Table of Contents

In [7]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import matplotlib as mpl
from cycler import cycler

plt.style.use('default')

font_size = 12

mpl.rcParams['axes.prop_cycle'] = cycler('color', ['#003057', '#EAAA00
', '#4B8B9B', '#B3A369', '#377117', '#1879DB', '#8E8B76', '#002233', '#
F5D580'])
mpl.rcParams['axes.titlesize'] = font_size
mpl.rcParams['axes.titleweight'] = 'ultralight'
mpl.rcParams['axes.labelsize'] = font_size
mpl.rcParams['axes.labelweight'] = 'ultralight'

mpl.rcParams['xtick.labelsize'] = font_size
mpl.rcParams['xtick.direction'] = 'in'

mpl.rcParams['ytick.labelsize'] = font_size
mpl.rcParams['ytick.left'] = True
mpl.rcParams['ytick.direction'] = 'in'

mpl.rcParams['lines.linewidth'] = 2
mpl.rcParams['lines.linestyle'] = '--'
#mpl.rcParams['lines.marker'] = 'o'

mpl.rcParams['figure.titlesize'] = font_size
mpl.rcParams['figure.titleweight'] = 'bold'
mpl.rcParams['figure.figsize'] = (8, 6)
mpl.rcParams['figure.dpi'] = 300
mpl.rcParams['figure.autolayout'] = True

mpl.rcParams['savefig.dpi'] = 300
mpl.rcParams['savefig.format'] = 'svg'
mpl.rcParams['savefig.transparent'] = True

mpl.rcParams['font.size'] = font_size
mpl.rcParams['font.family'] = 'sans-serif'
mpl.rcParams['font.sans-serif'] = 'Helvetica'
mpl.rcParams['font.style'] = 'normal'

mpl.rcParams['mathtext.default'] = 'regular'

mpl.rcParams['legend.fontsize'] = font_size - 3
```

Complexity Optimization

The key to machine learning is creating models that generalize to new examples. This means we are looking for models with enough complexity to describe the behavior, but not so much complexity that it just reproduces the data points.

- Underfitting: The model is just "guessing" at the data, and will be equally bad at the data it has been trained on and the data that it is tested on.
- Overfitting: The model has memorized all of the training data, and will be perfect on training data and terrible on testing data.
- Optimal complexity: The model has *learned* from the training data and can *generalize* to the training data. The performance should be approximately as good for both sets.

Consider the general form of a machine-learning model introduced earlier:

$\vec{y} = f(\vec{x}, \vec{w}(\vec{\eta}))$

The "complexity" of a model is defined by its hyperparameters ($\vec{\eta}$). The goal of machine learning is to **optimize the complexity** of a model so that it **generalizes to new examples**. In order to achieve this goal we first need a way to quantify complexity so that we can optimize it.

In general there are a few strategies:

- Number of parameters: "Complexity" varies linearly with number of parameters
- Information criteria: "Complexity" varies with number of parameters and is balanced by the model error.
- "Smoothness": "Complexity" is related to the maximum curvature of the model

> *"With four parameters I can fit an elephant, and with five I can make him wiggle his trunk."*
> - John von Neumann -

(see an example here (https://www.johndcook.com/blog/2011/06/21/how-to-fit-an-elephant/))

Information Criteria

The idea behind an "information criterion" is that it quantifies the tradeoff between the number of parameters and the model error. The most commonly used information criterion is the "Bayesian Information Criterion", or BIC. The derivation of the BIC is beyond the scope of this course, but conceptually a lower BIC corresponds to a *more* probable model.

If we assume that our error is normally distributed, the BIC can be easily computed as:

$ BIC = n\times\ln{(\sigma^2_e)} + k\times\ln(n)$

where n is the number of data points, σ_e is the standard deviation of the error, and k is the number of parameters.

There are a few other "information critera", with the Akaike Information Criterion, or AIC, being the other most commonly used. For now we will just consider the BIC, but they typically yield similar optimal models.

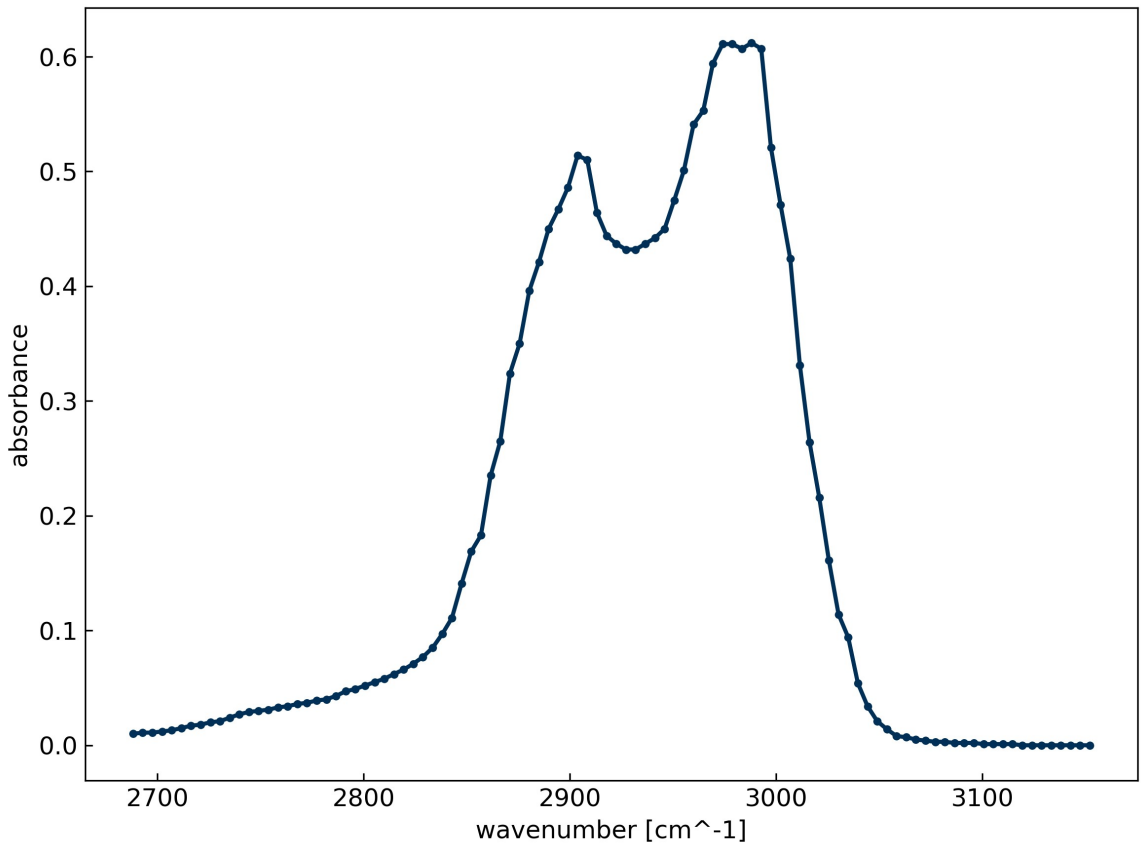
Let's implement the BIC in Python:

In [8]:
```
def BIC(y, yhat, k):
    err = y - yhat
    sigma = np.std(np.real(err))
    n = len(y)
    B = n*np.log(sigma**2) + k*np.log(n)
    return B
```

Now we will apply it to models with the spectra dataset:

In [9]:
```
import numpy as np
import pandas as pd

df = pd.read_csv('data/ethanol_IR.csv')
x_all = df['wavenumber [cm^-1]'].values
y_all = df['absorbance'].values

x_peak = x_all[475:575]
y_peak = y_all[475:575]

fig, ax = plt.subplots()
ax.plot(x_peak,y_peak, '-', marker='.')
ax.set_xlabel('wavenumber [cm^-1]')
ax.set_ylabel('absorbance');
```

```
findfont: Font family ['sans-serif'] not found. Falling back to DejaV
u Sans.
findfont: Font family ['sans-serif'] not found. Falling back to DejaV
u Sans.
```



Now, let's compare some of the many different models we have used for modeling the spectrum from the previous module and this module. We will look at the following models:

- Polynomial regression with 40 polynomials (40 parameters)
- Gaussian regression 20 evenly-spaced Gaussians (20 parameters)

We will re-implement the polynomial and Gaussian regressions using `scikit-learn` to make things easier:

In [10]:
```
from sklearn.linear_model import LinearRegression

def polynomial_features(x, N):
    # function to return a matrix of polynomials for x to order N
    # One-liner uses "list comprehension" to iterate through range 0 - N (note N+1 since range function is not inclusive)
    # The input, x, is raised to the power of N for each value of N
    # The result is converted to an array and transposed so that columns correspond to features and rows correspond to data points (individual x values)
    return np.array([x**k for k in range(0,N)]).T

N = 40
X_poly = polynomial_features(x_peak, N)

LR_poly = LinearRegression() #create a linear regression model instance
LR_poly.fit(X_poly, y_peak) #fit the model
yhat_poly = LR_poly.predict(X_poly)

BIC_poly = BIC(y_peak, yhat_poly, N)
BIC_poly
```

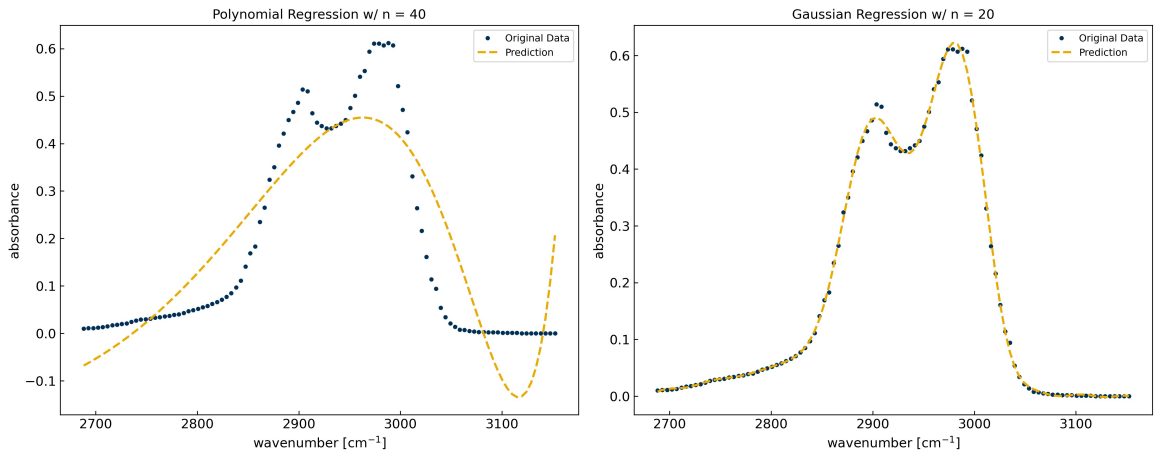
Out[10]:
```
-280.054905258297
```

In [11]:
```
def gaussian_features(x, N , sigma = 25):
    # x is a vector
    # sigma is the standard deviation
    xk_vec = np.linspace(min(x), max(x), N)
    features = []
    for xk in xk_vec:
        features.append(np.exp(-((x - xk)**2/(2*sigma**2))))
    return np.array(features).T

N = 20
X_gauss = gaussian_features(x_peak, N)

LR_gauss = LinearRegression() #create a linear regression model instanc
e
LR_gauss.fit(X_gauss, y_peak) #fit the model
yhat_gauss = LR_gauss.predict(X_gauss)

BIC_gauss = BIC(y_peak, yhat_gauss, N)
BIC_gauss
```

Out[11]:
```
-896.2526132005682
```

In [12]:
```
fig, axes = plt.subplots(1,2, figsize = (15, 6))
axes[0].plot(x_peak, y_peak, '.')
axes[1].plot(x_peak, y_peak, '.')

axes[0].plot(x_peak, yhat_poly, '--')
axes[1].plot(x_peak, yhat_gauss, '--')

for ax in axes:
    ax.set_xlabel('wavenumber [$cm^{-1}$]')
    ax.set_ylabel('absorbance')
    ax.legend(['Original Data', 'Prediction'])

axes[0].set_title('Polynomial Regression w/ n = 40')
axes[1].set_title('Gaussian Regression w/ n = 20');
```

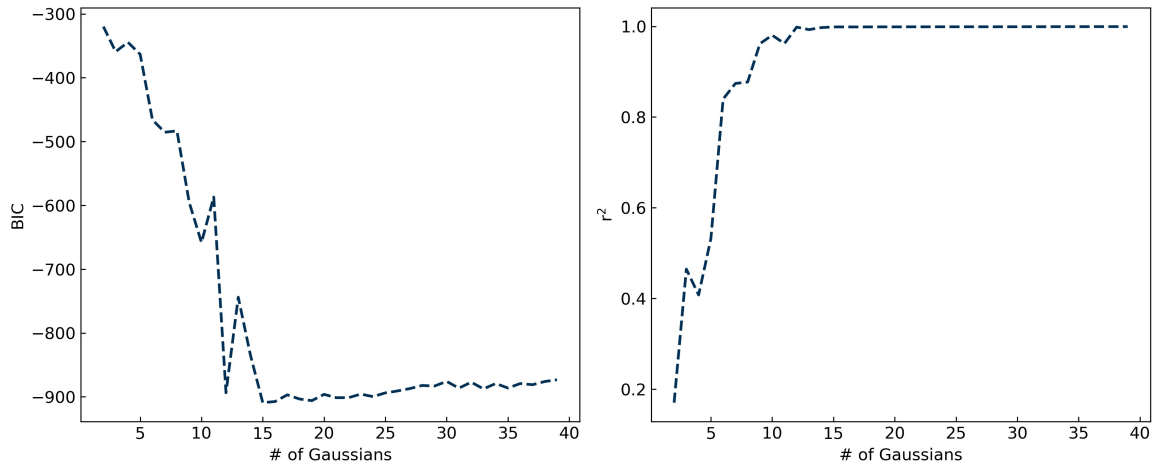
```
findfont: Font family ['sans-serif'] not found. Falling back to DejaV
u Sans.
```



We can see that the BIC correctly predicts that the Gaussian model is preferred.

Exercise: Use the BIC to determine the optimum number of evenly-spaced Gaussians for the spectra

In [14]:
```
N = 20
BICs = []
R2s = []
for N in range(2, 40):
    X_gauss = gaussian_features(x_peak, N)
    LR_gauss = LinearRegression()
    LR_gauss.fit(X_gauss, y_peak)
    yhat_gauss = LR_gauss.predict(X_gauss)

    BIC_gauss = BIC(y_peak, yhat_gauss, N)
    BICs.append(BIC_gauss)
    R2s.append(LR_gauss.score(X_gauss, y_peak))

fig, axes = plt.subplots(1, 2, figsize = (12, 5))
axes[0].plot(range(2, 40), BICs)
axes[0].set_xlabel('# of Gaussians')
axes[0].set_ylabel('BIC')
axes[1].plot(range(2, 40), R2s)
axes[1].set_xlabel('# of Gaussians')
axes[1].set_ylabel('r$^{2}$')
plt.show()
```



There are numerous other "information criteria" that can be used in a similar way. Another common one is the "Aikake information criterion":

$ AIC = 2\times(\ln{(\sigma^2_e)} + k)$

This is derived from a slightly different set of statistical assumptions. There are other criteria as well, and you can find them in the statistical literature if needed. However, it can often be very challenging to determine if your dataset fits the assumptions of the criteria, and in practice it is common to just use BIC. However, without knowing the details of the statistical assumptions these are not rigorous and should not be used as a substitute for common sense or intuition. However, the general idea of balancing number of parameters and model error provides a solid framework for thinking about complexity optimization.

One other challenge with information criteria is that non-parametric models have parameters that are not defined in the same way (the parameters and their values change depending on the training data). This makes it difficult (or impossible) to apply information critera with non-parametric models.

Regularization

Another way of penalizing complexity is by trying to penalize models that change very sharply. This is achieved by adding a penalty for parameters with very large values in the loss function. For example:

$L = \sum_i \epsilon_i^2 + \alpha \sqrt{\sum_j w_j^2}$

In this case, we introduce a new hyperparameter, α, which controls the strength of regularization. We also choose to regularize on the square root of the sum of squared parameters, which is often called the "L2 norm" and written as:

$L = \sum_i \epsilon_i^2 + \alpha ||\vec{w}||_2$

We can also regularize in other ways, which can have advantages in some cases. We will discuss this more later, but will focus on the L2 norm for now.

Discussion: Why is the "smoothness" of a model related to the size of its parameters?

> Take the second derivative. The larger w$_{j}$, the larger the derivative would be.

Regularization is especially critical in the case of non-parametric models, where the number of parameters is always greater than the number of data points. If we use a kernel and regularize on the sum of squared parameters it is called **Kernel Ridge Regression**, or KRR. We will not derive the equations here, but it can be done analytically (Hint: You should think about how you would do this).

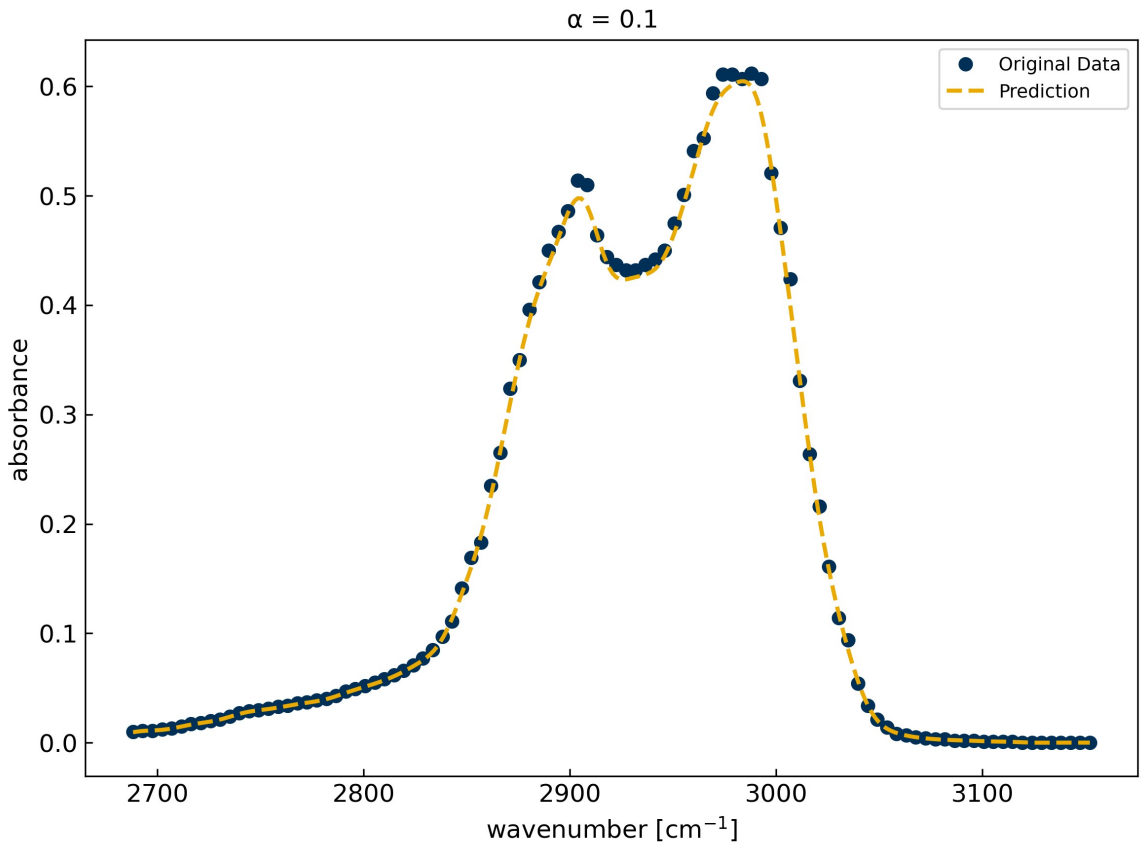
We will just use the `scikit-learn` implementation for now:

In [18]:
```
from sklearn.kernel_ridge import KernelRidge
#help(KernelRidge)
```

If we look at the parameters we see that we need to specify which kernel to use (we will use `rbf`), the `gamma` value corresponding to the width of the kernel, and `alpha` corresponding to the regularization strength. You are already familiar with the `rbf` kernel and `gamma` from the non-parametric models lecture. The only new thing here is the regularization strength, `alpha`.

In [19]:
```
sigma = 10
gamma = 1./(2*sigma**2)

alpha = 0.1

KRR = KernelRidge(alpha=alpha, kernel='rbf', gamma=gamma)
x_peak = x_peak.reshape(-1,1) #we need to convert these to columns
y_peak = y_peak.reshape(-1,1)

KRR.fit(x_peak, y_peak)

x_predict = np.linspace(min(x_peak), max(x_peak), 300) #create predicti
on data
yhat_KRR = KRR.predict(x_predict)

fig, ax = plt.subplots()
ax.plot(x_peak, y_peak, 'o')
ax.plot(x_predict, yhat_KRR, '--', markerfacecolor='none')
ax.set_xlabel('wavenumber [$cm^{-1}$]')
ax.set_ylabel('absorbance')
ax.legend(['Original Data', 'Prediction'])
ax.set_title(r'$\alpha$ = {}'.format(alpha))
```

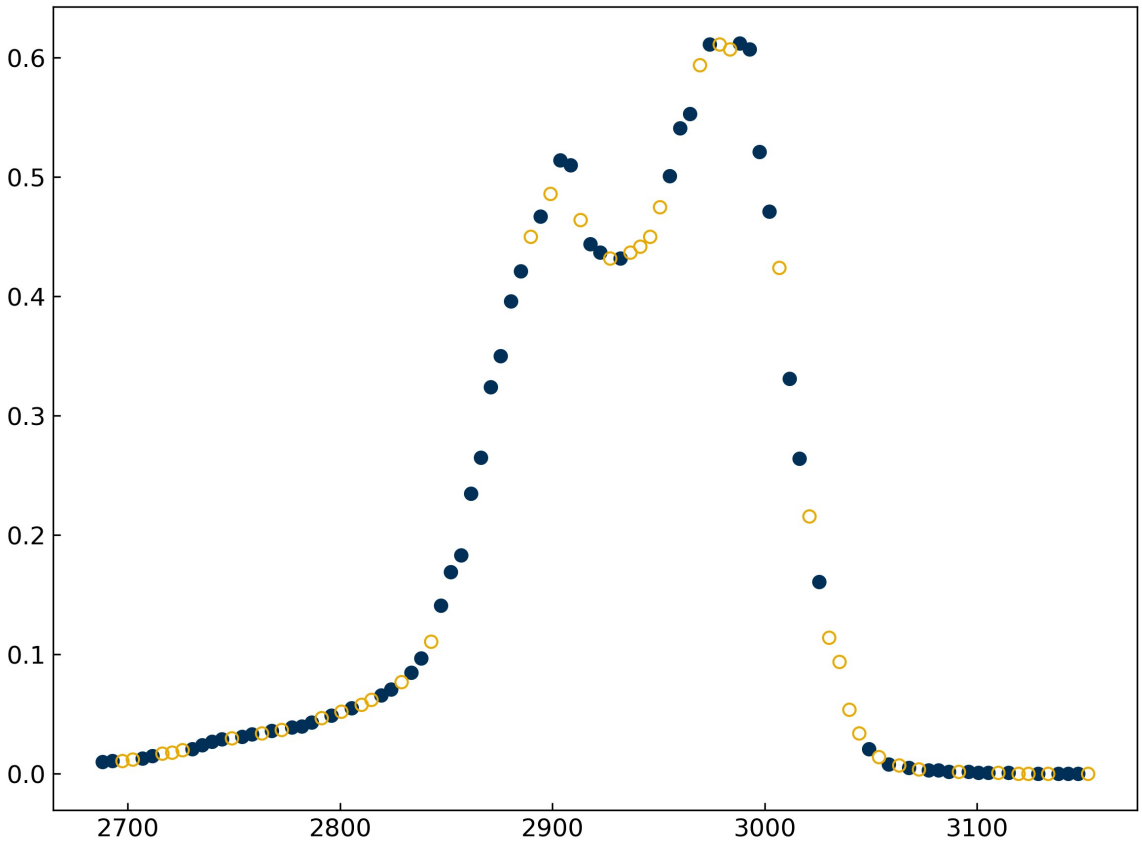
Out[19]:
```
Text(0.5, 1.0, '$\\alpha$ = 0.1')
```



Discussion: What happens as $\alpha \rightarrow$ 0 and $\alpha \rightarrow$ ∞?

We see that the regularization clearly affects the model, but sometimes it seems to make it worse. We need some strategy for assessing what value of regularization to choose. We can go back to the idea of cross-validation to achieve this:

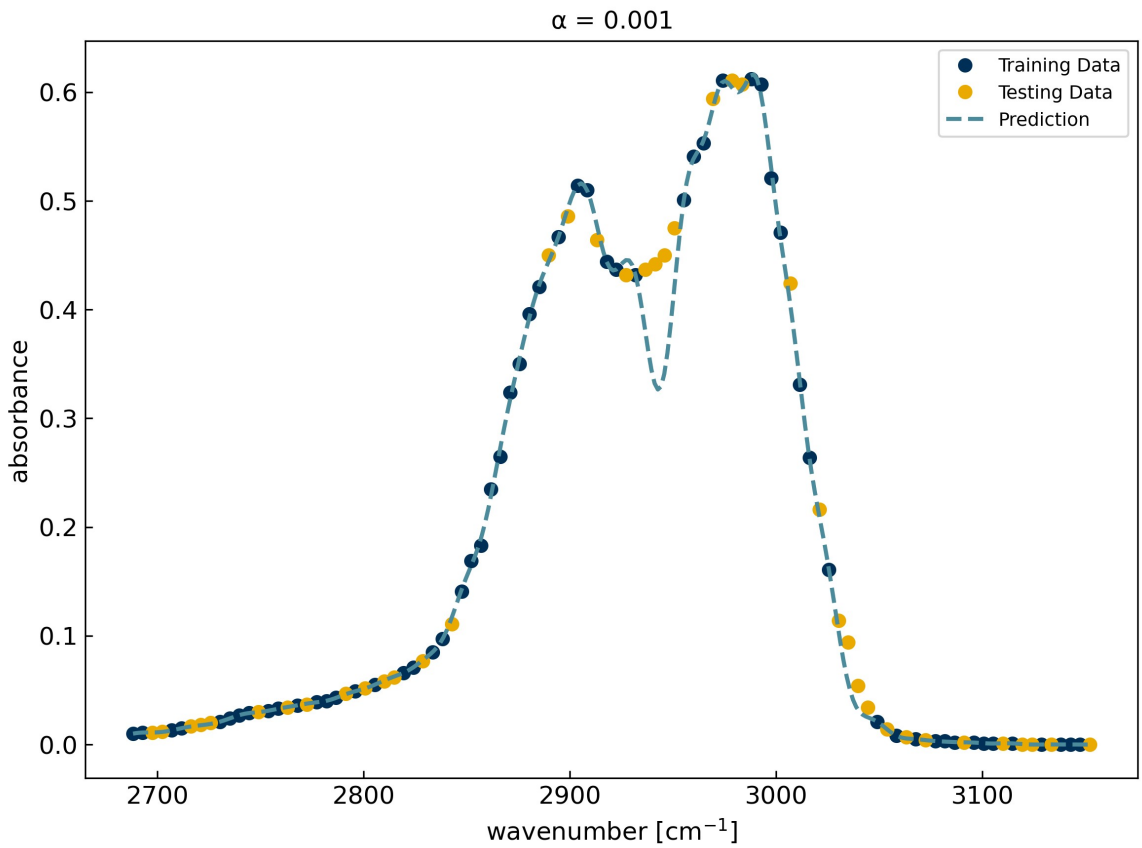
In [20]:
```
from sklearn.model_selection import train_test_split
np.random.seed(0)

x_train, x_test, y_train, y_test = train_test_split(x_peak, y_peak, tes
t_size=0.4)

fig, ax = plt.subplots()

ax.plot(x_train, y_train, 'o')
ax.plot(x_test, y_test, 'o', markerfacecolor='none');
```



We can use hold out to compute the error on the testing data as we vary the regularization strength:

In [21]:
```
sigma = 10
gamma = 1. / 2 / sigma**2

alpha = 1e-3

KRR = KernelRidge(alpha=alpha, kernel='rbf', gamma=gamma)
KRR.fit(x_train, y_train)

x_predict = np.linspace(min(x_peak), max(x_peak), 300) #create predicti
on data
yhat_KRR = KRR.predict(x_predict)

r2_test = KRR.score(x_test, y_test)
print(r2_test)

fig, ax = plt.subplots()
ax.plot(x_train, y_train, 'o')
ax.plot(x_test, y_test, 'o');
ax.plot(x_predict, yhat_KRR, '--', markerfacecolor='none')
ax.set_xlabel('wavenumber [$cm^{-1}$]')
ax.set_ylabel('absorbance')
ax.legend(['Training Data', 'Testing Data', 'Prediction'])
ax.set_title(r'$\alpha$ = {}'.format(alpha));
```

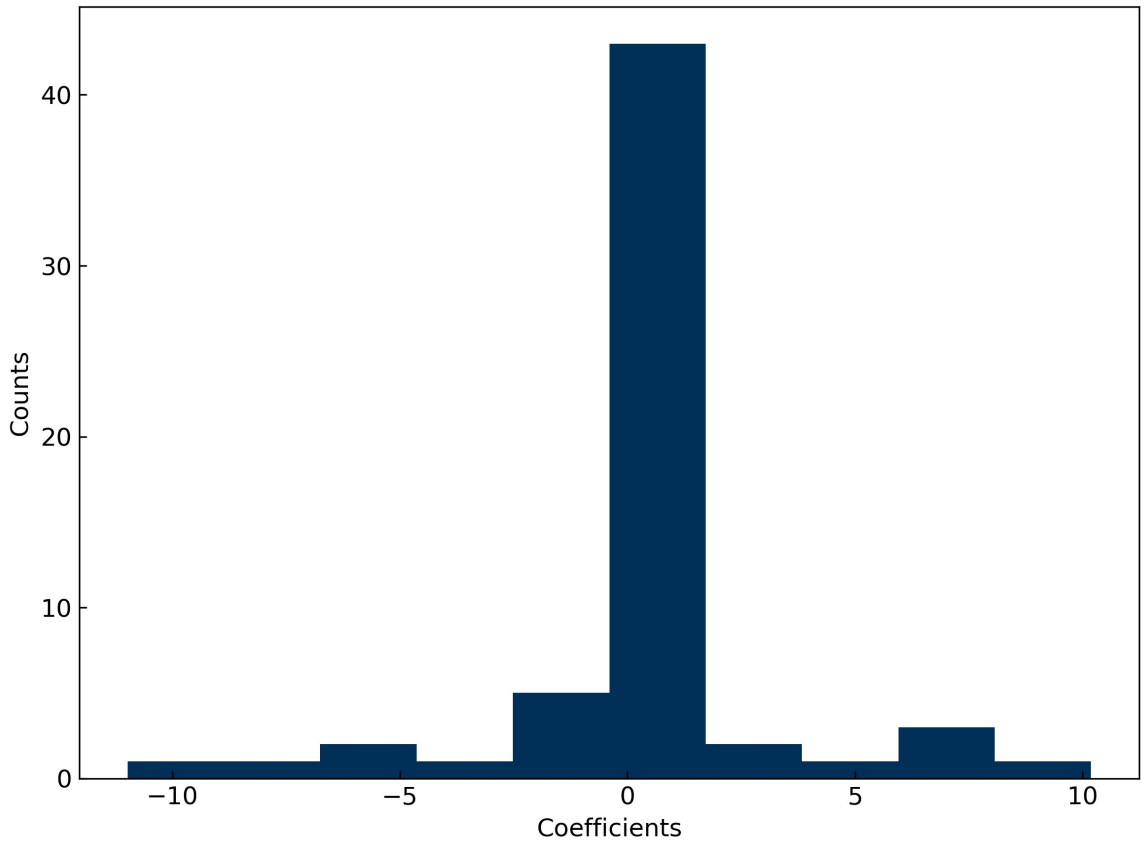
```
0.9816791895375319
```



You can also see how the regularization affects the parameters, $\vec{w}$, by looking at the (not intuitively named) `dual_coef_` attribute of the KRR model:

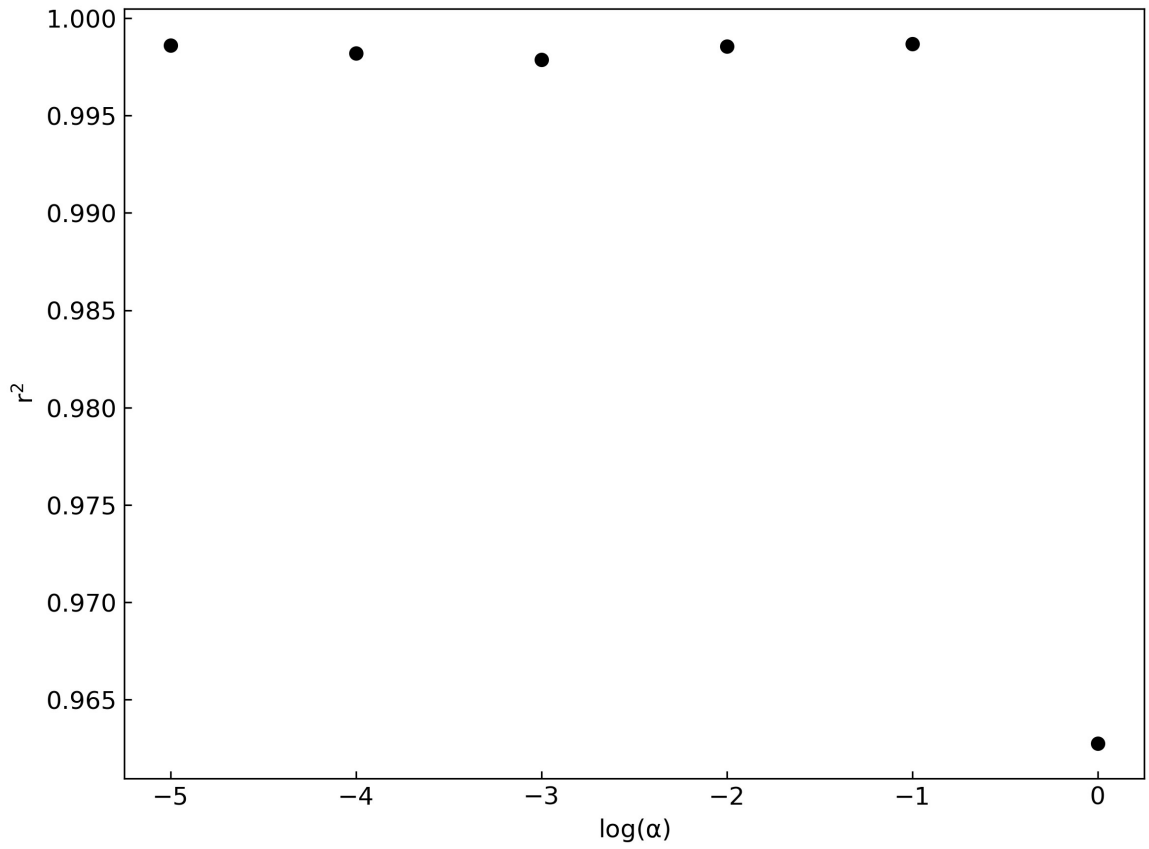
In [22]:
```
coeffs= KRR.dual_coef_
print('The model has {} coefficients.'.format(len(coeffs)))

fig, ax = plt.subplots()
ax.hist(coeffs)
ax.set_xlabel('Coefficients')
ax.set_ylabel('Counts')
print('The largest coefficient is {:.3f}.'.format(max(abs(coeff
s))[0]));
```

```
The model has 60 coefficients.
The largest coefficient is 10.984.
```



Exercise: Use cross validation to determine the optimal value of α when $\sigma=20$.

In [31]:
```
sigma = 20
gamma = 1. / 2 / sigma**2

alphas = [1e-5, 1e-4, 1e-3, 1e-2, 1e-1, 1]
r2s = []

for alpha in alphas:
    KRR = KernelRidge(alpha = alpha, kernel = 'rbf', gamma = gamma)
    KRR.fit(x_train, y_train)

    x_predict = np.linspace(min(x_peak), max(x_peak), 300)
    yhat_KRR = KRR.predict(x_predict)

    r2_test = KRR.score(x_test, y_test)
    r2s.append(r2_test)

fig, ax = plt.subplots()
log_alpha = np.log10(alphas)

ax.plot(log_alpha, r2s, 'ok')
ax.set_xlabel(r'log($\alpha$)')
ax.set_ylabel('r$^{2}$')
plt.show()
```



LASSO Regularization

Ridge regression provides a good way to penalize model "smoothness", but it doesn't actually reduce the number of parameters. We can see that all of the coefficients are non-zero:

In [24]:
```
nonzero = [f for f in np.isclose(coeffs,0) if f == False]
print('Total number of non-zero parameters: {}'.format(len(nonzero)))
```

```
Total number of non-zero parameters: 60
```

Ideally we could also use regularization to reduce the number of parameters. It turns out that this can be achieved using the L1 norm:

$||L_1|| = \sum_i |w_i|$

where $|.|$ is the absolute value. This is called "least absolute shrinkage and selection operator" regression, which is a terrible name with a great acronym: LASSO. The loss function for LASSO is defined as:

$L_{LASSO} = \sum_i \epsilon_i^2 + \alpha ||\vec{w}||_1$

This can be compared to the loss function for ridge regression:

$L_{ridge} = \sum_i \epsilon_i^2 + \alpha ||\vec{w}||_2$

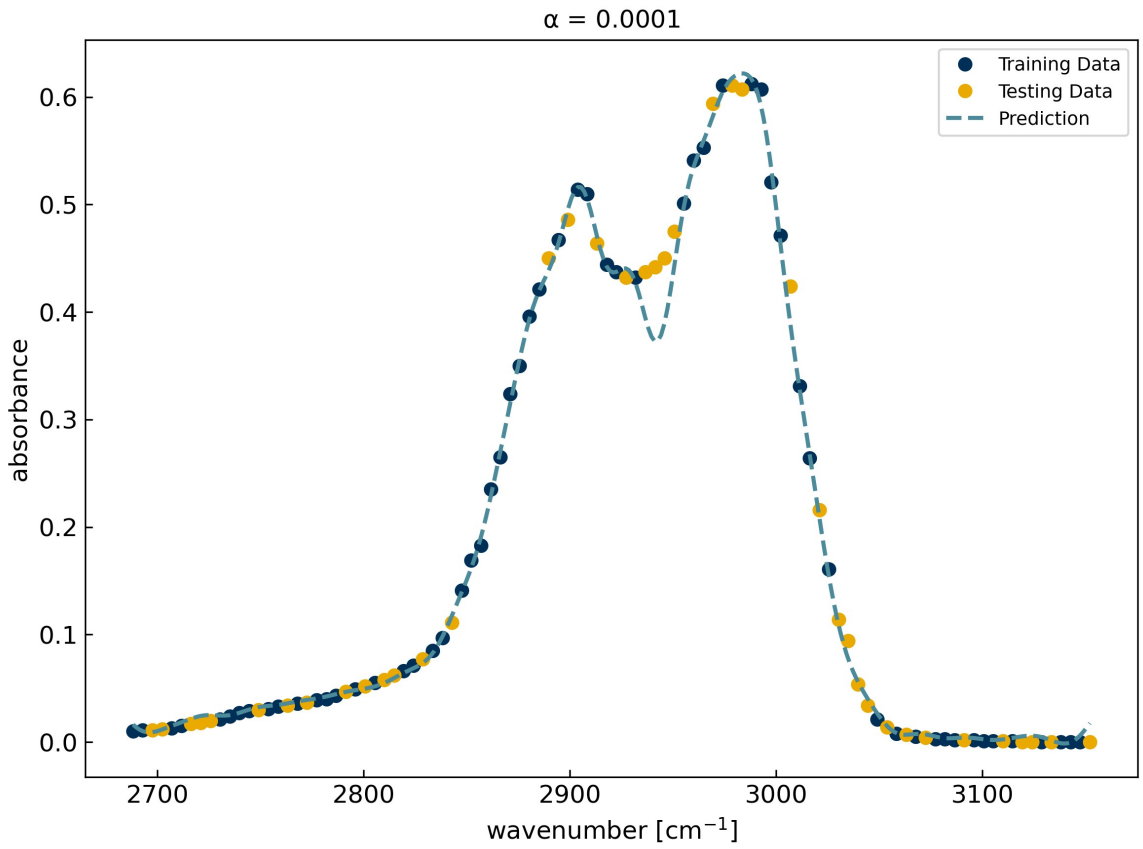
We will not go through the derivation of *why* the L1 norm causes parameters to go to zero, but the following schematic, borrowed from [this website (https://niallmartin.wordpress.com/2016/05/12/shrinkage-methods-ridge-and-lasso-regression/)](https://niallmartin.wordpress.com/2016/05/12/shrinkage-methods-ridge-and-lasso-regression/) may be useful (note that $\vec{\beta}$ is equivalent to $\vec{w}$:

We can also test it using `scikit-learn`. Unfortunately, we need to create our own feature (basis) matrix, X_{ij}, similar to linear regression, so we will need a function to evaluate the `rbf`. Instead of using our own, we can use the one from `scikit-learn`:

In [25]:
```
from sklearn.metrics.pairwise import rbf_kernel

sigma = 10
gamma = 1./(2*sigma**2)

X_train = rbf_kernel(x_train, x_train, gamma=gamma)
```

In [26]:
```
from sklearn.linear_model import Lasso

sigma = 10
gamma = 1./(2*sigma**2)

alpha = 1e-4

LASSO = Lasso(alpha=alpha)
LASSO.fit(X_train, y_train)
print(len(LASSO.coef_))

x_predict = np.linspace(min(x_peak), max(x_peak), 300) #create predicti
on data
X_predict = rbf_kernel(x_predict, x_train, gamma=gamma)

yhat_LASSO = LASSO.predict(X_predict)

fig, ax = plt.subplots()
ax.plot(x_train, y_train, 'o')
ax.plot(x_test, y_test, 'o')
ax.plot(x_predict, yhat_LASSO, '--')
ax.set_xlabel('wavenumber [$cm^{-1}$]')
ax.set_ylabel('absorbance')
ax.legend(['Training Data', 'Testing Data', 'Prediction'])
ax.set_title(r'$\alpha$ = {}'.format(alpha));
```

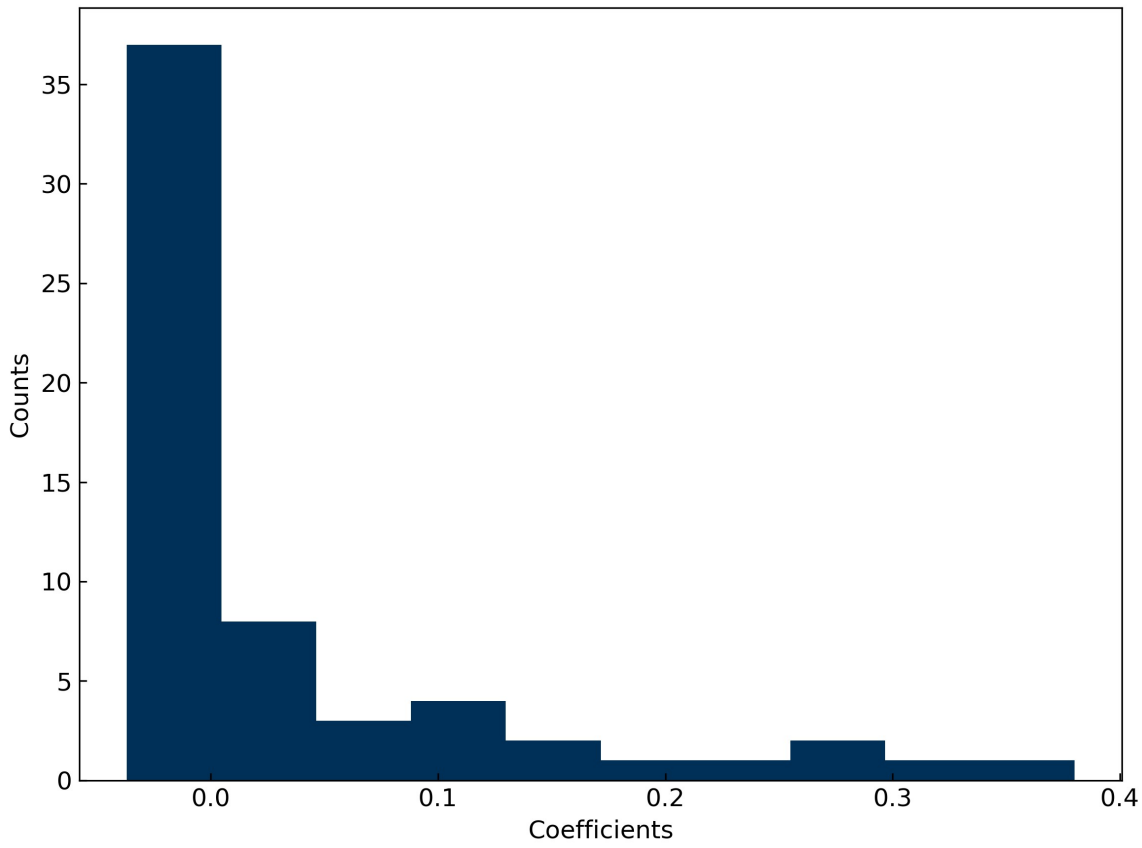
```
60
```



The results look similar to KRR. Now we can see how many non-zero parameters there are, and check the parameter values:

In [27]:
```
coeffs = LASSO.coef_

fig, ax = plt.subplots()
ax.hist(coeffs)
ax.set_xlabel('Coefficients')
ax.set_ylabel('Counts')

nonzero = [f for f in np.isclose(coeffs,0) if f == False]
print('Total number of non-zero parameters: {}'.format(len(nonzero)))
```

```
Total number of non-zero parameters: 41
```



We see that the LASSO regularization has a lot of coefficients that are equal to zero. This is equivalent to discarding these terms and finding which Gaussians should (or should not) be included.

Hyperparameter Tuning

The KRR and LASSO models above have 2 hyperparameters: γ $\left(=\frac{1}{2\sigma^2}\right)$ and α. So far, we have optimized α, but the model performance (and optimal α) will also depend on σ. You can probably see that optimizing these will get rather tedious.

Fortunately, `scikit-learn` has some nice built-in tools to help. The most commonly used is `GridSearchCV`, which is a brute-force approach that searches over a grid of hyperparameters, and uses cross-validation at each grid point to assess model performace.

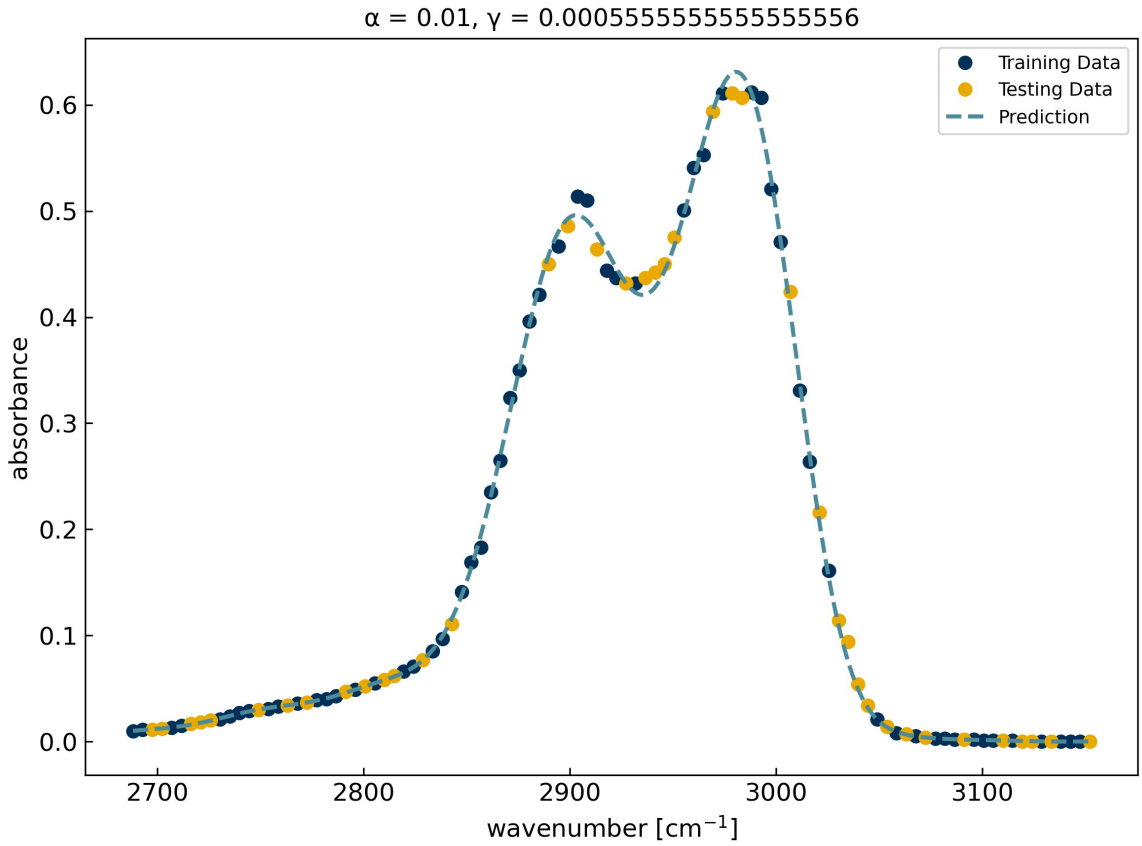
Here we will use GridSearchCV to find the optimum KRR model and its score (related to R^2):

In [33]:
```
from sklearn.model_selection import GridSearchCV

sigmas = np.array([5, 10, 15, 20, 25, 30,35, 40])
gammas = 1./(2*sigmas**2)

alphas = np.array([1e-9, 1e-5, 1e-4,1e-3, 1e-2,1e-1, 1])

parameter_ranges = {'alpha':alphas, 'gamma':gammas}

KRR = KernelRidge(kernel='rbf')

KRR_search = GridSearchCV(KRR, parameter_ranges, cv=3)
KRR_search.fit(x_train,y_train)
KRR_search.best_estimator_, KRR_search.best_score_
```

Out[33]:
```
(KernelRidge(alpha=0.01, gamma=0.0005555555555555556, kernel='rbf'),
 0.9953287405140007)
```

This tells us that the best performance comes from a model with $\alpha=0.01$ and $\gamma=0.000555$. We can check the performance of the model:

In [29]:
```
yhat_KRR = KRR_search.best_estimator_.predict(x_predict)

fig, ax = plt.subplots()
ax.plot(x_train, y_train, 'o')
ax.plot(x_test, y_test, 'o')
ax.plot(x_predict, yhat_KRR, '--', markerfacecolor='none')
ax.set_xlabel('wavenumber [$cm^{-1}$]')
ax.set_ylabel('absorbance')
ax.legend(['Training Data', 'Testing Data', 'Prediction'])
ax.set_title(r'$\alpha$ = {}, $\gamma$ = {}'.format(KRR_search.best_est
imator_.alpha, KRR_search.best_estimator_.gamma));
```



This is much faster than doing all the work yourself!

One note is that the best model will depend on the parameters you search over, as well as the cross-validation strategy. In this case, `cv=3` means that the model performs 3-fold cross-validation at each gridpoint.

Exercise: Optimize the hyperparameters of a LASSO model for the spectrum data

Search over the same values of α and σ as for KRR above, and use 3-fold cross validation.

Note: You will need to use a for loop over the σ values. Use `GridSearchCV.best_score_` as accuracy metric.

In [30]:
```
sigmas = np.array([5,10, 15, 20,25,30,35,40])
gammas = 1./(2*sigmas**2)

alphas = np.array([1e-9, 1e-5, 1e-4, 1e-3, 1e-2,1e-1, 1, 10, 100, 100
0])

#insert code here
```

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]:

In [ ]: