# » Hold-Out Method

- So far we've evaluated cost function on whole of training data ...
- ... but we're really interested in how well our model makes predictions for *new* data i.e. how well the model *generalises*.
- Split training data into (i) test data used to evaluate prediction performance and (i) training data used to train model. E.g. 20% is test data, 80% is training data.
- There is a trade-off in how we split the data. If we use more for the training part then we expect our model to be better trained, but then we have less data to test the model on. And vice versa. An 80/20 or 90/10 split is common.
- Typically split the data randomly. So as avoid inadvertently introducing bias.

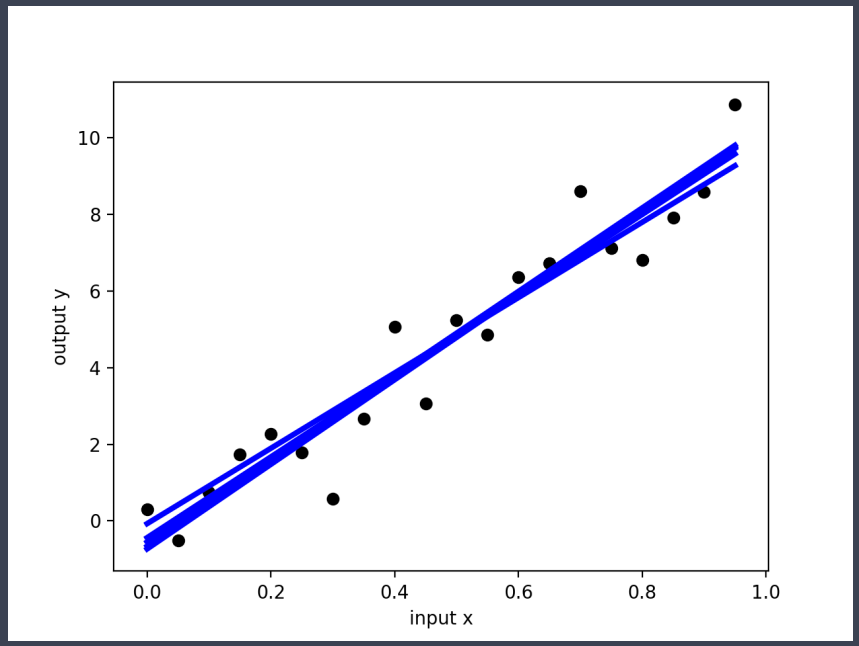
# » Hold-Out Method

```
import numpy as np
X = np.arange(0,1,0.05).reshape(-1, 1)
y = 10*X + np.random.normal(0.0,1.0,X.size).reshape(-1, 1)
for i in range(5):
        from sklearn.model_selection import train_test_split
        Xtrain, Xtest, ytrain, ytest = train_test_split(X,y,test_size=0.2)

        from sklearn.linear_model import LinearRegression
        model = LinearRegression().fit(Xtrain, ytrain)

        ypred = model.predict(Xtest)
        from sklearn.metrics import mean\_squared\_error
        print("intercept %f, slope %f, square error %f"%(model.intercept_, model.coef_,mean_squared_error(ytest,ypred)))
```

# » Hold-Out Method




intercept -0.146890, slope 10.174736, square error 0.680207
intercept -0.050447, slope 9.898857, square error 1.105285
intercept -0.154663, slope 10.048717, square error 1.212909
intercept -0.441200, slope 10.543796, square error 1.904468
intercept -0.117850, slope 9.859572, square error 1.412553

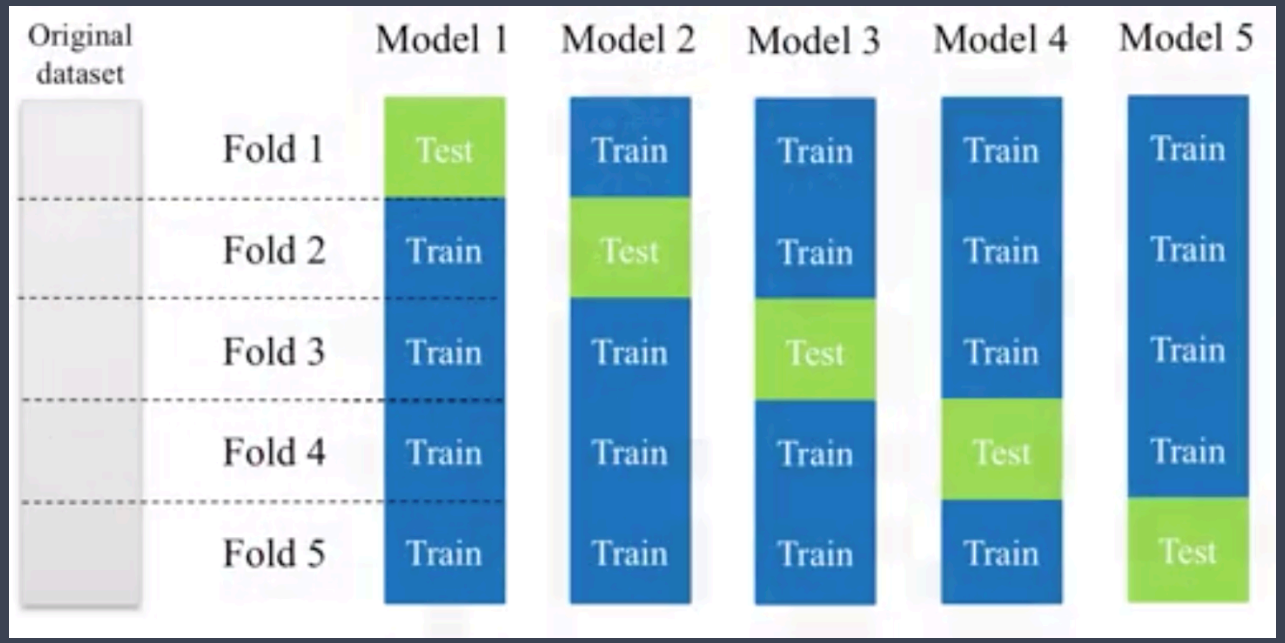
- As we use different subsets of the training data to train the model we get slightly different model parameters and predictions.
- By doing this repeatedly we get an idea of how robust/fragile our model and its predictions are

# » $k$-Fold Cross-validation

Repeatedly applying hold-out method using random splits is fine, but its more common to use $k$-fold cross-validation.

- Divide our data into $k$ equal sized parts
- Use part 1 as test data and the rest as training data. i.e. train model using all of the data except part 1, then calc $J(\theta)$ for part 1 data
- Use part 2 as test data and the rest as training data, and so on.
- This gives us $k$ estimates of $J(\theta)$ and so we can use this to estimate the average and the spread of values.

# » *k*-Fold Cross-validation



# » sklearn *k*-Fold Cross-validation

```
from sklearn.model_selection import cross_val_score
scores = cross_val_score(model, X, y, cv=5, scoring='neg_mean_squared_error')
print(scores)
print("Accuracy: %0.2f (+/- %0.2f)" % (scores.mean(), scores.std()))

from sklearn.model_selection import KFold
kf = KFold(n_splits=5)
for train, test in kf.split(X):
        from sklearn.linear_model import LinearRegression
        model = LinearRegression().fit(X[train], y[train])
        ypred = model.predict(X[test])
        from sklearn.metrics import mean_squared_error
        print("intercept %f, slope %f, square error %f"%(model.intercept_, model.coef_,mean_squared_error(y[test],ypred)))
```

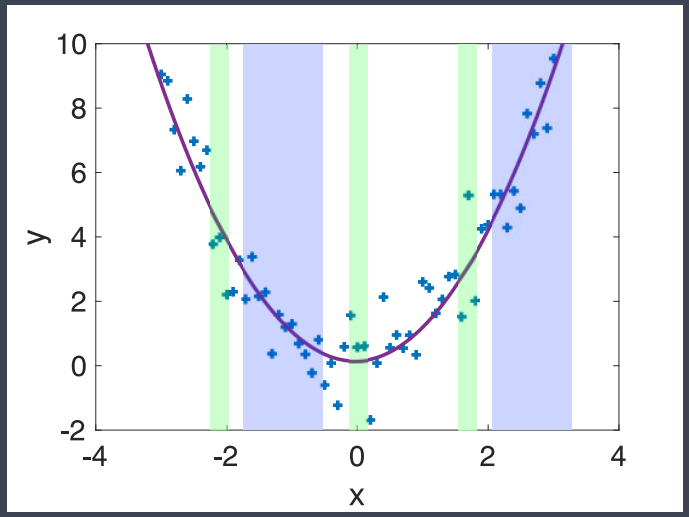
# » sklearn *k*-Fold Cross-validation

## Example output:

```
Accuracy: -1.08 (+/- 0.91)
intercept -0.810381, slope 11.420786, square error 0.274081
intercept -0.741972, slope 11.354662, square error 0.693850
intercept -0.946450, slope 11.580333, square error 2.841125
intercept -0.845299, slope 11.667406, square error 0.584412
intercept -0.601880, slope 10.690939, square error 1.013331
```

# » $k$-Fold Cross-validation

* How to choose $k$ ? Common choices are $k = 5$ or $k = 10$.
* When we calculate the accuracy of model predictions on test data there are two main reasons why the calculated accuracy value will tend to fluctuate:
  1. The test data is noisy. The realisation of the noise changes from one set of test data to another (think of tossing a coin $n/k$ times, the pattern of heads/tails changes in each set of $n/k$ tosses).
  2. The training data is noisy and so the learned model parameters change from one set of training data to another.




* Each test set has $n/k$ points. We average over these to calculate the prediction accuracy $\rightarrow$ averaging smooths out the noise provided $n/k$ is large enough i.e. $k$ small enough.

# » $k$-Fold Cross-validation

1. The test data is noisy.
2. The training data is noisy and so the learned model parameters change from one set of training data to another.

- We want to use as much data as possible to train the model, so that we learn representative parameter values $\rightarrow$ fluctuations in model are not due to inadequate training. So want $k$ large and so training data size $(k-1)/n$ large.
- Also, as $k$ increases the computation times increases (remember we need to fit the model $k$ times), so don't want $k$ to be too large.
- $k = 5$ or $k = 10$ is a reasonable compromise value, but also sometimes use other values e.g. *leave one out* cross-validation uses $k = m$, #training data points.

# » Tuning Model Hyperparameters

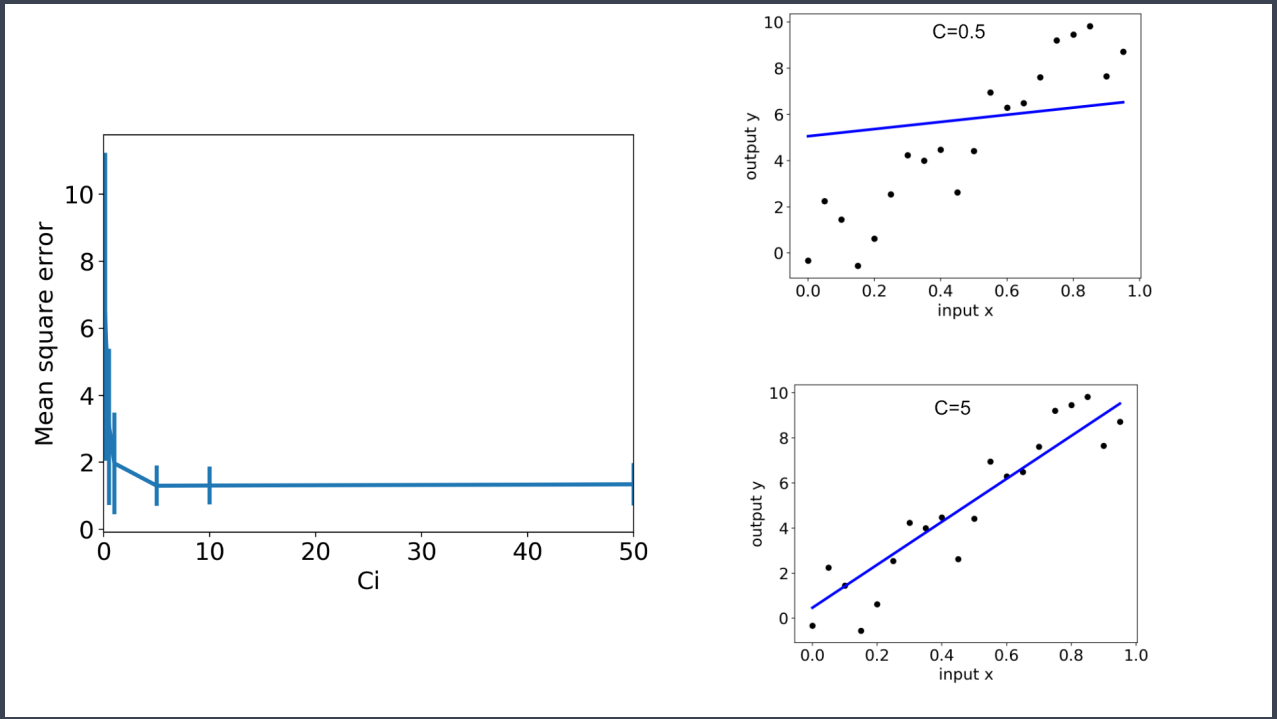
Suppose we add a penalty to the linear regression cost function:

- Model: $h_\theta(x) = \theta^T x$
- Cost Function: $J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \theta^T \theta / C$
- This variant of linear regression is called *Ridge Regression*
- Parameter $C$ in cost function is called a *hyperparameter* (to distinguish it from model parameters $\theta$.
- How to choose value for $C$?
- ... use cross-validation. Scan across a range of values for $C$, do cross-validation for each value of $C$ and plot distribution of prediction error.
- Rule of thumb: increase $C$ value by factor of 5 or 10 so can quickly scan across a large range e.g. [0.1,1,10,100] or [0.1, 0.5, 1, 5, 10, 50, 100]

# » Tuning Model Hyperparameters

```
mean_error=[]; std_error=[]
Ci_range = [0.1, 0.5, 1, 5, 10, 50, 100]
for Ci in Ci_range:
        from sklearn.linear_model import Ridge
        model = Ridge(alpha=1/(2*Ci))
        temp=[]
        from sklearn.model_selection import KFold
        kf = KFold(n_splits=5)
        for train, test in kf.split(X):
                model.fit(X[train], y[train])
                ypred = model.predict(X[test])
                from sklearn.metrics import mean_squared_error
                temp.append(mean_squared_error(y[test],ypred))
        mean_error.append(np.array(temp).mean())
        std_error.append(np.array(temp).std())
import matplotlib.pyplot as plt
plt.errorbar(Ci_range,mean_error,yerr=std_error)
plt.xlabel('Ci'); plt.ylabel('Mean square error')
plt.xlim((0,50))
plt.show()
```

# » Tuning Model Hyperparameters




* Choosing $C$ too small (making the $\theta^T\theta/C$ penalty too strong) increases the prediction error
* $C$ values $\geq 5$ look ok. *Would you choose $5$ or $50$ for $C$?*
  * Avoiding over-fitting: try to use "simplest" model possible $\rightarrow$ smallest value of $C$
* In your assignments and projects, when hyperparameter values need to be selected then unless otherwise stated it is mandatory to present cross-validation analysis to support your choice of values.

# » Tuning Model Hyperparameters

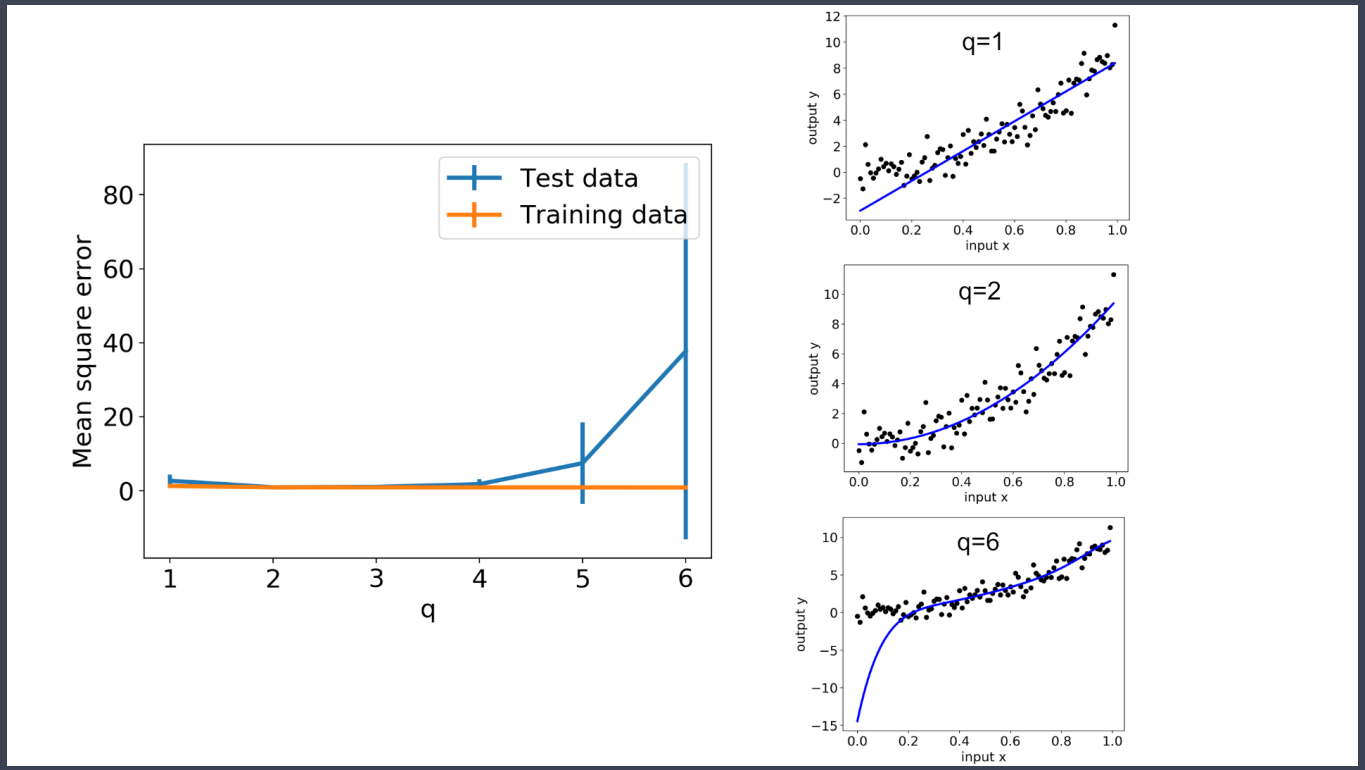
Another example, polynomial features.

- Linear regression (no penalty)
- Actual data is quadratic plus noise, but suppose we don't know this
- Use features $[1,\ x,\ x^2,\ x^3,\ x^4,\ \ldots x^q]$
- How to choose value for hyperparameter $q$?
- Scan across a range of values for $q$, do cross-validation for each value of $q$ and plot distribution of prediction error.
- How to choose range of values to scan across?

# » Tuning Model Hyperparameters

```
import numpy as np
X = np.arange(0,1,0.01).reshape(-1, 1)
y = 10*(X**2) + np.random.normal(0.0,1.0,X.size).reshape(-1, 1)
from sklearn.model_selection import KFold
kf = KFold(n_splits=5)
import matplotlib.pyplot as plt
plt.rc('font', size=18); plt.rcParams['figure.constrained_layout.use'] = True
mean_error=[]; std_error=[]
q_range = [1,2,3,4,5,6]
for q in q_range:
        from sklearn.preprocessing import PolynomialFeatures
        Xpoly = PolynomialFeatures(q).fit_transform(X)
        from sklearn.linear_model import LinearRegression
        model = LinearRegression()
        temp=[]; plotted = False
        for train, test in kf.split(Xpoly):
                model.fit(Xpoly[train], y[train])
                ypred = model.predict(Xpoly[test])
                from sklearn.metrics import mean_squared_error
                temp.append(mean_squared_error(y[test],ypred))
                if ((q==1) or (q==2) or (q==6)) and not plotted:
                        plt.scatter(X, y, color='black')
                        ypred = model.predict(Xpoly)
                        plt.plot(X, ypred, color='blue', linewidth=3)
                        plt.xlabel("input x"); plt.ylabel("output y")
                        plt.show()
                        plotted = True
        mean_error.append(np.array(temp).mean())
        std_error.append(np.array(temp).std())
plt.errorbar(q_range,mean_error,yerr=std_error,linewidth=3)
plt.xlabel('q')
plt.ylabel('Mean square error')
plt.show()
```

# » Tuning Model Hyperparameters



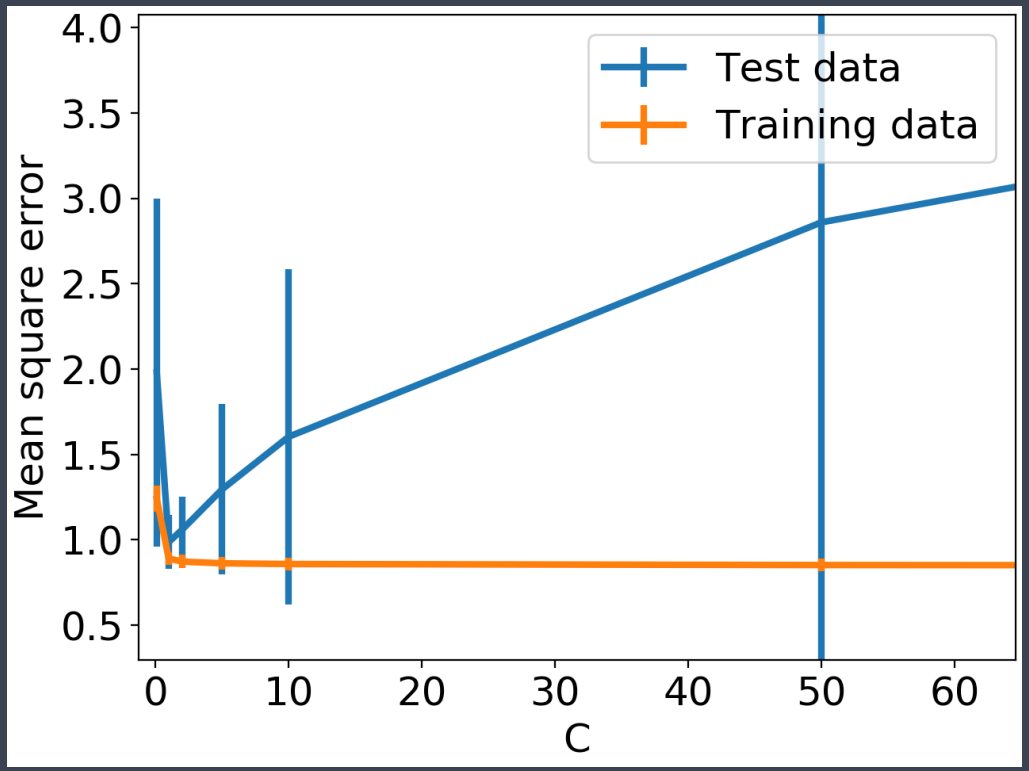

- Choosing $q$ too small or too large increases the prediction error *on the test data* but not on the training data. *Why?*
- Notice the test data error bars become large for larger $q$ (high-order polynomials tend to be badly behaved)
- Costs for $q = 1$, 2 or 3 look about the same, but $q = 2$ has smaller error bars. That's normal - often the best choice isn't that clear and some judgement is needed.

# » Tuning Model Hyperparameters

Let's combine both. Use polynomial features with $q = 6$ and add ridge regression penalty

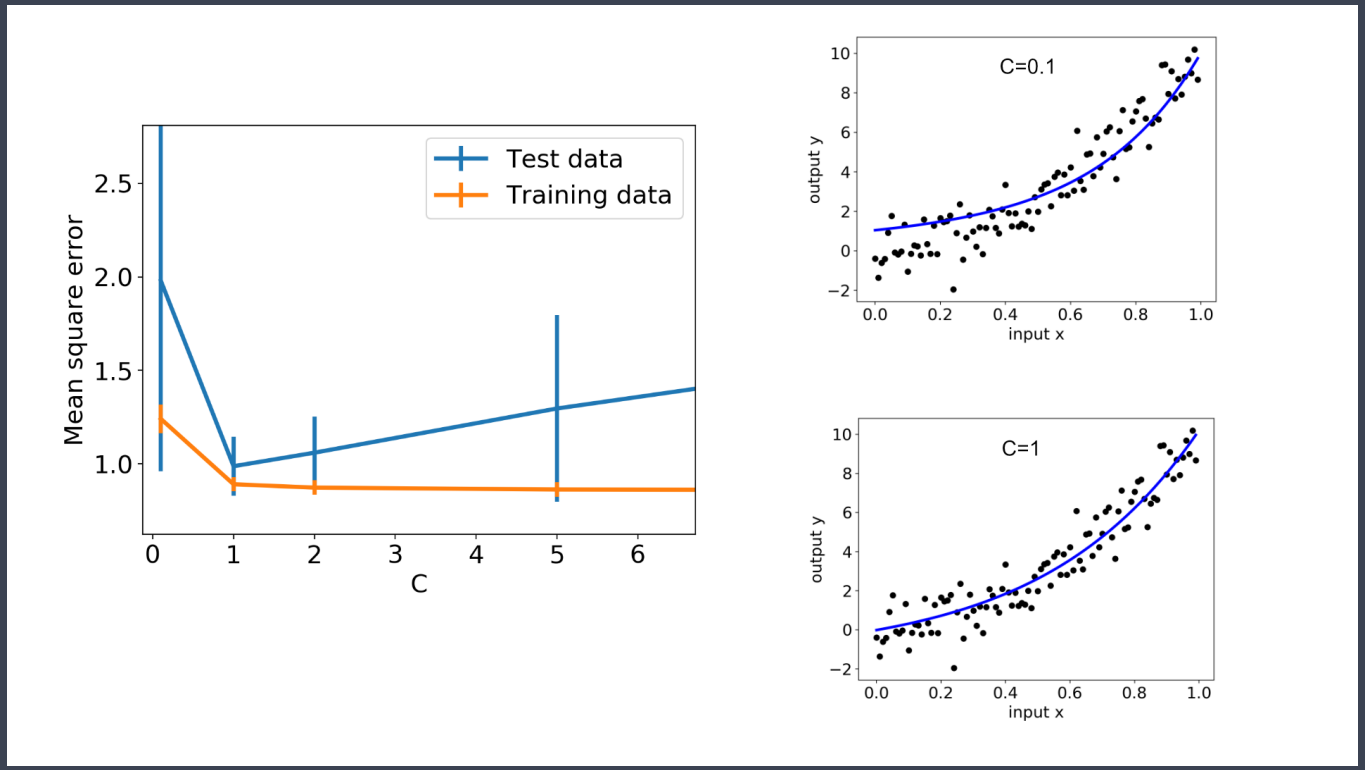
- Ridge regression cost function:
  $J(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 + \theta^T \theta / C$
- Use feature $[1,\ x,\ x^2,\ x^3,\ x^4,\ x^5,\ x^q]$
- Scan across a range of values for $C$

# » Tuning Model Hyperparameters




- Remember increasing $C$ decreases contribution of $\theta^T\theta/C$ penalty to cost, so tend to revert to previous linear regression behaviour as $C$ gets large.
- Predictions for training data always improves as $C$ made larger, but predictions for test data start getting worse
- Bearing in mind the error bars, values of $C$ around 1 look like a reasonable choice.

# » Tuning Model Hyperparameters




* Taking error bars into account, choice of *C* around 1 looks ok

# » Overfitting and Underfitting

- As our model gets more rich/complex we start to fit the "noise" in the training data, called *overfitting*. Predictions for new data become poor.
- E.g. use of polynomial features with $q = 6$ when data is quadratic
- If our model is too simple it is not able to capture the behaviour of the data, called *underfitting*. Again, predictions become poor.
- E.g. use of $q = 1$ (purely linear model) when data is quadratic
- Striking the right balance between over-fitting and under-fitting is a key task in supervised machine learning, and is intrinsic to all supervised learning approaches i.e it can't be avoided.

# » Model Selection

There are two main approaches to model selection (both can be used together):

* **Sequential Model Selection**:
  *repeat* {
  Add a new feature, fit model, use cross-val to evaluate
  } *until* predictions start getting worse or improvement is small.
* **Regularisation**:
  Change the cost function to add a penalty e.g. in linear regression change cost fn to $\frac{1}{m}\sum_{i=1}^{m}(\theta^T x^{(i)} - y^{(i)})^2 + \frac{1}{C}\sum_{j=1}^{n}\theta_j^2$

  Here $\frac{1}{C}\sum_{j=1}^{n}\theta_j^2$ is the **penalty** term, decreasing $C$ makes this term bigger, increasing $Ca$ makes it smaller (when $C = \infty$ then $\frac{1}{C} = 0$ and we are back to original setup with no penalty).

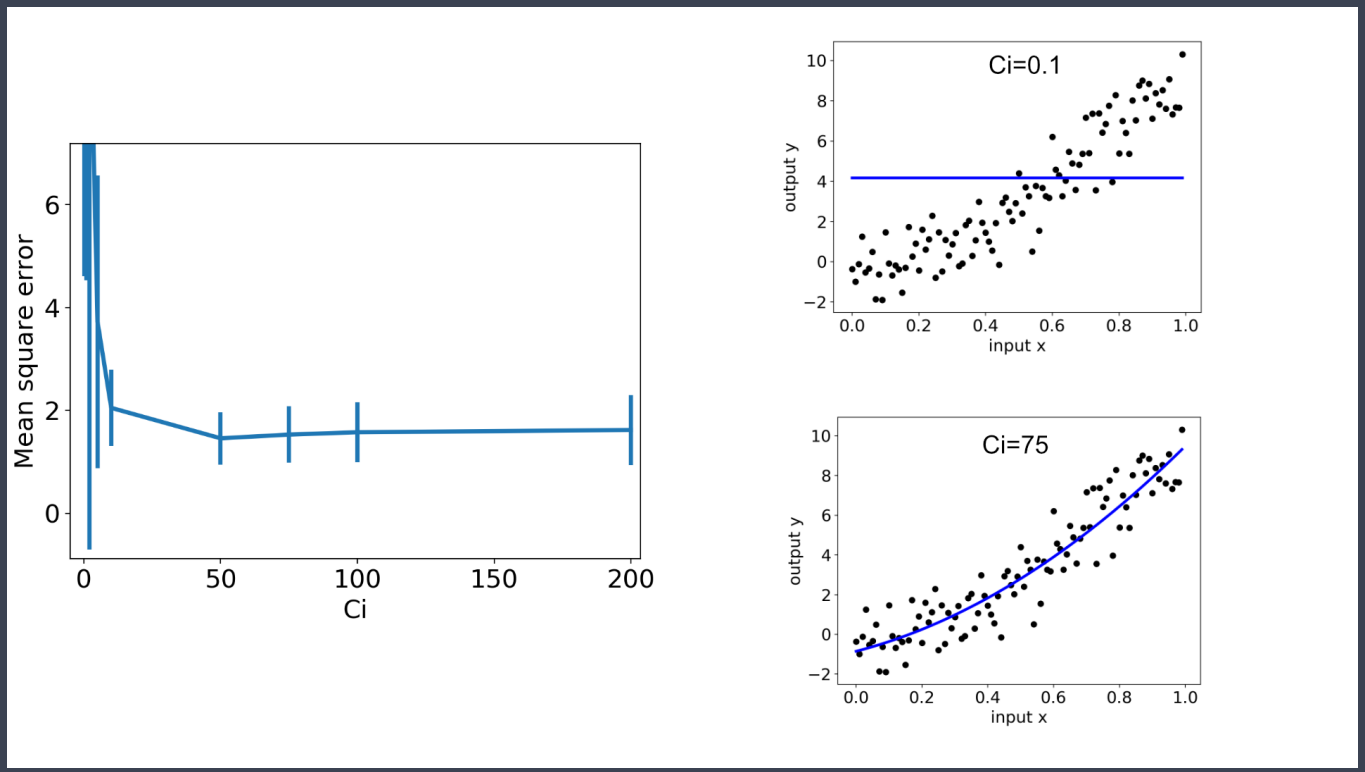
# » Regularisation

Two common regularisation penalties:

- Quadratic/L2 penalty: $R(\theta) = \theta^T\theta = \sum_{j=1}^{n} \theta_j^2$. Also called Tikhonov regularisation. Encourages elements of $\theta$ to have small value.
  - Adding quadratic penalty to linear regression cost function $\rightarrow$ ridge regression, see above.
  - A quadratic penalty is always included in SVM cost function
  - Can add quadratic penalty to logistic regression too.
- L1 penalty: $R(\theta) = \sum_{j=1}^{n} |\theta_j|$.
  - Encourages sparsity of solution i.e. few non-zero elements in $\theta$.

# » LASSO Regression

* $J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 + \frac{1}{C}\sum_{j=1}^{n}|\theta_j|$.




* When $C = 0.1$ the model parameters $\theta = [0, 0, 0, 0, 0, 0, 0]$
* When $C = 75$ typical $\theta = [0, 0, 8.02371592, 1.65407384, 0, 0, 0]$
* Observe that L1 penalty tends to make as many elments of $\theta$ zero as possible.