PSET 1, MACS 33002: Machine Learning

Ezra Max

01.17.2020

1. Consider data with m variables, of the form $Z = (Z^1, \ldots, Z^m)$. We assume we have access to N observations $(Z_i = (Z_i^1, \ldots, Z_i^m))_{1 \leq i \leq N}$. In both supervised and unsupervised learning, we use the known observations (Z_i) to model relationships between the variables $Z^1, \ldots, Z^m$. Theoretically, this information is contained in the joint probability density for $Z = (Z^1, \ldots, Z^m)$, $P(Z)$.

   In unsupervised learning, our goal is to model precisely this density $P(Z)$, based on the distribution of the known observations (Z_i). In supervised learning, we break Z into two components: feature data $X = (Z_{\ell_1}, \ldots, Z_{\ell_k})$, and response data $Y = (Z^{\ell_{k+1}}, \ldots, Z^{\ell_m})$. Our goal is not to model the entire relationship between the variables $Z^1, \ldots, Z^k$. Instead, we only want to model the response Y based on the feature data X.

   Statistically, choosing our features and responses in supervised learning changes and somewhat simplifies our task. Instead of modeling $P(Z)$:

   $$P(Z) = P(X, Y) = P(X)P(Y \mid X)$$

   as with unsupervised learning, we only need to model $P(Y \mid X)$, the probability density of Y conditioned on X.

   In the supervised scenario, we can model $P(Y \mid X)$ as a map $f : X \to Y$. We can evaluate the validity of a model $\hat{Y}(X) = f(X)$ by using our "true" data, the observations $(X_i, Y_i)_{1 \leq i \leq N}$. In particular, we can introduce an appropriate loss function $L(f)$ measuring the distance between our predicted response $\hat{Y}_i = \hat{Y}(X_i)$ and our observations Y_i for each $X_i, i = 1, \ldots, N$. The loss function gives a proxy for our well our model f approximates the conditional density $P(Y \mid X)$: via our loss function $L(f)$, we assess the accuracy of f by comparing predicted responses $\hat{Y}_i$ and actual responses Y_i for each feature observation X_i.

   A student's learning is "supervised" by an instructor: the instructor evaluates the student's understanding of the world by asking the student questions that the instructor knows the answer to (e.g., on a Pset or exam), and by using an appropriate measure (grades, of some sort) for how closely the student's answers align with the facts as known by the instructor. Similarly, in supervised learning we can evaluate the validity and ac-

curacy of a model f by asking it to predict a response Y_i from an observation (X_i, Y_i) as $\hat{Y_i}$, and measuring the distance (in some appropriate sense via $L(f)$) between $\hat{Y_i}$ and Y_i.

As the name might suggest, there is no such "teacher" in the unsupervised scenario: we do not choose features and response data, and thus cannot find a uniform quantitative measure for the accuracy of a model $f(Z)$ for $P(Z) = P(Z^1, \ldots, Z^n)$. Of course, in both supervised and unsupervised learning, our choice of the *class* of a model f (e.g., linear, polynomial, etc.) is often the result of educated guesswork and experience. But with supervised learning, because we have chosen our feature and response variables, we can evaluate different instances of this class/different parameters for a model f according to $L(f)$; in unsupervised learning, the accuracy of our model cannot be so easily evaluated through direct, uniform, quantitative measures.

This of course does not mean there are no well-known and even trusted methods for modeling $P(Z)$ from observed data and for assessing the validity of unsupervised learning models, or that there are no special cases in which we have good measures for the accuracy of our model (there are shades of grey here, and semi-supervised learning methods exist.) But, it does mean that in very general terms, with unsupervised learning we often have fewer and less-objective measures at our disposal to evaluate how closely the relationships we have modeled aligns with the true structure lurking under our data, $P(Z)$.[1]

2. *All code written in Python.*

(a+b) We begin by importing some useful packages and classes (from math, numpy, pandas, matplotlib, scikitlearn, etc.)

```
import math
from scipy.stats import t
import numpy as np
import pandas as pd
import matplotlib
import matplotlib.pyplot as plt
from sklearn import linear_model
from sklearn.preprocessing import PolynomialFeatures
from sklearn import metrics
```

Using sckikitlearn's LinearRegression class in the linear_model package, we will predict mpg as a function of the number of cylinders (cyl).

[1] Background information and inspiration from ESLII, p.485-487

The code below randomly splits our data into a training set (80% of data) and testing set (20% of data).[2,3] It then trains a regression model on our training data and returns the model's parameters $\beta = (\beta_0, \beta_1)$.

```
#Read in CSV to dataframe, and split dataframe into test data (aprx. 20%)
#and train data (aprx. 80%)
df=pd.read_csv('/Users/emax/Downloads/mtcars.csv')
np.random.seed(3)
msk = np.random.rand(len(df)) < 0.8

train_data=df[msk]
test_data=df[~msk]

#Save data for training in 2D arrays
Y_train=[[val] for val in train_data['mpg'].values]
X_train=[[val] for val in train_data['cyl'].values]

#Save data for testing in 1D arrays
Y_test=test_data['mpg'].values
X_test=test_data['cyl'].values

#Train linear model on training data
reg=linear_model.LinearRegression(fit_intercept=True)
reg.fit(X_train,Y_train)

#Save and print params and save model as function
beta0=reg.intercept_[0]
beta1=reg.coef_[0][0]
print("Y~ "+str(round(beta0,2))+str(round(beta1,2))+"* x")

def y(x):
    return(beta0+beta1*x)
```

From the code we have $\beta = (\beta_0, \beta_1) = (38.39, -2.96)$ so that our population regression function becomes

$$\hat{Y}(x) = 38.39 - 2.96 \cdot x \tag{1}$$

where x is the number of cylinders.

[2]Credit for code to Andy Hayden: https://stackoverflow.com/questions/24147278/how-do-i-create-test-and-train-samples-from-one-dataframe-with-pandas

[3]Note that we have seeded the value 3 for the "random" split.

To answer the rest of the question, we will apply our model $\hat{Y}(x)$ to the training data and testing data, compute relevant statistics for residuals/errors, and plot the results (figs 1 and 2).[4]

```
#Apply model to training data
X_train_plot=[val[0] for val in X_train]
Y_train_pred=[y(x) for x in X_train_plot]
Y_train_true=[val[0] for val in Y_train]

#Plot model and data
plt.plot(X_train_plot, Y_train_pred, color='blue', linewidth=2, markersize=1.5)
plt.scatter(X_train_plot,Y_train_true, s=3,c='green')

#Compute MSE
mse=metrics.mean_squared_error(Y_train_pred,Y_train_true)

patch=matplotlib.patches.Patch(color='Red',
    label='Mean Squared Error: '+str(round(mse,2)))
plt.legend(handles=[patch])

plt.xlabel('cyl')
plt.ylabel('mpg')
plt.show()
```

```
#Apply model to testing data
X_test_plot=X_test
Y_test_pred=[y(x) for x in X_test_plot]
Y_test_true=Y_test

#Plot model and data
plt.plot(X_test_plot, Y_test_pred, color='blue', linewidth=2, markersize=1.5)
plt.scatter(X_test_plot,Y_test_true, s=3,c='green')
mse=metrics.mean_squared_error(Y_test_pred,Y_test_true)

patch=matplotlib.patches.Patch(color='Red',
    label='Mean Squared Error: '+str(round(mse,2)))
plt.legend(handles=[patch])

plt.xlabel('cyl')
```

[4]Code for plot legend taken directly from matplotlib's legend guide, https://matplotlib.org/tutorials/intermediate/legend_guide.html.

```
plt.ylabel('mpg')
plt.show()
```

Our outputs are shown below:

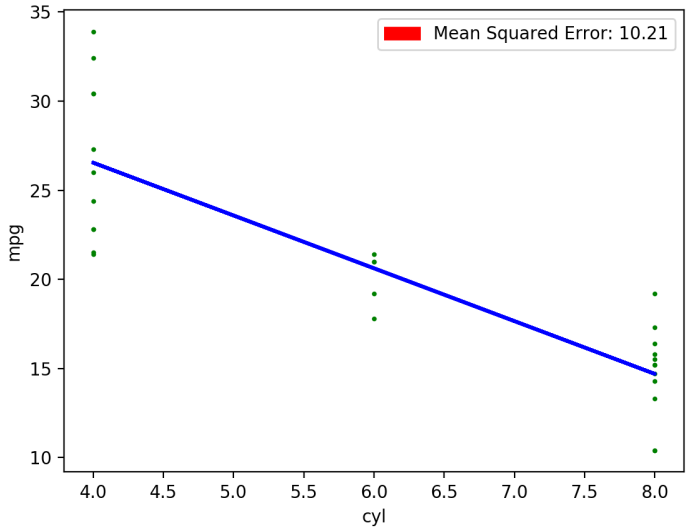

Figure 1: Model applied to training data (Blue) and training data (Green)

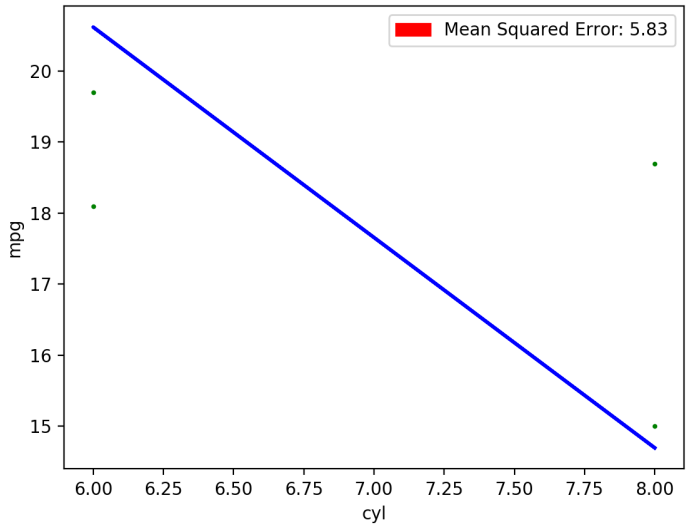

Figure 2: Model appllied to testing data (Blue) and testing data (Green)

(c) We now want to add a feature for weight (wt) to our model. We use the same test/train split as above and add the feature directly to our X_train and X_test arrays. We then train our model on the new data.

```
#Using the same test/train split above, add a second feature
#(wt) to 2D array for training data and 2D array for testing
#data
X_train=train_data[['cyl','wt']].values
X_test=test_data[['cyl','wt']].values

#Train linear model on training data
reg=linear_model.LinearRegression(fit_intercept=True)
reg.fit(X_train,Y_train)
```

As above, we save our parameters, save our model to a python function, and print out our model.

```
#Save and print params and save model as function
beta0=reg.intercept_[0]
beta1=reg.coef_[0][0]
beta2=reg.coef_[0][1]
def y(x1,x2):
    return(beta0+beta1*x1+beta2*x2)
print("Y~ "+str(round(beta0,2))+" "+str(round(beta1,2))+
```

```
"*x1 "+str(round(beta2,2))+"* x2")
```

Using our recorded value for $\beta = (\beta_0, \beta_1, \beta_2) = (39.9, -1.54, 03.17)$ our model becomes

$$\hat{Y}(x) = 39.9 - 1.54 \cdot x_1 - 3.17 \cdot x_2 \tag{2}$$

where x_1 =# cylinders and x_2 =weight.

We can compare this model (2) and the one-feature model (1). The intercept β_0 has increased slightly (from 38.89 to 39.9). More noticeably, the effect of the # of cylinders has halved from the one-feature to the two-feature model. This makes sense if we consider that vehicle weight and # cylinders are themselves likely correlated; it's not so much extra work in Python to plot them against each other and confirm this intuition (fig 3, below).

```
#Intuitive check for correlation of wt and cyl
Y_check=[val[1] for val in X_train]
X_check=[val[0] for val in X_train]

reg.fit([[x] for x in X_check],[[y] for y in Y_check])
beta0=reg.intercept_[0]
beta1=reg.coef_[0][0]

Y_check_pred=[beta0+beta1*x for x in X_check]

plt.scatter(X_check,Y_check, c='red', s=.8)
plt.plot(X_check,Y_check_pred,color='blue',linewidth=1, markersize=1)
plt.xlabel('# of Cylinders')
plt.ylabel('Weight')
plt.show()
```

This correlation would suggest that some of the explanatory effect of weight on mpg was being expressed through the cylinder feature; since the part of the mpg explained by weight can of course be explained better by weight than by its proxy (# cylinders) in the one-feature model, it would make sense that in the two-feature model weight would eat up some of the coefficient/significance of # cylinders.

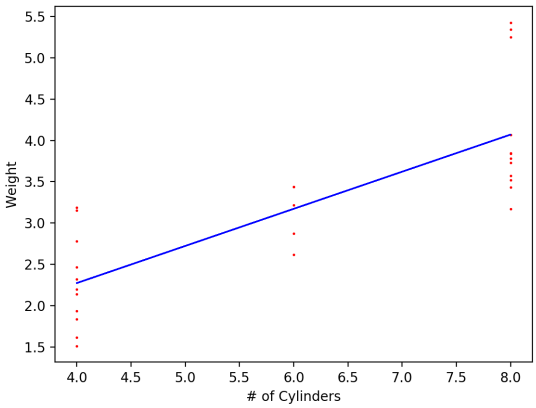

Figure 3: Figure 3: Relationship between # cylinders and weight

We can look at the MSE for our model as well. We will do this in Python below.

```
#Apply model to testing data and training data and compute MSE
Y_train_pred=[y(val[0],val[1]) for val in X_train]
Y_train_true=Y_train
mse=metrics.mean_squared_error(Y_train_pred,Y_train_true)
print('Training MSE: '+str(round(mse,2)))

Y_test_pred=[y(val[0],val[1]) for val in X_test]
Y_test_true=Y_test
mse=metrics.mean_squared_error(Y_test_pred,Y_test_true)
print('Testing MSE: '+str(round(mse,2)))
```

And we have as our results:

Training MSE:	6.37
Testing MSE:	3.26

Both MSEs are better for the two-feature model than the one-feature model. This makes intuitive sense, even at the level of linear algebra: you can draw a flat line (zero-feature model) through one point, a sloped line (one-feature model) through any two points, a plane (two-feature model) through any 3 points, a (k-1)-dimensional hyperplane through any k points. So (and if you close your eyes, you can kind of see this) it seems like a natural extension of this fact that you can get closer to $n >> k$ points with a k- feature model as k grows, and therefore that the model's MSE would decrease as k grows. (Obviously, however, there is a limit to how useful models

become for large k; the fact you can nominally minimize MSE just by including many features is precisely the issue of overfitting; in particular, I would hesitate to use the two-feature model's smaller MSE to claim it is "better" or more accurate than the one-feature model.)

(d) Now, we want to add an interaction term. We can directly add a term cyl·wt to our our 2D array of training data and 1D array of testing data and then train our model again:

```
#Add interaction term to training and testing data
X_train=[[val[0],val[1],val[0]*val[1]] for val in X_train]
X_test=[[val[0],val[1],val[0]*val[1]] for val in X_test]
```

As above, we train our (now three-feature) model on the traning data, save and print our parameters, and save our model in a python function.

```
#Train linear model on training data
reg=linear_model.LinearRegression(fit_intercept=True)
reg.fit(X_train,Y_train)

#Save and print params and save model as function
beta0=reg.intercept_[0]
beta1=reg.coef_[0][0]
beta2=reg.coef_[0][1]
beta3=reg.coef_[0][2]

def y(x1,x2):
    return(beta0+beta1*x1+beta2*x2+beta3*x1*x2)


print("Y~ "+str(round(beta0,2))+" "+str(round(beta1,2))+
    "*x1 "+str(round(beta2,2))+"* x2 "+str(round(beta3,2))+"* x1x2")
```

Using our reported value for $\beta = (\beta_0, \beta_1, \beta_2, \beta_3) = (54.08, -3.79, -8.47, 0.79)$ our model becomes

$$\hat{Y}(x) = 54.08 - 3.79 \cdot x_1 - 8.47 \cdot x_2 + 0.79 \cdot x_1 x_2 \tag{3}$$

where $x_1 =$# cylinders and $x_2 =$# weight.

Comparing this model to the two-feature model (2), we see our intercept has increased

from 39.9 to 54.08 and that the coefficients β_1, β_2 have increased noticeably (roughly doubling and tripling, respectively.) However, we find a small, positive coefficient β_3 for the interaction term. This term tells us that in our model at higher weights the marginal cylinder has a smaller effect on the vehicle's MPG, and similarly that with many cylinders the marginal unit of weight has a smaller effect on the vehicle's MPG.

Mathematically, we can quantify this more exactly, with

$$\frac{\partial Y}{\partial x_1} = -3.79 + .79x_2$$
$$\frac{\partial Y}{\partial x_2} = -9.2 + .79x_1$$

According to this model, when $x_2 = 4.8$, the marginal cylinder no longer detracts from mpg and thereafter that fuel efficiency increases with the # of cylinders; similarly, once there are around 11-12 cylinders, the marginal unit of weight no longer detracts from mpg and thereafter that fuel efficiency increases with the amount of weight (though this is an inappropriate extrapolation, since 11-12 cylinders is outside of the range of the data.)

On the surface, as stated, the naive reading of these equations would be that for heavy cars, adding a marginal cylinder has only a small impact on mileage, and that similarly for cars with many cylinders, a marginal unit of weight has only small impact on mileage. This explanation could be true.

But another, perhaps more believable interpretation would be to consider the correlation between the variables discussed earlier: if weight and # of cylinders are correlated, perhaps the model's x_1x_2 term is hiding what should properly be an x_1^2 or x_2^2 term (since in a very handwavy sense the correlation would suggest $x_1^2 \sim x_1x_2 \sim x_2^2$. Stated plainly, for heavy cars the marginal cylinder makes little difference on mpg because heavy cars already have many cylinders, and thus $\frac{\partial Y}{\partial x_1} \sim -3.79 + 7.9O(x_1)$, and similarly for cars with many cylinders the marginal unit of weight makes little difference on mpg because cars with many cylinders tend to be heavy.

There is too little data to make an assessment about whether this is really the correct interpretation, but intuitively the idea that the correlation between weight and cylinders is driving these dynamics, and that they are explained in part by diminishing marginal impact of weight and cylinders on mpg, respectively, would seem to make

sense.

We also want to get the MSE for the training and testing data; we do that below

```
#Apply model to testing data and training data and compute MSE
Y_train_pred=[y(val[0],val[1]) for val in X_train]
Y_train_true=Y_train
mse=metrics.mean_squared_error(Y_train_pred,Y_train_true)
print('Training MSE: '+str(round(mse,2)))

Y_test_pred=[y(val[0],val[1]) for val in X_test]
Y_test_true=Y_test
mse=metrics.mean_squared_error(Y_test_pred,Y_test_true)
print('Testing MSE: '+str(round(mse,2)))
```

And we have our results:

Training MSE:	5.31
Testing MSE:	2.13

The MSE has improved for both data sets, which is of course to be expected since we have essentially added a new feature x_1x_2 (see my point above.)

3. *All code written in Python*

   (a) We once again start by importing our packages.

```
import math
import pandas as pd
from scipy.stats import t
import matplotlib.pyplot as plt from sklearn import linear_model
from sklearn.preprocessing import PolynomialFeatures
from sklearn import metrics
```

Both intuitively and mathematically, a fitting a second-order polynomial regression on one variable is like fitting a linear regression on two features: when fitting a polynomial regression we are trying to find the values of $\beta_0, \beta_1, \beta_2$ that minimize the square error of our residuals in a model:

$$\hat{Y}(x) = \beta_0 + \beta_1 x_1 + \beta_2 x^2$$

but of course there is nothing stopping us from calling $x_2 = x^2$, and minimizing the square error of our residuals in a linear model with two features:

$$\hat{Y}(x) = \beta_0 + \beta_1 x_1 + \beta_2 x_2$$

This logic underlies our procedure here: we will treat the second-order term as another feature (practically, as another column in our data), and use the same linear regression models as above to fit our polynomial regression.

The code below reads our data into a pandas dataframe and adds a new column to the dataframe corresponding to the second-order age term (i.e., age^2.) It then splits our data into a training set (80% of data) and testing set (20% of data), as above.

```
#Read in CSV to dataframe
df=pd.read_csv('/Users/emax/Downloads/wage_data.csv')
#Add CSV column for age^2
df['age_sqr']=df.apply(lambda row: row['age']**2, axis=1)

#Split dataframe into test data and train data
np.random.seed(3)
msk = np.random.rand(len(df)) < 0.8

train_data=df[msk]
test_data=df[~msk]
```

We again format our data by creating two 2D arrays (for the training data) and two 1D arrays (for the testing data).

```
#Format training and testing data
X_train=train_data[['age','age_sqr']].values
Y_train=[[val] for val in train_data['wage'].values]

X_test=test_data['age'].values
Y_test=test_data['wage'].values
```

Now, we train our linear model, save and print our parameters, and store our model as a python function of 1 variable.

```
#Train linear model on training data
reg=linear_model.LinearRegression(fit_intercept=True)
reg.fit(X_train,Y_train)
```

```
#Save and print params and save model as function
beta0=reg.intercept_[0]
beta1=reg.coef_[0][0]
beta2=reg.coef_[0][1]
print("Y~ "+str(round(beta0,2))+" "+str(round(beta1,2))+"*x "
    +str(round(beta2,2))+"* x^2 ")
def y(x):
    return(beta0+beta1*x+beta2*x**2)
```

We find $\beta = (\beta_0, \beta_1, \beta_2) = (-13.31, 5.39, -0.05)$ so that our model becomes:

$$\hat{Y}(x) = -13.31 + 5.39 \cdot x - 0.05 \cdot x^2 \tag{4}$$

b. Thanks to my poor choice of ML package, I don't have a built-in method for confidence intervals. Maybe, though, there is some gain in understanding from programming this all by hand.

By the same logic as above, we can approach confidence intervals just as when fitting a two-feature linear regression. Let us consider the math for the confidence interval around predictions from a multi-feature linear model.[5]

If we have k features, then we as usual can propend a column of ones to our data to get an $n-$by-$(k+1)$ array X. If X_i is the ith row of our propended matrix X, and $\hat{Y}_i$ is the corresponding prediction, and $MS_{Res} = \frac{1}{\text{deg f}} \sum_{i=1}^{n} r_i^2$ where deg f =degrees of freedom$= n - k - 1$, then the standard error of the prediction $\hat{Y}_i$ is just

$$SE_{\hat{Y}_i} = (MS_{\text{Res}} \cdot X_i (X^T X)^{-1} X_i^T)^{\frac{1}{2}}$$

and therefore the 95% confidence interval $\hat{Y}_i$ is given by

$$\hat{Y}_i \pm t_c \cdot SE_{\hat{Y}_i}$$

where t_c is the critical value or the t distribution with $n - k - 1$ degrees of freedom and significance .025.

This is not so hard to implement in Python. The first step is to use our model to make

[5]Math taken from "Real Statistics Using Excel" by Charles Zaiontz. http://www.real-statistics.com/multiple-regression/confidence-and-prediction-intervals/

predictions based on the training data, and then use these predictions to calculuate MS_{Res} and t_c.

```
#Get predictions from training data
Y_train_pred=[y(val[0]) for val in X_train]

#Get degrees of freedom
deg_f=len(Y_train_pred)-3

#Compute MSres from training data
pred_true_df=pd.DataFrame({'Pred.':Y_train_pred,
    'True':[val[0] for val in Y_train]})
pred_true_df['Resid_sqr']=pred_true_df.apply(lambda row:
    (row['Pred.']-row['True'])**2, axis=1)

RSS=sum(pred_true_df['Resid_sqr'])
MSres=RSS/deg_f

#Get tc critical value from t distribution
t_c=t.ppf(1-.025, df=deg_f)
```

Next, we format our training data as X (above), and compute the term $(X^TX)^{-1}$:

```
#Save training data for plotting
Y_train_plot=[val[0] for val in Y_train]
X_train_plot=[val[0] for val in X_train]

#Propend column of 1s to data array and interpret as matrix
X_train=np.asmatrix([[1,val[0],val[1]] for val in X_train])
X_train_T=X_train.transpose()
C=np.dot(X_train_T,X_train).I
```

Now, using our formula above we can define the upper and lower confidence bounds around our model, at least for each observation X_i. But, loosely speaking, the estimate can be interpreted for the function as a whole. So we define upper and lower bounds for the confidence interval around $\hat{Y}(x)$:

```
#Upper confidence window for y
def y_up(x):
    A=np.asmatrix([1,x,x**2])
    se=math.sqrt(MSres*np.dot(np.dot(A,C),A.transpose()))
    return(y(x)+t_c*se)
#Lower confidence window for y
```

```
def y_down(x):
    A=np.asmatrix([1,x,x**2])
    se=math.sqrt(MSres*np.dot(np.dot(A,C),A.transpose()))
    return(y(x)-t_c*se)
```

All that's left is to plot our function and the associated confidence intervals. We will include the real and training data in our plot (fig 4, below).

```
#Define range for function
X_fine=[i for i in range(min(min(X_test),min(X_train_plot)),
    max(max(X_test),max(X_train_plot)))]

Y_fine=[y(x) for x in X_fine]
Y_lower=[y_down(x) for x in X_fine]
Y_upper=[y_up(x) for x in X_fine]

#Plot it with confidence bounds
plt.plot(X_fine,Y_fine,color='blue',linewidth=1, markersize=.25)
plt.plot(X_fine,Y_lower,color='green',linewidth=1, markersize=.25)
plt.plot(X_fine,Y_upper,color='orange',linewidth=1, markersize=.25)
plt.fill_between(X_fine,Y_lower,Y_upper,facecolor='grey', alpha=0.3)

#Plot rest of data
plt.scatter(X_test,Y_test, c='red', s=.15)
plt.scatter(X_train_plot,Y_train_plot, c='purple', s=.15)
plt.xlabel('Age (yrs)')
plt.ylabel('Wage (1000s USD)')

plt.show()
```

c. The first substantive observation is that the data are likely too scattered for the model to be useful. This makes sense if we consider reality: age would intuitive have far less of an explanatory effect for income than education, location, etc. Nonetheless, there are some interesting insights we can get from a polynomial model when modeling income with age.

The most obvious of these is that a quadratic function with a negative β_2 term will take a local maximum; for the function given in (4), this maximum is at $(53, 151.53)$.

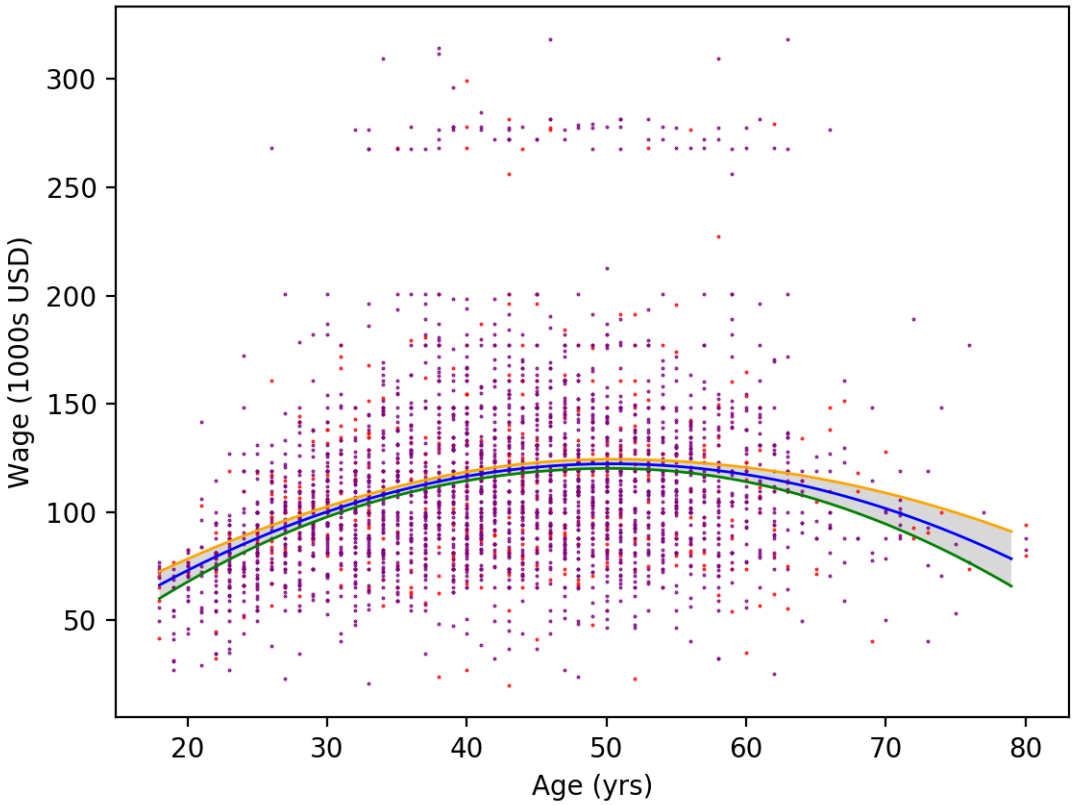

Figure 4: Model appllied to testing data (Blue) and testing data (Green)

It would certainly seem intuitive that income peaks somewhere in middle age before retirement.[6]

d. As stated in (a), statistically we can view kth-order polynomial regressions like k feature linear regressions: we are still trying to minimize least squares by choosing parameters $\beta_0, \ldots, \beta_k$, and the relationship between the features $x_1 = x, x_2 = x^2, \ldots, x_k = x^k$ has no effect on the results minimization problem.

Just because the results of the minimization problem are the same, however, does not mean of course that we can interpret the model as we might any other linear regression: when we know that the feature variables are not independent (and in fact

[6]In fact, at risk of overeading the data, according to a CNBC article college-educated men's wages peak at 53. An interesting coincidence, certainly to be taken with a grain of salt. https://www.cnbc.com/2018/11/02/the-age-at-which-youll-earn-the-most-money-in-your-career.html

are quite far from independent). Implicitly, as k grows there is a risk of overfitting when fitting a kth order polynomial regression: in the most extreme case, for any k observations $(x, y_1), \ldots, x, y_k)$, we could just use a kth order polynomial model, $\hat{Y}(x) = (x - y_1)(x - y_2) \cdots (x - y_k)$, which will pass through every observation!

This is consistent with the general bias-variance trade-off: the further away we move from conventional low- (and independent-) feature, linear regression, the lower the bias and the higher the variance. (A canonical and even more extreme example of this phenomenon than the above would be 1NN fitting.) Thus the more high-order terms in a polynomial regression, the bigger the risk of producing a model that so closely fits our observations as to offer minimal explanatory benefits over the raw observations themselves.

That being said, fitting a second-order polynomial regression does seem to be a reasonably appropriate method for the given dataset, since a) with so many terms which are so disperse, two features, even when highly correlated, are unlikely to lead to overfitting and b) intuitively, as discussed above, we would expect one's lifetime earnings to have a local maximum and a shape similar to a concave second-order polynomial.