Why should we care about overfitting?

Learning Objective: This tutorial illustrates the idea of overfitting and why we should care about it.

Prediction Question: predict movie rating using review sentiment

Note: the codes for this notebook is unimportant to the class, just make sure you understand the graphs and the idea of overfitting.

In [1]:
```
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings("ignore")
```

In [2]:
```
# read in the training data
df = pd.read_csv('movie_review_train_500.csv')
df.head(2)
```

Out[2]:

	Unnamed: 0	Index	User_name	Date	Title	Rating	Spoilers	Content	Helpful	sentiment
0	0	1	yupman	2 January 2018	The Last Jedi was just magical	1	yes	SPOILER: This movie was just magical.The Force...	1,137/1,332	0.018911
1	1	2	shoresk-37122	14 February 2018	I made an account just to say how disappointed...	3	yes	This didn't feel like Star Wars. Now, I know p...	529/620	0.112811

In [3]:
```
# say we have a variable called sentiment that evaluates the sentiment of a review
# let's say we only use the sentiment value to predict review rating
 # convert both columns to numpy arrays for better graphing
x = df['sentiment'].to_numpy()
y = df['Rating'].to_numpy()
```

In [4]:
```
# scatter plot of review sentiment and rating
import matplotlib.pyplot as plt
%matplotlib inline

plt.figure(1)
# plot the (x,y) points with dots of size 7
plt.scatter(x,y,s=7,)
plt.title("Review Rating at each sentiment score")
plt.xlabel("Review sentiment")
plt.ylabel("Rating")
plt.autoscale(tight=True)
plt.grid(True, linestyle='-', color='0.75') #draw a slightly opaque, dashed grid

ax = plt.gca()
ax.set_ylim([1, 10])

plt.show()
```

We will attempt to fit several polynomials to this set of data. But first let's define a way to measure error

i.e. how off are we in our prediction with the polynomial vs. the actuals. The error function below is just the sum of squared differences (SSE).

- the smaller the error is, the better "goodness-of-fit" that our model gets

In [5]:
```
def error(f,x,y):
    return np.sum((f(x)-y)**2)
```

Let's fit a 1st order polynomial (i.e., linear regression)

In [6]:
```
fp1, residuals, rank, sv, rcond = np.polyfit(x,y,1, full=True)
# The polyfit() function returns the parameters of the fitted model function, fp1

print("Model parameters: %s" % fp1)
print("Error: %s" %residuals)
```

```
Model parameters: [9.53024928 1.36054969]
Error: [362.73101894]
```

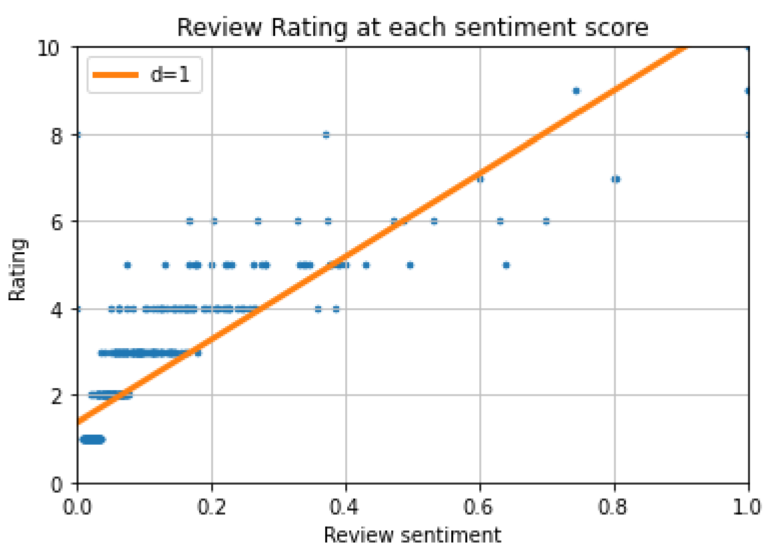
We can see that we fit a polynomial of order 1 with the parameters 9.53 and 1.36; This is a straight line of the form

- y = 9.53x + 1.36
- The error is 362.73

We can now form this line using generated x values and then plot it on the same graph to observe the fit in the scatter plot

In [7]:
```
# graph model f1 onto the graph
# use poly1d() to create a polynomial function from the model parameters.
f1 = np.poly1d(fp1)
print("1st degree polynomial error is ", error(f1,x,y))

plt.figure(2)
# plot the (x,y) points with dots of size 7
plt.scatter(x,y,s=7,)
plt.title("Review Rating at each sentiment score")
plt.xlabel("Review sentiment")
plt.ylabel("Rating")
plt.autoscale(tight=True)
plt.grid(True, linestyle='-', color='0.75') #draw a slightly opaque, dashed grid

#set y axis
ax = plt.gca()
ax.set_ylim([0, 10])

#adding a straight line into figure 2
fx = np.linspace(0,1,100) #generate x-values for plotting
plt.plot(fx, f1(fx), linewidth=3, color = 'tab:orange')
plt.legend(["d=%i" % f1.order], loc="upper left")
plt.show()
```

```
1st degree polynomial error is  362.7310189425334
```



Let's try fitting a second order polynomial

In [8]:
```
f2p = np.polyfit(x,y,2)
print(f2p)
f2 = np.poly1d(f2p)
print(error(f2,x,y))
```

```
[-9.81812004 16.86957061  0.98093818]
259.65646775874427
```

The quadratic that we fit is

- y = -9.82x^2 + 16.87 x + 0.98
- Note The error here is 259.65, which is smaller than the error of linear regression (362.73), indicating that f2 is a better model than f1 in our training data

In [9]:
```
plt.figure(3)
print("1st degree polynomial error is ", error(f1,x,y))
print("2nd degree polynomial error is ", error(f2,x,y))
# plot the (x,y) points with dots of size 7
plt.scatter(x,y,s=7,)
plt.title("Review Rating at each sentiment score")
plt.xlabel("Review sentiment")
plt.ylabel("Rating")
plt.autoscale(tight=True)
plt.grid(True, linestyle='-', color='0.75') #draw a slightly opaque, dashed grid

ax = plt.gca()
ax.set_ylim([0, 10])

#adding a straight line into figure 2
fx = np.linspace(0,1,100) #generate x-values for plotting
plt.plot(fx, f1(fx), linewidth=3)
plt.legend(["d=%i" % f1.order], loc="upper left")

#adding the quadratic function into figure 2
plt.plot(fx, f2(fx), linewidth=3)
plt.legend(["d=%i" % f1.order,"d=%i" % f2.order], loc="upper left")
```

```
1st degree polynomial error is  362.7310189425334
2nd degree polynomial error is  259.65646775874427
```

Out[9]:
```
<matplotlib.legend.Legend at 0x7fb6f79b3850>
```

Indeed, visually we see that f2 seems to fit the data better than f1

What if we were to try more complex model by fitting polynomials functions higher orders

let's try 3 and 10 in this case:

In [10]:

```
# plot f3 and f10 on the graph
f3p = np.polyfit(x,y,3)
f3 = np.poly1d(f3p)

f10p = np.polyfit(x,y,10)
f10 = np.poly1d(f10p)

print("Error order 1: ", error(f1,x,y))
print("Error order 2: ",error(f2,x,y))
print("Error order 3: ",error(f3,x,y))
print("Error order 10: ",error(f10,x,y))


#plotting the initial scatter plot
plt.figure(4)

# plot the (x,y) points with dots of size 7
plt.scatter(x,y,s=7,)
plt.title("Review Rating at each sentiment score")
plt.xlabel("Review sentiment")
plt.ylabel("Rating")
plt.autoscale(tight=True)
plt.grid(True, linestyle='-', color='0.75') #draw a slightly opaque, dashed grid

ax = plt.gca()
ax.set_ylim([0, 10])

#adding a straight line into figure 2
fx = np.linspace(0,1,100) #generate x-values for plotting
plt.plot(fx, f1(fx), linewidth=3)
plt.legend(["d=%i" % f1.order], loc="upper left")

#adding the polynomials of order 2, 3 and 10
plt.plot(fx, f2(fx), linewidth=3)
plt.plot(fx, f3(fx), linewidth=3)
plt.plot(fx, f10(fx), linewidth=3)
plt.legend(["d=%i" % f1.order,"d=%i" % f2.order,"d=%i" % f3.order,"d=%i" % f10.order], loc="upper left")
```

```
Error order 1:  362.7310189425334
Error order 2:  259.65646775874427
Error order 3:  196.41606734571735
Error order 10:  143.51596744908235
```

Out[10]: <matplotlib.legend.Legend at 0x7fb6f5f03bb0>

- We can see that the fit is getting better mathematically. The calculated errors are decreasing as well.
- This could only be a good thing right? - Even though the model is mathematically better, we see that visually the 10th degree polynomial looks a bit crazy.

What happens when we keep increasing our polynomial order. Let's go all the way up to order 99.

In [11]:

```
f99p = np.polyfit(x,y,99)
f99 = np.poly1d(f99p)

#plotting the initial scatter plot
plt.figure(5)


# plot the (x,y) points with dots of size 7
plt.scatter(x,y,s=7,)
plt.title("Review Rating at each sentiment score")
plt.xlabel("Review sentiment")
plt.ylabel("Rating")
plt.autoscale(tight=True)
plt.grid(True, linestyle='-', color='0.75') #draw a slightly opaque, dashed grid

ax = plt.gca()
ax.set_ylim([0, 10])

#adding a straight line into figure 2
fx = np.linspace(0,1,100) #generate x-values for plotting
plt.plot(fx, f1(fx), linewidth=3)
#plt.legend(["d=%i" % f1.order], loc="upper left")

#adding the polynomials of order 2, 3 and 10

plt.plot(fx, f99(fx), linewidth=3)
plt.legend(["d=%i" % f1.order,"d=%i" % f99.order], loc="upper left")

print("Error of order 1: ",error(f1,x,y))
print("Error of order 99: ",error(f99,x,y))
```

```
Error of order 1:  362.7310189425334
Error of order 99:  105.21530177295622
```

Judging from the error, it seems like f99 is way better than f1.

- Just from the training data, it seems that the more complicated our model is, the better performance it gets.
- If this is true, we should use the "f99" model for the rating prediction.
- but what happens when we fit our models onto the test data?

In [12]:
```
test = pd.read_csv('movie_review_test_100.csv')
test.head(2)
```

Out[12]:

	Unnamed: 0	Index	User_name	Date	Title	Rating	Spoilers	Content	Helpful	sentiment
0	0	3901	stephanebenyamin	4 January 2018	So bad I made a video with explanations	2	no	The characters and the plot lines have many is...	1/2	0.173107
1	1	3902	drharding	4 January 2018	NO IDEALS OR CHARACTER! NO INTEGRITY PERIOD!	1	no	Go to hell D(sney. I am boycotting this film a...	1/2	0.000000

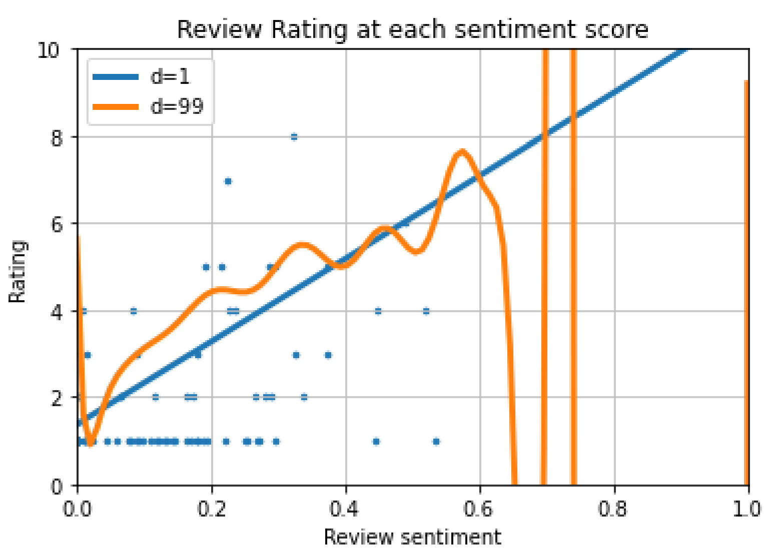
- Note that in the "Rating" variable is the ground truth, that we don't observe when training

In [13]:
```
x_test = test['sentiment'].to_numpy()
y_test = test['Rating'].to_numpy()
```

Let's apply our "worst" and "best" models, f1 and f99, on to the test data

In [14]:
```
# plot the (x,y) points with dots of size 7
plt.scatter(x_test,y_test,s=7,)
plt.title("Review Rating at each sentiment score")
plt.xlabel("Review sentiment")
plt.ylabel("Rating")
plt.autoscale(tight=True)
plt.grid(True, linestyle='-', color='0.75') #draw a slightly opaque, dashed grid

ax = plt.gca()
ax.set_ylim([0, 10])

#adding a straight line into figure 2
fx = np.linspace(0,1,100) #generate x-values for plotting
plt.plot(fx, f1(fx), linewidth=3)
plt.legend(["d=%i" % f1.order], loc="upper left")

#adding the polynomials of order 2, 3 and 10

plt.plot(fx, f99(fx), linewidth=3)
plt.legend(["d=%i" % f1.order,"d=%i" % f99.order], loc="upper left")

print("Error of order 1: ",error(f1,x_test,y_test))
print("Error of order 99: ",error(f99,x_test,y_test))
```

```
Error of order 1:  252.9163429565317
Error of order 99:  1078.7484853710052
```



Here we see that model f99 is performing way worse than model f1!

- Why? Because model f99 tries to account for the outliers in the training data, and these specific outliers don't exist in the test data
- On the other hand, model f1, albeit simple, does better in predicting the overall trend of the data
- Here we have a typical case of overfitting.

In [ ]:

```
```

How can we test for overfitting?

split our data into training set and validation set, and see if our model is consistent across different splits

In [15]:

```
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LinearRegression
# prepare data: turn data to dataframe or matrix
X = x.reshape(-1, 1) #
y = y.reshape(-1, 1)
```

let's split the training data into 80% train and 20% validation

- we train the model on X_train and y_train
- then validate the model on X_test and y_test and calculate the accuracy

In [17]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=0)
```

In [55]:

```
# let's try to fit a linear regression model
reg = LinearRegression().fit(X_train, y_train)
reg.score(X_test, y_test)
```

Out[55]:

```
0.6280703825965999
```

by changing the random_state option, we randomly sample data again

In [56]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=1)
reg = LinearRegression().fit(X_train, y_train)
reg.score(X_test, y_test)
```

Out[56]: 0.7381105328372455

In [57]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=2)
reg = LinearRegression().fit(X_train, y_train)
reg.score(X_test, y_test)
```

Out[57]: 0.7411472509940018

In [58]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=3)
reg = LinearRegression().fit(X_train, y_train)
reg.score(X_test, y_test)
```

Out[58]: 0.5707009731977994

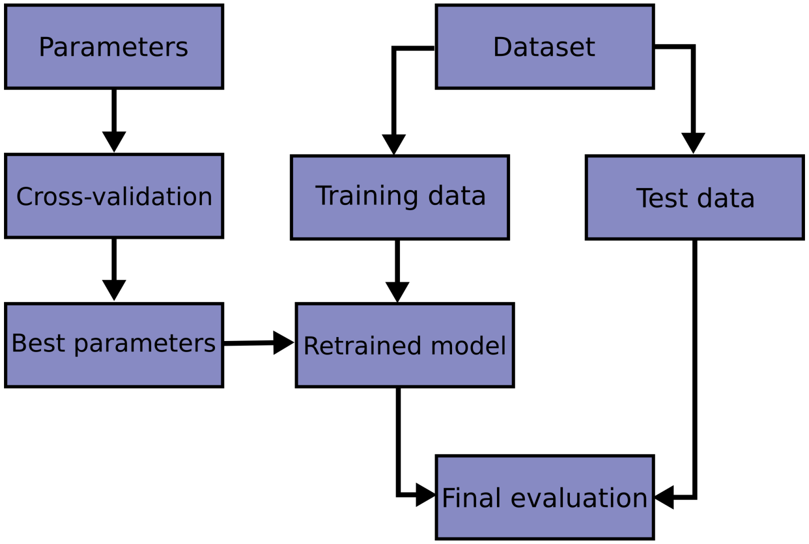
In [59]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=4)
reg = LinearRegression().fit(X_train, y_train)
reg.score(X_test, y_test)
```

Out[59]: 0.7701400781699572

by randomly splitting a bunch of times, we ensure that our model has ~ 60-70% accuracy

In [65]:
```
from IPython.display import Image
Image("https://scikit-learn.org/stable/_images/grid_search_workflow.png", width = 400)
```

Out[65]:



Cross validation

- corss validation is a common technique to test for overfitting
- it does the above step for us by splitting the training data into k folds, and each time using k-1 of the folds as the training data, and 1 fold as the test data, repeating this k times. This is called "k-fold-cross-validation"
- all we have to specify is the number of cross-validation
- here let's try a 5-fold-cv

In [72]:
```
scores = cross_val_score(reg, X, y, cv=5)
scores
```

Out[72]:
```
array([0.69615985, 0.63132167, 0.7096076 , 0.58980489, 0.78215211])
```

In [73]:
```
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores.mean(), scores.std()))
```

```
0.68 accuracy with a standard deviation of 0.07
```

In [94]:
```
# let's try a quadratic function
# this cell of code fits a linear model with 2 degress of polynomial
from sklearn.preprocessing import PolynomialFeatures
from sklearn.pipeline import make_pipeline

reg2 = make_pipeline(PolynomialFeatures(2), LinearRegression())
reg2.fit(X_train, y_train)
reg2.score(X_test, y_test)
```

Out[94]:
```
0.8451142562883324
```

Here we see that the f2 is a better model than linear regression, even after testing for overfitting

In [92]:
```
# cross validation on f2
scores2 = cross_val_score(reg2, X, y, cv=5)
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores2.mean(), scores2.std()))
```

```
0.77 accuracy with a standard deviation of 0.07
```

Here we see that f99 drastically overfits the model:

- i.e., the standard deviation high, indicating the f99 model is not stable

In [93]:
```
reg99 = make_pipeline(PolynomialFeatures(99), LinearRegression())
reg99.fit(X_train, y_train)
scores99 = cross_val_score(reg99, X, y, cv=5)
print("%0.2f accuracy with a standard deviation of %0.2f" % (scores99.mean(), scores99.std()))
```

```
-3136239950446.21 accuracy with a standard deviation of 6240750674088.55
```

In [ ]:

In [ ]: