

Assignment 2

August 30, 2017

*You are currently looking at **version 1.2** of this notebook. To download notebooks and datafiles, as well as get help on Jupyter notebooks in the Coursera platform, visit the [Jupyter Notebook FAQ](#) course resource.*

1 Assignment 2

In this assignment you'll explore the relationship between model complexity and generalization performance, by adjusting key parameters of various supervised learning models. Part 1 of this assignment will look at regression and Part 2 will look at classification.

1.1 Part 1 - Regression

First, run the following block to set up the variables needed for later sections.

```
In [2]: import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
np.random.seed(0)
n = 15
x = np.linspace(0,10,n) + np.random.randn(n)/5
y = np.sin(x)+x/6 + np.random.randn(n)/10
X_train, X_test, y_train, y_test = train_test_split(x, y, random_state=0)

# You can use this function to help you visualize the dataset by
# plotting a scatterplot of the data points
# in the training and test sets.
def part1_scatter():
    %matplotlib notebook
    plt.figure()
    plt.scatter(X_train, y_train, label='training data')
    plt.scatter(X_test, y_test, label='test data')
    plt.legend(loc=4);
    #plt.show()
```

```
# NOTE: Uncomment the function below to visualize the data, but be sure
# to **re-comment it before submitting this assignment to the autograder*.
#part1_scatter()
```

1.1.1 Question 1

Write a function that fits a polynomial LinearRegression model on the *training data* `X_train` for degrees 1, 3, 6, and 9. (Use `PolynomialFeatures` in `sklearn.preprocessing` to create the polynomial features and then fit a linear regression model) For each model, find 100 predicted values over the interval $x = 0$ to 10 (e.g. `np.linspace(0,10,100)`) and store this in a numpy array. The first row of this array should correspond to the output from the model trained on degree 1, the second row degree 3, the third row degree 6, and the fourth row degree 9.

The figure above shows the fitted models plotted on top of the original data (using `plot_one()`).

This function should return a numpy array with shape (4, 100)

```
In [3]: from sklearn.linear_model import LinearRegression
        from sklearn.preprocessing import PolynomialFeatures
        def predict_poly(n):
            X = x[:, np.newaxis]
            poly = PolynomialFeatures(degree=n)
            X_poly = poly.fit_transform(X)
            X_train, X_test, y_train, y_test = train_test_split\
                (X_poly, y, random_state = 0)
            linreg = LinearRegression().fit(X_train, y_train)
            x_test = np.linspace(0,10,100)
            X_test = x_test[:, np.newaxis]
            X_poly_test = poly.fit_transform(X_test)
            y_test = linreg.predict(X_poly_test)
            y_res = y_test[:, np.newaxis]
            return np.transpose(y_res)

        def answer_one():
            # Your code here
            res = predict_poly(1)
            y_test = predict_poly(3)
            res = np.concatenate((res, y_test))
            y_test = predict_poly(6)
            res = np.concatenate((res, y_test))
            y_test = predict_poly(9)
            res = np.concatenate((res, y_test))
            return res
        answer_one()
```

```
Out[3]: array([[ 2.53040195e-01,  2.69201547e-01,  2.85362899e-01,
                  3.01524251e-01,  3.17685603e-01,  3.33846955e-01,
                  3.50008306e-01,  3.66169658e-01,  3.82331010e-01,
```

3.98492362e-01,	4.14653714e-01,	4.30815066e-01,
4.46976417e-01,	4.63137769e-01,	4.79299121e-01,
4.95460473e-01,	5.11621825e-01,	5.27783177e-01,
5.43944529e-01,	5.60105880e-01,	5.76267232e-01,
5.92428584e-01,	6.08589936e-01,	6.24751288e-01,
6.40912640e-01,	6.57073992e-01,	6.73235343e-01,
6.89396695e-01,	7.05558047e-01,	7.21719399e-01,
7.37880751e-01,	7.54042103e-01,	7.70203454e-01,
7.86364806e-01,	8.02526158e-01,	8.18687510e-01,
8.34848862e-01,	8.51010214e-01,	8.67171566e-01,
8.83332917e-01,	8.99494269e-01,	9.15655621e-01,
9.31816973e-01,	9.47978325e-01,	9.64139677e-01,
9.80301028e-01,	9.96462380e-01,	1.01262373e+00,
1.02878508e+00,	1.04494644e+00,	1.06110779e+00,
1.07726914e+00,	1.09343049e+00,	1.10959184e+00,
1.12575320e+00,	1.14191455e+00,	1.15807590e+00,
1.17423725e+00,	1.19039860e+00,	1.20655995e+00,
1.22272131e+00,	1.23888266e+00,	1.25504401e+00,
1.27120536e+00,	1.28736671e+00,	1.30352807e+00,
1.31968942e+00,	1.33585077e+00,	1.35201212e+00,
1.36817347e+00,	1.38433482e+00,	1.40049618e+00,
1.41665753e+00,	1.43281888e+00,	1.44898023e+00,
1.46514158e+00,	1.48130294e+00,	1.49746429e+00,
1.51362564e+00,	1.52978699e+00,	1.54594834e+00,
1.56210969e+00,	1.57827105e+00,	1.59443240e+00,
1.61059375e+00,	1.62675510e+00,	1.64291645e+00,
1.65907781e+00,	1.67523916e+00,	1.69140051e+00,
1.70756186e+00,	1.72372321e+00,	1.73988457e+00,
1.75604592e+00,	1.77220727e+00,	1.78836862e+00,
1.80452997e+00,	1.82069132e+00,	1.83685268e+00,
1.85301403e+00],		
[1.22989539e+00,	1.15143628e+00,	1.07722393e+00,
1.00717881e+00,	9.41221419e-01,	8.79272234e-01,
8.21251741e-01,	7.67080426e-01,	7.16678772e-01,
6.69967266e-01,	6.26866391e-01,	5.87296632e-01,
5.51178474e-01,	5.18432402e-01,	4.88978901e-01,
4.62738455e-01,	4.39631549e-01,	4.19578668e-01,
4.02500297e-01,	3.88316920e-01,	3.76949022e-01,
3.68317088e-01,	3.62341603e-01,	3.58943051e-01,
3.58041918e-01,	3.59558687e-01,	3.63413845e-01,
3.69527874e-01,	3.77821261e-01,	3.88214491e-01,
4.00628046e-01,	4.14982414e-01,	4.31198078e-01,
4.49195522e-01,	4.68895233e-01,	4.90217694e-01,
5.13083391e-01,	5.37412808e-01,	5.63126429e-01,
5.90144741e-01,	6.18388226e-01,	6.47777371e-01,
6.78232660e-01,	7.09674578e-01,	7.42023609e-01,
7.75200238e-01,	8.09124950e-01,	8.43718230e-01,
8.78900563e-01,	9.14592432e-01,	9.50714324e-01,

9.87186723e-01,	1.02393011e+00,	1.06086498e+00,
1.09791181e+00,	1.13499108e+00,	1.17202328e+00,
1.20892890e+00,	1.24562842e+00,	1.28204233e+00,
1.31809110e+00,	1.35369523e+00,	1.38877520e+00,
1.42325149e+00,	1.45704459e+00,	1.49007498e+00,
1.52226316e+00,	1.55352959e+00,	1.58379478e+00,
1.61297919e+00,	1.64100332e+00,	1.66778766e+00,
1.69325268e+00,	1.71731887e+00,	1.73990672e+00,
1.76093671e+00,	1.78032933e+00,	1.79800506e+00,
1.81388438e+00,	1.82788778e+00,	1.83993575e+00,
1.84994877e+00,	1.85784732e+00,	1.86355189e+00,
1.86698296e+00,	1.86806103e+00,	1.86670656e+00,
1.86284006e+00,	1.85638200e+00,	1.84725286e+00,
1.83537314e+00,	1.82066332e+00,	1.80304388e+00,
1.78243530e+00,	1.75875808e+00,	1.73193269e+00,
1.70187963e+00,	1.66851936e+00,	1.63177240e+00,
1.59155920e+00],		
[-1.99554310e-01,	-3.95192729e-03,	1.79851752e-01,
3.51005136e-01,	5.08831706e-01,	6.52819233e-01,
7.82609240e-01,	8.97986721e-01,	9.98870117e-01,
1.08530155e+00,	1.15743729e+00,	1.21553852e+00,
1.25996233e+00,	1.29115292e+00,	1.30963316e+00,
1.31599632e+00,	1.31089811e+00,	1.29504889e+00,
1.26920626e+00,	1.23416782e+00,	1.19076415e+00,
1.13985218e+00,	1.08230867e+00,	1.01902405e+00,
9.50896441e-01,	8.78825970e-01,	8.03709344e-01,
7.26434655e-01,	6.47876457e-01,	5.68891088e-01,
4.90312256e-01,	4.12946874e-01,	3.37571147e-01,
2.64926923e-01,	1.95718291e-01,	1.30608438e-01,
7.02167560e-02,	1.51162118e-02,	-3.41690366e-02,
-7.71657636e-02,	-1.13453547e-01,	-1.42666382e-01,
-1.64494044e-01,	-1.78683194e-01,	-1.85038228e-01,
-1.83421873e-01,	-1.73755533e-01,	-1.56019368e-01,
-1.30252132e-01,	-9.65507463e-02,	-5.50696232e-02,
-6.01973203e-03,	5.03325882e-02,	1.13667071e-01,
1.83611221e-01,	2.59742264e-01,	3.41589357e-01,
4.28636046e-01,	5.20322987e-01,	6.16050916e-01,
7.15183874e-01,	8.17052690e-01,	9.20958717e-01,
1.02617782e+00,	1.13196463e+00,	1.23755703e+00,
1.34218093e+00,	1.44505526e+00,	1.54539723e+00,
1.64242789e+00,	1.73537785e+00,	1.82349336e+00,
1.90604254e+00,	1.98232198e+00,	2.05166348e+00,
2.11344114e+00,	2.16707864e+00,	2.21205680e+00,
2.24792141e+00,	2.27429129e+00,	2.29086658e+00,
2.29743739e+00,	2.29389257e+00,	2.28022881e+00,
2.25656001e+00,	2.22312684e+00,	2.18030664e+00,
2.12862347e+00,	2.06875850e+00,	2.00156065e+00,
1.92805743e+00,	1.84946605e+00,	1.76720485e+00,

```

1.68290491e+00, 1.59842194e+00, 1.51584842e+00,
1.43752602e+00, 1.36605824e+00, 1.30432333e+00,
1.25548743e+00],
[ 6.79502315e+00, 4.14319965e+00, 2.23123316e+00,
9.10495361e-01, 5.49800913e-02, -4.41344739e-01,
-6.66950747e-01, -6.94943197e-01, -5.85049920e-01,
-3.85418713e-01, -1.34236348e-01, 1.38818291e-01,
4.11274949e-01, 6.66715203e-01, 8.93747233e-01,
1.08510180e+00, 1.23683958e+00, 1.34766048e+00,
1.41830612e+00, 1.45104704e+00, 1.44924675e+00,
1.41699514e+00, 1.35880424e+00, 1.27935965e+00,
1.18332161e+00, 1.07516974e+00, 9.59086200e-01,
8.38872245e-01, 7.17893445e-01, 5.99049383e-01,
4.84763838e-01, 3.76991851e-01, 2.77240389e-01,
1.86599613e-01, 1.05782065e-01, 3.51673710e-02,
-2.51496888e-02, -7.53096020e-02, -1.15638682e-01,
-1.46601154e-01, -1.68753939e-01, -1.82705104e-01,
-1.89076735e-01, -1.88472830e-01, -1.81452582e-01,
-1.68509336e-01, -1.50055280e-01, -1.26411838e-01,
-9.78055951e-02, -6.43694666e-02, -2.61487236e-02,
1.68885955e-02, 6.48374453e-02, 1.17838320e-01,
1.76057261e-01, 2.39664033e-01, 3.08809213e-01,
3.83600954e-01, 4.64082174e-01, 5.50208937e-01,
6.41830759e-01, 7.38673538e-01, 8.40325779e-01,
9.46228700e-01, 1.05567078e+00, 1.16778720e+00,
1.28156451e+00, 1.39585081e+00, 1.50937164e+00,
1.62075148e+00, 1.72854081e+00, 1.83124847e+00,
1.92737884e+00, 2.01547318e+00, 2.09415447e+00,
2.16217455e+00, 2.21846249e+00, 2.26217266e+00,
2.29273088e+00, 2.30987664e+00, 2.31369924e+00,
2.30466539e+00, 2.28363552e+00, 2.25186572e+00,
2.21099192e+00, 2.16299272e+00, 2.11012682e+00,
2.05484055e+00, 1.99964106e+00, 1.94692977e+00,
1.89879084e+00, 1.85672866e+00, 1.82134809e+00,
1.79197090e+00, 1.76618106e+00, 1.73929145e+00,
1.70372403e+00, 1.64829476e+00, 1.55739451e+00,
1.41005645e+00]]))

```

```

In [46]: # feel free to use the function plot_one() to replicate the figure
# from the prompt once you have completed question one
import matplotlib.pyplot as plt
def plot_one(degree_predictions):
    plt.figure(figsize=(10,5))
    plt.plot(X_train, y_train, 'o', label='training data', markersize=10)
    plt.plot(X_test, y_test, 'o', label='test data', markersize=10)
    for i, degree in enumerate([1,3,6,9]):
        plt.plot(np.linspace(0,10,100), degree_predictions[i],\
                 alpha=0.8, lw=2, label='degree={}'.format(degree))

```

```
plt.ylim(-1,2.5)
plt.legend(loc=4)
plt.show()
```

```
#plot_one(answer_one())
```

1.1.2 Question 2

Write a function that fits a polynomial LinearRegression model on the training data `X_train` for degrees 0 through 9. For each model compute the R^2 (coefficient of determination) regression score on the training data as well as the the test data, and return both of these arrays in a tuple.

This function should return one tuple of numpy arrays (`r2_train`, `r2_test`). Both arrays should have shape (10,)

```
In [39]: from sklearn.linear_model import LinearRegression
         from sklearn.preprocessing import PolynomialFeatures
         from sklearn.metrics.regression import r2_score

         def get_R2(n,r2_train,r2_test):
             X = x[:, np.newaxis]
             poly = PolynomialFeatures(degree=n)
             X_poly = poly.fit_transform(X)
             X_train, X_test, y_train, y_test = train_test_split\
                 (X_poly, y, random_state = 0)
             linreg = LinearRegression().fit(X_train, y_train)
             r2_train.append(linreg.score(X_train, y_train))
             r2_test.append(linreg.score(X_test, y_test))
         def answer_two():
             # Your code here
             r2_train = []
             r2_test = []
             res = ()
             for i in range(0, 10):
                 get_R2(i,r2_train,r2_test)
             res = r2_train,r2_test
             return res
         answer_two()
```

```
Out [39]: ([0.0,
            0.42924577812346632,
            0.4510998044408247,
            0.58719953687798476,
            0.91941944717693158,
            0.97578641430681867,
            0.99018233247950882,
            0.99352509278402779,
            0.99637545387745396,
            0.99803706256582003],
```

```
[-0.47808641737141788,
 -0.45237104233936676,
 -0.068569841499158901,
 0.0053310529457279987,
 0.73004942818715057,
 0.87708300916504855,
 0.9214093981277145,
 0.92021504118071573,
 0.63247943762578418,
 -0.64525446868451841])
```

1.1.3 Question 3

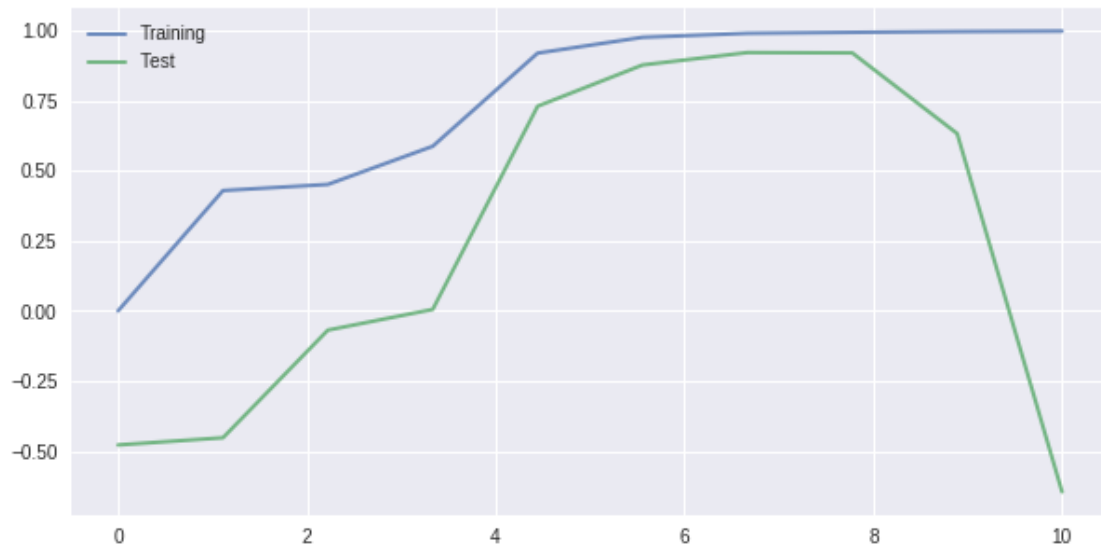
Based on the R^2 scores from question 2 (degree levels 0 through 9), what degree level corresponds to a model that is underfitting? What degree level corresponds to a model that is overfitting? What choice of degree level would provide a model with good generalization performance on this dataset? Note: there may be multiple correct solutions to this question.

(Hint: Try plotting the R^2 scores from question 2 to visualize the relationship between degree level and R^2)

This function should return one tuple with the degree values in this order: (Underfitting, Overfitting, Good_Generalization)

```
In [65]: def plot_two(R2_scores):
    plt.figure(figsize=(10,5))
    plt.plot(np.linspace(0,10,10), R2_scores[0],\
             alpha=0.8, lw=2, label="Training")
    plt.plot(np.linspace(0,10,10), R2_scores[1],\
             alpha=0.8, lw=2, label="Test")
    plt.legend()
    plt.show()

def answer_three():
    # Your code here
    R2_scores = answer_two()
    #plot_two(R2_scores)
    train = R2_scores[0][0]
    test = R2_scores[1][0]
    for i in range(1,10):
        if (R2_scores[0][i] > train and R2_scores[1][i] > test):
            train = R2_scores[0][i]
            test = R2_scores[1][i]
            under = i - 1
        if (R2_scores[0][i] >= train and R2_scores[1][i] < test):
            over = i
            break
    return tuple([under, over, i-1]) # Return your answer
answer_three()
```



([0.0, 0.42924577812346632, 0.4510998044408247, 0.58719953687798476, 0.919419447176, 0.9510998044408247, 0.98719953687798476, 0.99419447176, 1.0, 1.0])

Out[65]: (5, 7, 6)

1.1.4 Question 4

Training models on high degree polynomial features can result in overly complex models that overfit, so we often use regularized versions of the model to constrain model complexity, as we saw with Ridge and Lasso linear regression.

For this question, train two models: a non-regularized LinearRegression model (default parameters) and a regularized Lasso Regression model (with parameters $\alpha=0.01$, $\text{max_iter}=10000$) on polynomial features of degree 12. Return the R^2 score for both the LinearRegression and Lasso model's test sets.

This function should return one tuple (LinearRegression_R2_test_score, Lasso_R2_test_score)

```
In [41]: def answer_four():
    from sklearn.preprocessing import PolynomialFeatures
    from sklearn.linear_model import Lasso, LinearRegression
    from sklearn.metrics.regression import r2_score

    # Your code here
    X = x[:, np.newaxis]
    poly = PolynomialFeatures(degree=12)
    X_poly = poly.fit_transform(X)
    X_train, X_test, y_train, y_test = train_test_split(X_poly, y,
                                                         random_state = 0)
    linreg = LinearRegression().fit(X_train, y_train)
```



```

linlasso = Lasso(alpha=0.01, max_iter = 10000).fit(X_train, \
                                                    y_train)
return linreg.score(X_test, y_test), linlasso.score(X_test, y_test) # 1
answer_four()

```

```

/opt/conda/lib/python3.5/site-packages/sklearn/linear_model/coordinate_descent.py:4
ConvergenceWarning)

```

Out [41]: (-4.3119904517942373, 0.8406625614750235)

1.2 Part 2 - Classification

Here's an application of machine learning that could save your life! For this section of the assignment we will be working with the [UCI Mushroom Data Set](#) stored in `mushrooms.csv`. The data will be used to train a model to predict whether or not a mushroom is poisonous. The following attributes are provided:

Attribute Information:

1. cap-shape: bell=b, conical=c, convex=x, flat=f, knobbed=k, sunken=s
2. cap-surface: fibrous=f, grooves=g, scaly=y, smooth=s
3. cap-color: brown=n, buff=b, cinnamon=c, gray=g, green=r, pink=p, purple=u, red=e, white=w, yellow=y
4. bruises?: bruises=t, no=f
5. odor: almond=a, anise=l, creosote=c, fishy=y, foul=f, musty=m, none=n, pungent=p, spicy=s
6. gill-attachment: attached=a, descending=d, free=f, notched=n
7. gill-spacing: close=c, crowded=w, distant=d
8. gill-size: broad=b, narrow=n
9. gill-color: black=k, brown=n, buff=b, chocolate=h, gray=g, green=r, orange=o, pink=p, purple=u, red=e, white=w, yellow=y
10. stalk-shape: enlarging=e, tapering=t
11. stalk-root: bulbous=b, club=c, cup=u, equal=e, rhizomorphs=z, rooted=r, missing=?
12. stalk-surface-above-ring: fibrous=f, scaly=y, silky=k, smooth=s
13. stalk-surface-below-ring: fibrous=f, scaly=y, silky=k, smooth=s
14. stalk-color-above-ring: brown=n, buff=b, cinnamon=c, gray=g, orange=o, pink=p, red=e, white=w, yellow=y
15. stalk-color-below-ring: brown=n, buff=b, cinnamon=c, gray=g, orange=o, pink=p, red=e, white=w, yellow=y
16. veil-type: partial=p, universal=u
17. veil-color: brown=n, orange=o, white=w, yellow=y
18. ring-number: none=n, one=o, two=t
19. ring-type: cobwebby=c, evanescent=e, flaring=f, large=l, none=n, pendant=p, sheathing=s, zone=z
20. spore-print-color: black=k, brown=n, buff=b, chocolate=h, green=r, orange=o, purple=u, white=w, yellow=y
21. population: abundant=a, clustered=c, numerous=n, scattered=s, several=v, solitary=y

22. habitat: grasses=g, leaves=l, meadows=m, paths=p, urban=u, waste=w, woods=d

The data in the mushrooms dataset is currently encoded with strings. These values will need to be encoded to numeric to work with sklearn. We'll use `pd.get_dummies` to convert the categorical variables into indicator variables.

```
In [42]: import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split

mush_df = pd.read_csv('mushrooms.csv')
mush_df2 = pd.get_dummies(mush_df)

X_mush = mush_df2.iloc[:,2:]
y_mush = mush_df2.iloc[:,1]

# use the variables X_train2, y_train2 for Question 5
X_train2, X_test2, y_train2, y_test2 = train_test_split(X_mush, y_mush, ra

# For performance reasons in Questions 6 and 7, we will create a smaller v
# entire mushroom dataset for use in those questions. For simplicity we'l
# the 25% test split created above as the representative subset.
#
# Use the variables X_subset, y_subset for Questions 6 and 7.
X_subset = X_test2
y_subset = y_test2
```

1.2.1 Question 5

Using `X_train2` and `y_train2` from the preceeding cell, train a `DecisionTreeClassifier` with default parameters and `random_state=0`. What are the 5 most important features found by the decision tree?

As a reminder, the feature names are available in the `X_train2.columns` property, and the order of the features in `X_train2.columns` matches the order of the feature importance values in the classifier's `feature_importances_` property.

This function should return a list of length 5 containing the feature names in descending order of importance.

```
In [47]: def answer_five():
    from sklearn.tree import DecisionTreeClassifier
    clf = DecisionTreeClassifier().fit(X_train2, y_train2)
    impfeature = []
    sort = sorted(zip(clf.feature_importances_, X_train2.columns),
                  reverse=True)
    for i in range(0,5):
        impfeature.append(sort[i][1])
    # Your code here
```

```

        return impfeature
    answer_five()

```

```

Out[47]: ['odor_n',
          'stalk-root_c',
          'stalk-root_r',
          'spore-print-color_r',
          'spore-print-color_u']

```

1.2.2 Question 6

For this question, we're going to use the `validation_curve` function in `sklearn.model_selection` to determine training and test scores for a Support Vector Classifier (SVC) with varying parameter values. Recall that the `validation_curve` function, in addition to taking an initialized unfitted classifier object, takes a dataset as input and does its own internal train-test splits to compute results.

Because creating a validation curve requires fitting multiple models, for performance reasons this question will use just a subset of the original mushroom dataset: please use the variables `X_subset` and `y_subset` as input to the validation curve function (instead of `X_mush` and `y_mush`) to reduce computation time.

The initialized unfitted classifier object we'll be using is a Support Vector Classifier with radial basis kernel. So your first step is to create an SVC object with default parameters (i.e. `kernel='rbf'`, `C=1`) and `random_state=0`. Recall that the kernel width of the RBF kernel is controlled using the `gamma` parameter.

With this classifier, and the dataset in `X_subset`, `y_subset`, explore the effect of `gamma` on classifier accuracy by using the `validation_curve` function to find the training and test scores for 6 values of `gamma` from 0.0001 to 10 (i.e. `np.logspace(-4, 1, 6)`). Recall that you can specify what scoring metric you want `validation_curve` to use by setting the "scoring" parameter. In this case, we want to use "accuracy" as the scoring metric.

For each level of `gamma`, `validation_curve` will fit 3 models on different subsets of the data, returning two 6x3 (6 levels of `gamma` x 3 fits per level) arrays of the scores for the training and test sets.

Find the mean score across the three models for each level of `gamma` for both arrays, creating two arrays of length 6, and return a tuple with the two arrays.

e.g.

if one of your array of scores is

```

array([[ 0.5,   0.4,   0.6],
       [ 0.7,   0.8,   0.7],
       [ 0.9,   0.8,   0.8],
       [ 0.8,   0.7,   0.8],
       [ 0.7,   0.6,   0.6],
       [ 0.4,   0.6,   0.5]])

```

it should then become

```

array([ 0.5,  0.73333333,  0.83333333,  0.76666667,  0.63333333,  0.5])

```

This function should return one tuple of numpy arrays (training_scores, test_scores) where each array in the tuple has shape (6,).

```
In [44]: def answer_six():
    from sklearn.svm import SVC
    from sklearn.model_selection import validation_curve

    # Your code here
    param_range = np.logspace(-4,1,6)
    train_scores, test_scores = validation_curve(SVC(kernel = 'rbf',C = 1),
                                                param_name='gamma',
                                                param_range=param_range, cv=3,
                                                train_scores_mean = np.mean(train_scores, axis=1)
                                                test_scores_mean = np.mean(test_scores, axis=1)
    return train_scores_mean,test_scores_mean # Your answer here
answer_six()

Out[44]: (array([ 0.56647847,  0.93155951,  0.99039881,  1.          ,  1.          ,
                  0.56768547,  0.92959558,  0.98965952,  1.          ,  0.99507994,
                  0.52240279]))
```

1.2.3 Question 7

Based on the scores from question 6, what gamma value corresponds to a model that is underfitting (and has the worst test set accuracy)? What gamma value corresponds to a model that is overfitting (and has the worst test set accuracy)? What choice of gamma would be the best choice for a model with good generalization performance on this dataset (high accuracy on both training and test set)? Note: there may be multiple correct solutions to this question.

(Hint: Try plotting the scores from question 6 to visualize the relationship between gamma and accuracy.)

This function should return one tuple with the degree values in this order: (Underfitting, Overfitting, Good_Generalization)

```
In [66]: def plot_three(scores):
    plt.figure(figsize=(10,5))
    plt.plot( np.logspace(-4,1,6), scores[0],\
              alpha=0.8, lw=2, label="Training")
    plt.plot( np.logspace(-4,1,6), scores[1],\
              alpha=0.8, lw=2, label="Test")
    plt.legend()
    plt.show()

    def answer_seven():

        # Your code here
        scores = answer_six()
        #plot_three(scores)
        train = scores[0][0]
        test = scores[1][0]
        for i in range(1,6):
```

```
    if (scores[0][i] > train and scores[1][i] > test):
        train = scores[0][i]
        test = scores[1][i]
        under = i - 1
    if (scores[0][i] >= train and scores[1][i] < test):
        over = i
        break
    return tuple([under, over, i-1]) # Return your answer
answer_seven()
```

Out[66]: (2, 4, 3)

In []: