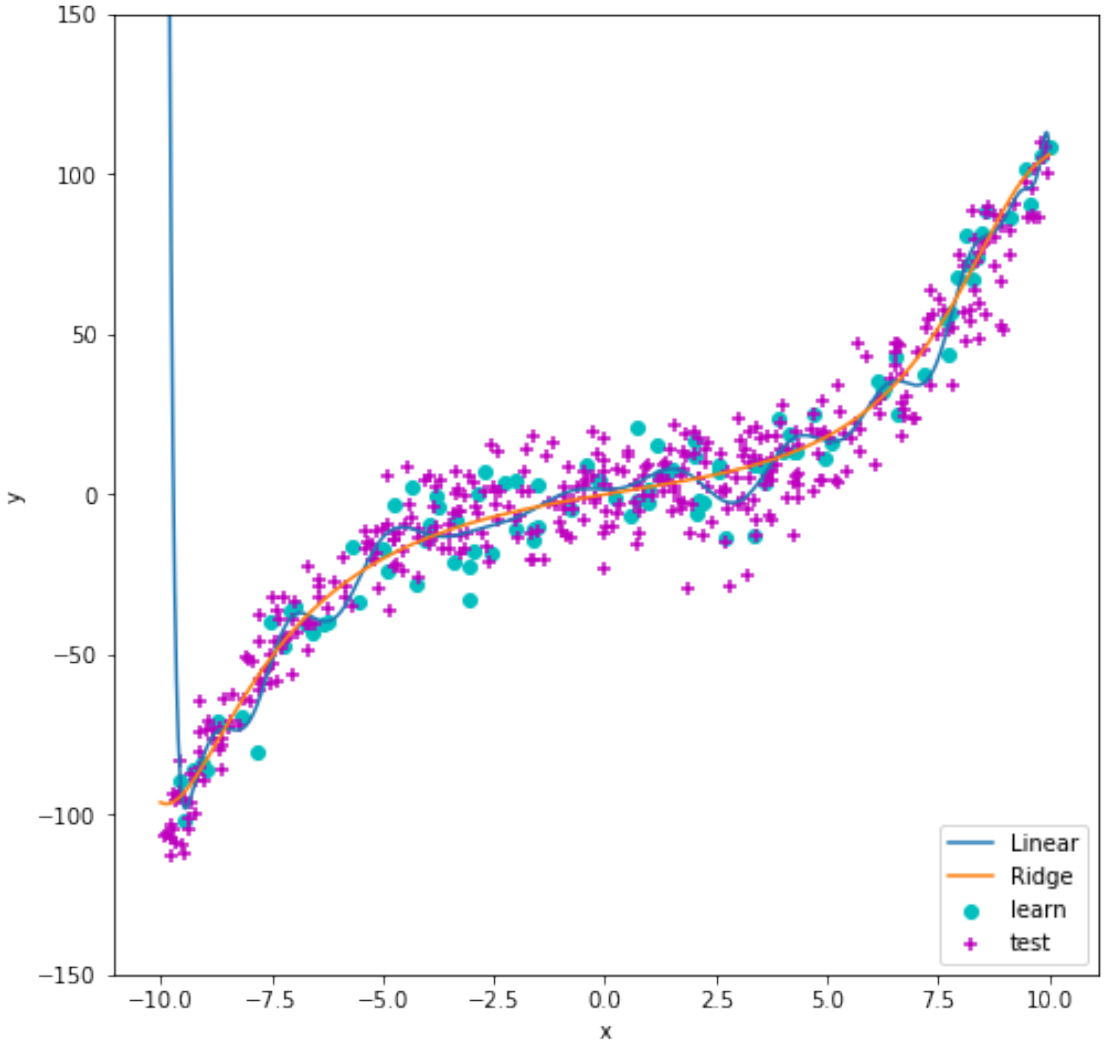
[36]:
```
alpha = max(score_dict, key=score_dict.get)
l2_reg = Ridge(alpha=alpha, max_iter=10000, normalize=True)
l2_reg.fit(poly.fit_transform(X_learn.reshape(-1, 1)), y_learn)
poly_reg.fit(poly.fit_transform(X_learn.reshape(-1, 1)), y_learn)

l2_test_score = l2_reg.score(poly.fit_transform(X_test.reshape(-1, 1)), y_test)
poly_reg_score = poly_reg.score(poly.fit_transform(X_test.reshape(-1, 1)),␣
↪y_test)
```

[37]:
```
print("Scores after retraining:\n\nL2 test score: {}\nLinear test score: {}\n".
↪format(l2_test_score, poly_reg_score))
```

```
Scores after retraining:

L2 test score: 0.9424282025878136
Linear test score: -0.5103994309555069
```

[38]:
```
# generate predicted curves to plot
y_hat_poly = poly_reg.predict(poly.fit_transform(X.reshape(-1, 1)))
y_hat_l2 = l2_reg.predict(poly.fit_transform(X.reshape(-1, 1)))
```

[39]:
```
fig = plt.figure(figsize=(8, 8))
plt.scatter(X_learn, y_learn, color='c', label='learn')
plt.scatter(X_test, y_test, color='m', marker='+', label='test')
plt.plot(X, y_hat_poly, label='Linear')
plt.plot(X, y_hat_l2, label='Ridge')
plt.xlabel('x')
plt.ylabel('y')
plt.ylim(-150, 150)
plt.legend()
plt.show()
```



Here we can visualize the results. As expected, and seen on previous evaluations the regularized regression produces a better fit.

The three-way holdout provides some improvements by maintaining one completely independant set, however, the validation and training sets are still sampled randomly, which may result in some examples never being used to validate the model.

You could perform multiple random splits with the three-way holdout, as with the standard repeated holdout method, however there is a better option known as the k-Fold Cross-Validation.

5 k-Fold Cross-Validation

To alleviate some of these issues we can repeat the splitting processing, k, number of times, however we need to make sure that each time we split the data that we have completely new examples for validation. This ensures that over all folds every example is used for both training ($k-1$ times) and validation (once). In this way we are sampling without replacement for the validation set and we can be more confident that the validation scores/errors provide better estimates of the test error.

This process is illustrated below (source: https://scikit-learn.org/stable/modules/cross_validation.html)

There is a bias-variance tradeoff when it comes to the choice of k. The more folds you use, the more data is available for training and the lower the bias will be. However the variance will be higher (although good choices for k should have lower variance compared to the holdout methods), as you are validating on small validation sets which may differ greatly depending on the split. More splits also results in an increase in computational costs.

The process is essentially a repeated three-way holdout method, except that the validation sets must remain independant.

The process would be as follows:

1. Randomly split the dataset into two parts: "learning" and test sets. Keep the test set aside to evaluate generalization performance.
2. Split the learning set into a validation and training set, k, times, each time ensuring that an independant validation set is sampled.
3. Fit a model with specified hyperparameter values on each training set and evaluate on each corresponding validation set.
4. Repeat step 3 with different hyperparameters.
5. Compare the average validation performance of the trained models and choose the best performing model and set of hyperparameter values. Optionally train the chosen model with the best performing hyperparameters on the combined training + validation data.
6. Evaluate the generalization performance of the chosen model on the test set.
7. If the test set performance is acceptable you can retrain on the entire dataset with the selected model and hyperparameters from step 4.

[40]:
```
def getCVScores(regressors, X_learn, y_learn, cv=5):
    score_dict = {}
    scoring = ["r2", "neg_mean_squared_error"]

    poly = PolynomialFeatures(degree=25, include_bias=False)
```

```
    for name, regressor in regressors.items():
        if name != 'simple_reg':
            X_in = poly.fit_transform(X_learn.reshape(-1, 1))
        else:
            X_in = X_learn.reshape(-1, 1)

        cv_scores = cross_validate(
            regressor,
            X_in,
            y_learn,
            cv=cv,
            scoring=scoring,
            return_estimator=True)

        score_dict[name] = {'test_r2': cv_scores['test_r2'],
                            'test_MSE':␣
 ↪-cv_scores["test_neg_mean_squared_error"],
                            'estimators': cv_scores['estimator']}
    return score_dict
```

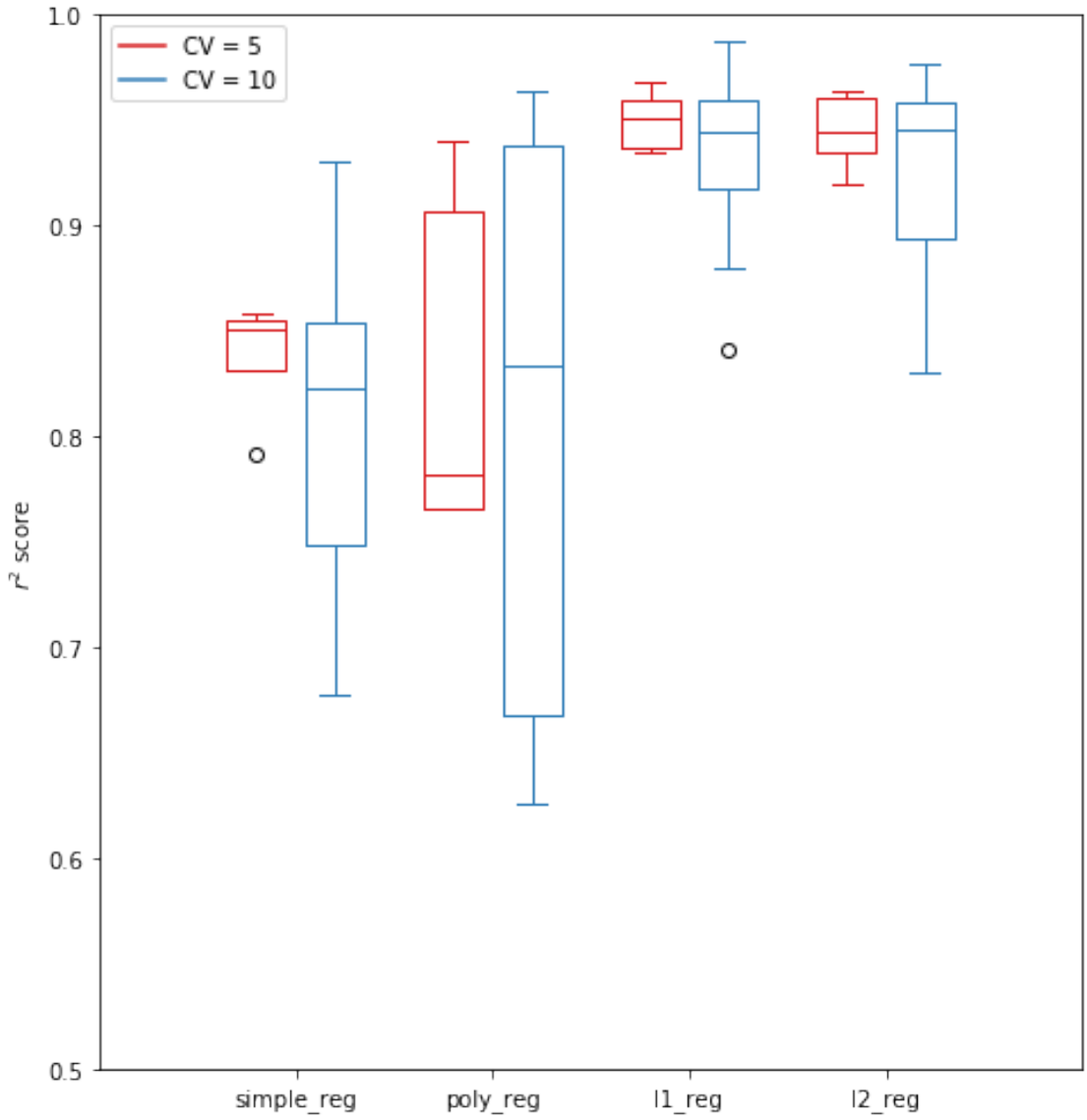
[44]:
```
score_dict_5 = getCVScores(regressors, X_learn, y_learn, cv=5)
score_dict_10 = getCVScores(regressors, X_learn, y_learn, cv=10)
```

[55]:
```
r2_scores_5 = [list(value['test_r2']) for value in list(score_dict_5.values())]
r2_scores_10 = [list(value['test_r2']) for value in list(score_dict_10.
 ↪values())]
```

You could then use the average score across all of the folds for a given model to estimate the generalization performance. To illustrate the performance of the different models we can once again use a box plot.

[66]:
```
c1 = '#D7191C'
c2 = '#2C7BB6'
median_c = 'orange'
fig = plt.figure(figsize=(7, 7))

def set_box_color(bp, color):
    plt.setp(bp['boxes'], color=color)
    plt.setp(bp['whiskers'], color=color)
    plt.setp(bp['caps'], color=color)
    plt.setp(bp['medians'], color=color)
    plt.setp(bp['fliers'], color=color)

bp1 = plt.boxplot(
    r2_scores_5,
    positions=np.array(range(len(r2_scores_5)))*2.0-0.4,
    widths=0.6,
```

```
)

bp2 = plt.boxplot(
    r2_scores_10,
    positions=np.array(range(len(r2_scores_10)))*2.0+0.4,
    widths=0.6,
)

set_box_color(bp1, c1)
set_box_color(bp2, c2)

plt.plot([], c=c1, label='CV = 5')
plt.plot([], c=c2, label='CV = 10')
plt.legend()
plt.ylabel('$r^2$ score')

ticks = list(score_dict_5.keys())
plt.xticks(range(0, len(ticks) * 2, 2), ticks)
plt.xlim(-2, len(ticks)*2)
plt.tight_layout()


plt.ylim(0.5, 1)
plt.show()
```

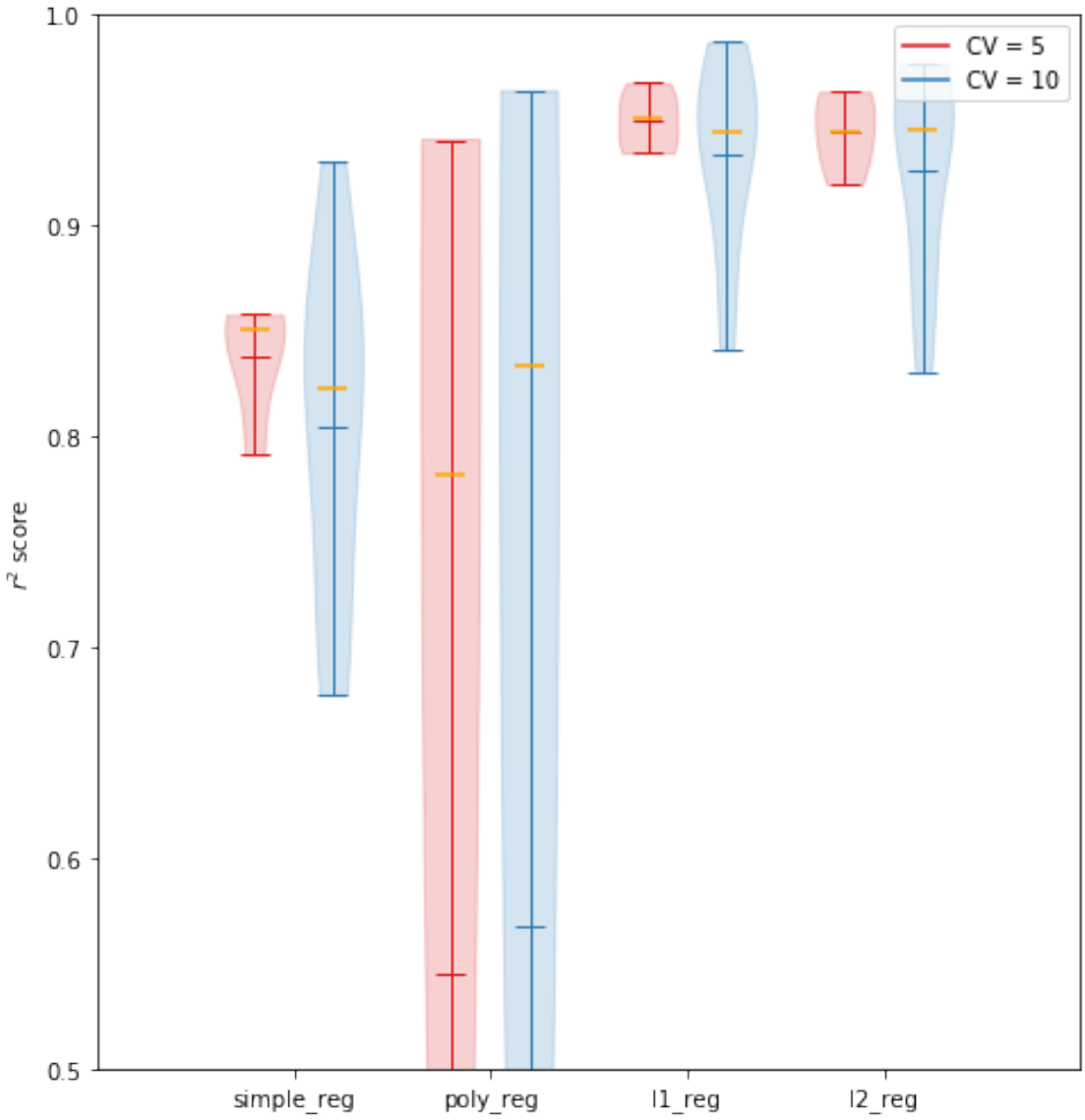


Here we can see how the different models perform, and for different k values. The linear regression model fit on polynomial features exhibits high variance, whereas the regularized regression models show less variance and lower bias. The 10 fold CV produces more variability in the scores, but sometimes scores higher - indicating lower bias.

Another type of plot you could use is a violin plot, which is similar to a box plot, except that it shows the distribution of values.

[68]:
```
fig = plt.figure(figsize=(7, 7))

vp1 = plt.violinplot(
    r2_scores_5,
    positions=np.array(range(len(r2_scores_5)))*2.0-0.4,
```

```
    widths=0.6,
    showmeans=True,
    showmedians=True,
    showextrema=True
)

vp2 = plt.violinplot(
    r2_scores_10,
    positions=np.array(range(len(r2_scores_10)))*2.0+0.4,
    widths=0.6,
    showmeans=True,
    showmedians=True,
    showextrema=True
)

plt.plot([], c=c1, label='CV = 5')
plt.plot([], c=c2, label='CV = 10')
plt.legend()
plt.ylabel('$r^2$ score')

ticks = list(score_dict_5.keys())
plt.xticks(range(0, len(ticks) * 2, 2), ticks)
plt.xlim(-2, len(ticks)*2)
plt.tight_layout()

ymin = min(min(r2_scores_5 + r2_scores_10))

def set_colour(plot, colour):
    for pc in plot['bodies']:
        pc.set_facecolor(colour)
        pc.set_edgecolor(colour)
        pc.set_alpha(0.2)
    for partname in ('cbars','cmins','cmaxes','cmeans'):
        vp = plot[partname]
        vp.set_edgecolor(colour)
        vp.set_linewidth(1)
    plot['cmedians'].set_edgecolor('orange')


set_colour(vp1, c1)
set_colour(vp2, c2)


plt.ylim(0.5, 1)
plt.show()
```



You should now have a better understanding of how the choices of splitting data might affect our performance estimates. k-Fold Cross-validation is widely used for models that do not take large amounts of time to train, however, as we start moving into more complex models with large datasets it becomes less feasible and not as necessary.