project_2_final

May 19, 2022

0.1 Econ 412 Project 2

Alex Hong: 905857714

Gedian Wang: 705638831

Youssef Mahmoud: 905854027

Zachary DeBar: 705867064

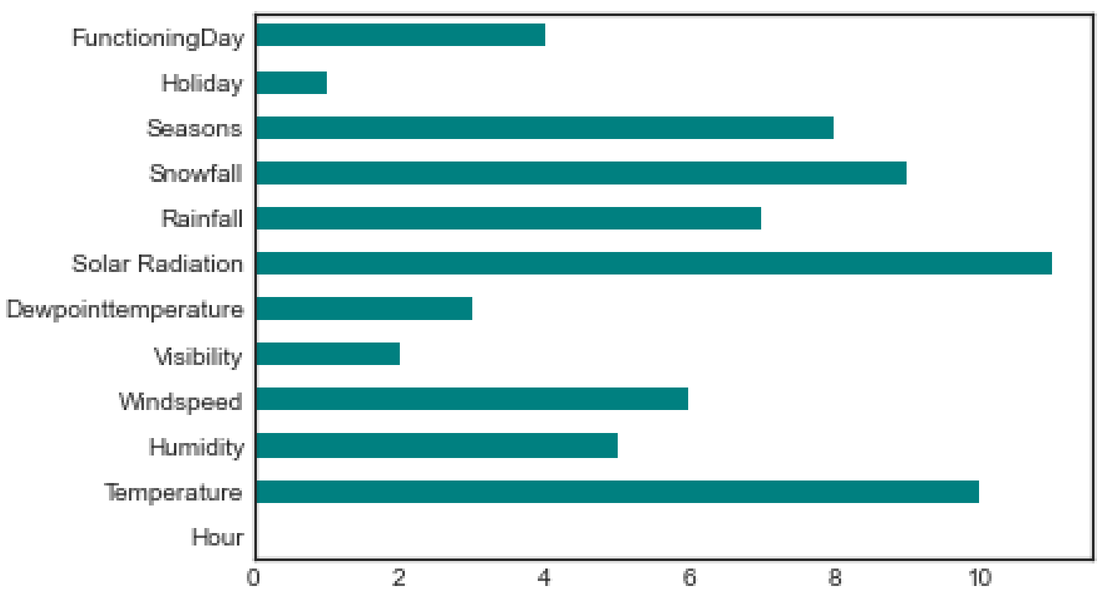
[1]:
```
# Import Libraries

import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import graphviz
import seaborn as sns
from sklearn import tree

from sklearn.preprocessing import PolynomialFeatures
import statsmodels.api as sm
import statsmodels.formula.api as smf
from patsy import dmatrix

from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier,␣
 ↪export_graphviz
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor
from sklearn.metrics import confusion_matrix, mean_squared_error

#! pip install skfeature-chappers

from skfeature.function.similarity_based import fisher_score
%matplotlib inline
plt.style.use('seaborn-white')
```

[2]:
```
#google colab file upload
#from google.colab import files
```

```
#uploaded = files.upload()
```

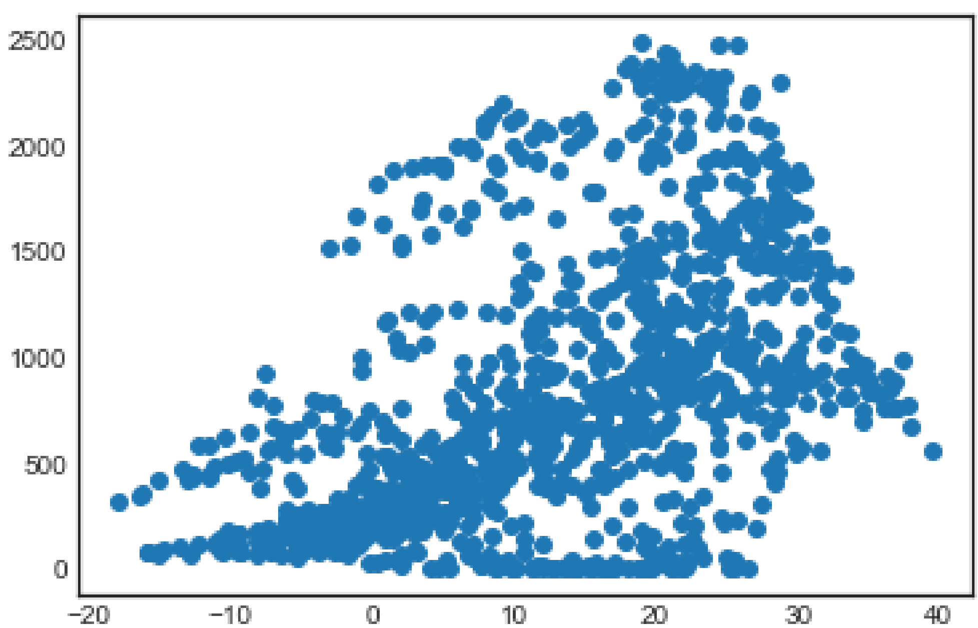
0.2 Part I

[3]:
```
# import dataset
data = pd.read_csv("SeoulBikeData.csv")
data.columns = ['Date', 'RentedBikeCount', 'Hour', 'Temperature', 'Humidity',␣
↪'Windspeed', 'Visibility', 'Dewpointtemperature',
       'Solar Radiation', 'Rainfall', 'Snowfall', 'Seasons', 'Holiday',␣
↪'FunctioningDay']
data['Seasons'] = data['Seasons'].map({'Winter': 1.0, 'Autumn': 2.0, 'Summer':␣
↪3.0,'Spring': 4.0,})
data['Holiday'] = data['Holiday'].map({'No Holiday': 1.0,  'Holiday': 2.0})
data['FunctioningDay'] = data['FunctioningDay'].map({'Yes': 1.0, 'NO': 2.0})

# sampling - 1 in 8
data = data[::8]

# fisher score - calculation
X = data.iloc[:, 2:].values
y = data.RentedBikeCount.values
ranks = fisher_score.fisher_score(X, y)

# fisher score - result
feat_importances = pd.Series(ranks, data.columns[2:len(data.columns)])
feat_importances.plot(kind = 'barh', color = 'teal')
plt.show()
```



Based on the fisher score, the top 5 predictors are Solar Radiation, Temperature, Snowfall, Seasons, and Rainfall. Since temperature is related to all of them, we decide to use temperature as the main predictor to conduct piecewise polynomial. This also allows us to avoid any troubles caused by intercorrelation between these predictors.

[4]:
```
# Piecewise Polynomial - Using Temperature as the predictor

# scatterplot - Rent~Temperature
plt.scatter(data.Temperature, data.RentedBikeCount)
plt.show()

# Split 'Temperature' into 4 equal-distance pieces
data_cut, bins = pd.cut(data.Temperature, 4, retbins = True, right = True)
print(data_cut.value_counts(sort = False))

# Combine variables into one dataframe
data_steps = pd.concat([data.Temperature, data_cut, data.RentedBikeCount], keys␣
↪= ['Temperature', 'Temp_cuts', 'Rent_Counts'], axis = 1)
print(data_steps.head(5))

# Create dummy variables for the age groups
data_steps_dummies = pd.get_dummies(data_steps['Temp_cuts'])

# Statsmodels requires explicit adding of a constant (intercept)
data_steps_dummies = sm.add_constant(data_steps_dummies)
print(data_steps_dummies.head(5))
```



```
(-17.857, -3.5]    120
(-3.5, 10.8]       357
(10.8, 25.1]       430
(25.1, 39.4]       188
Name: Temperature, dtype: int64
    Temperature        Temp_cuts  Rent_Counts
0          -5.2  (-17.857, -3.5]          254
8          -7.6  (-17.857, -3.5]          930
16          1.2     (-3.5, 10.8]          484
24         -1.8     (-3.5, 10.8]          328
32         -4.2  (-17.857, -3.5]          219
    const  (-17.857, -3.5]  (-3.5, 10.8]  (10.8, 25.1]  (25.1, 39.4]
0     1.0                1             0             0             0
8     1.0                1             0             0             0
16    1.0                0             1             0             0
24    1.0                0             1             0             0
32    1.0                1             0             0             0

C:\Users\Zachary DeBar\anaconda3\lib\site-
packages\statsmodels\tsa\tsatools.py:142: FutureWarning: In a future version of
pandas all arguments of concat except for the argument 'objs' will be keyword-
only
  x = pd.concat(x[::order], 1)
```

[5]:
```
data_steps_dummies.columns[1]
```

[5]:
```
Interval(-17.857, -3.5, closed='right')
```

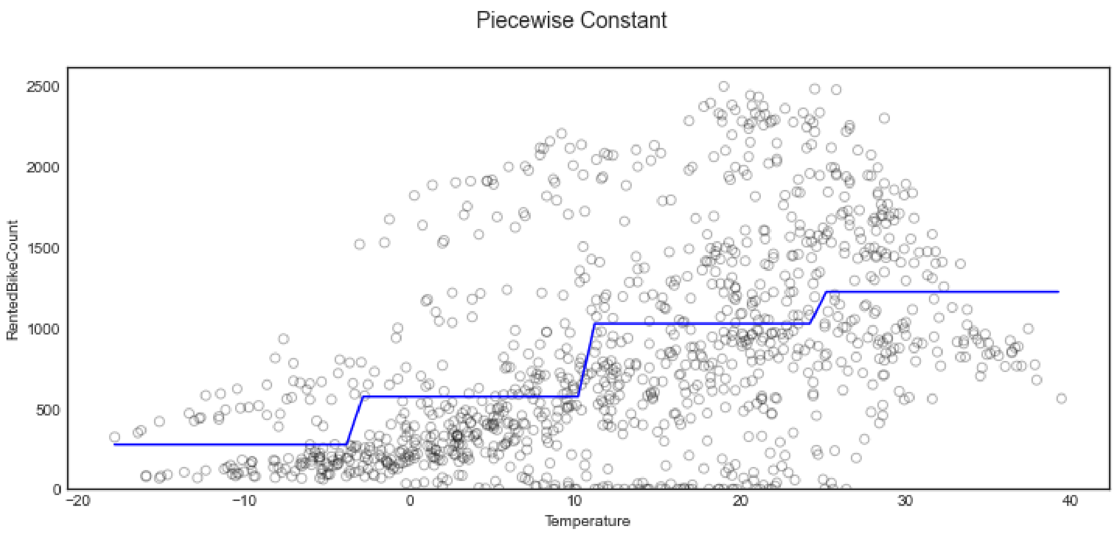
[6]:
```
# Piecewise Polynomial - Cont. (OK)

# Fit a Linear Regression using a Step Function

step_fit = sm.GLM(data_steps.Rent_Counts, data_steps_dummies.
 ↪drop(data_steps_dummies.columns[1], axis=1)).fit()
print(step_fit.summary().tables[1])

## Put the test data in the same bins as the training data.
Temp_grid = np.arange(data.Temperature.min(), data.Temperature.max()).
 ↪reshape(-1, 1)
bin_mapping = np.digitize(Temp_grid.ravel(), bins)
print(bin_mapping)

## Get dummies, drop first dummy category, add constant
X_test2 = sm.add_constant(pd.get_dummies(bin_mapping).drop(1, axis=1))

## Compute the fitted values - linear
pred_step_linear = step_fit.predict(X_test2)
```

```python
## creating plots
fig, (ax1) = plt.subplots(figsize=(12,5))
fig.suptitle('Piecewise Constant', fontsize=14)

## Scatter plot with polynomial regression line
ax1.scatter(data.Temperature, data.RentedBikeCount, facecolor='None',␣
 ↪edgecolor='k', alpha=0.3)
ax1.plot(Temp_grid, pred_step_linear, c='b')
ax1.set_xlabel('Temperature')
ax1.set_ylabel('RentedBikeCount')
ax1.set_ylim(ymin=0)



print(step_fit.summary().tables[1])
```

```
================================================================================
                   coef    std err          z      P>|z|      [0.025      0.975]
--------------------------------------------------------------------------------
const          276.0083     50.965      5.416      0.000     176.119     375.898
(-3.5, 10.8]   298.3026     58.911      5.064      0.000     182.839     413.766
(10.8, 25.1]   747.7498     57.639     12.973      0.000     634.779     860.721
(25.1, 39.4]   945.6672     65.233     14.497      0.000     817.812    1073.522
================================================================================
[1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3
 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4]
```


```
================================================================================
                   coef    std err          z      P>|z|      [0.025      0.975]
--------------------------------------------------------------------------------
const          276.0083     50.965      5.416      0.000     176.119     375.898
(-3.5, 10.8]   298.3026     58.911      5.064      0.000     182.839     413.766
(10.8, 25.1]   747.7498     57.639     12.973      0.000     634.779     860.721
(25.1, 39.4]   945.6672     65.233     14.497      0.000     817.812    1073.522
================================================================================
```


```
C:\Users\Zachary DeBar\anaconda3\lib\site-
packages\statsmodels\tsa\tsatools.py:142: FutureWarning: In a future version of
pandas all arguments of concat except for the argument 'objs' will be keyword-
only
  x = pd.concat(x[::order], 1)
```



Our initial piecewise test is shown above, with only constant estimates for each segment. We will next begin testing piecewise polynomial models.

[7]:
```
X_subset1 = data[(data['Temperature']>-17.857) & (data['Temperature']< -3.5)]
X_subset2 = data[(data['Temperature']>-3.5) & (data['Temperature']< 10.8)]
X_subset3 = data[(data['Temperature']>10.8) & (data['Temperature']< 25.1)]
X_subset4 = data[(data['Temperature']>25.1) & (data['Temperature']< 39.4)]
```

[8]:
```
#linear-subset1
X_subset11 = PolynomialFeatures(1).fit_transform(X_subset1.Temperature.values.
 ↪reshape(-1,1))
fit11 = sm.GLS(X_subset1.RentedBikeCount, X_subset11).fit()
```

[9]:
```
#quadratic-subset1
X_subset12 = PolynomialFeatures(2).fit_transform(X_subset1.Temperature.values.
 ↪reshape(-1,1))
fit12 = sm.GLS(X_subset1.RentedBikeCount, X_subset12).fit()
fit12.summary().tables[1]
```

[9]: <class 'statsmodels.iolib.table.SimpleTable'>

[10]:
```
#cubic-subset1
X_subset13 = PolynomialFeatures(3).fit_transform(X_subset1.Temperature.values.
 ↪reshape(-1,1))
fit13 = sm.GLS(X_subset1.RentedBikeCount, X_subset13).fit()
fit13.summary().tables[1]
```

[10]: <class 'statsmodels.iolib.table.SimpleTable'>

[11]:
```
#linear-subset2
X_subset21 = PolynomialFeatures(1).fit_transform(X_subset2.Temperature.values.
↪reshape(-1,1))
fit21 = sm.GLS(X_subset2.RentedBikeCount, X_subset21).fit()
```

[12]:
```
#quadratic-subset2
X_subset22 = PolynomialFeatures(2).fit_transform(X_subset2.Temperature.values.
↪reshape(-1,1))
fit22 = sm.GLS(X_subset2.RentedBikeCount, X_subset22).fit()
fit22.summary().tables[1]
```

[12]: <class 'statsmodels.iolib.table.SimpleTable'>

[13]:
```
#cubic-subset2
X_subset23 = PolynomialFeatures(3).fit_transform(X_subset2.Temperature.values.
↪reshape(-1,1))
fit23 = sm.GLS(X_subset2.RentedBikeCount, X_subset23).fit()
fit23.summary().tables[1]
```

[13]: <class 'statsmodels.iolib.table.SimpleTable'>

[14]:
```
#linear-subset3
X_subset31 = PolynomialFeatures(1).fit_transform(X_subset3.Temperature.values.
↪reshape(-1,1))
fit31 = sm.GLS(X_subset3.RentedBikeCount, X_subset31).fit()
```

[15]:
```
#quadratic-subset3
X_subset32 = PolynomialFeatures(2).fit_transform(X_subset3.Temperature.values.
↪reshape(-1,1))
fit32 = sm.GLS(X_subset3.RentedBikeCount, X_subset32).fit()
fit32.summary().tables[1]
```

[15]: <class 'statsmodels.iolib.table.SimpleTable'>

[16]:
```
#cubic-subset3
X_subset33 = PolynomialFeatures(3).fit_transform(X_subset3.Temperature.values.
↪reshape(-1,1))
fit33 = sm.GLS(X_subset3.RentedBikeCount, X_subset33).fit()
fit33.summary().tables[1]
```

[16]: <class 'statsmodels.iolib.table.SimpleTable'>

[17]:
```
#linear-subset4
X_subset41 = PolynomialFeatures(1).fit_transform(X_subset4.Temperature.values.
↪reshape(-1,1))
fit41 = sm.GLS(X_subset4.RentedBikeCount, X_subset41).fit()
```

[18]:
```
#quadratic-subset4
X_subset42 = PolynomialFeatures(2).fit_transform(X_subset4.Temperature.values.
↪reshape(-1,1))
fit42 = sm.GLS(X_subset4.RentedBikeCount, X_subset42).fit()
fit42.summary().tables[1]
```

[18]: <class 'statsmodels.iolib.table.SimpleTable'>

[19]:
```
#cubic-subset4
X_subset43 = PolynomialFeatures(3).fit_transform(X_subset4.Temperature.values.
↪reshape(-1,1))
fit43 = sm.GLS(X_subset4.RentedBikeCount, X_subset43).fit()
fit43.summary().tables[1]
```

[19]: <class 'statsmodels.iolib.table.SimpleTable'>

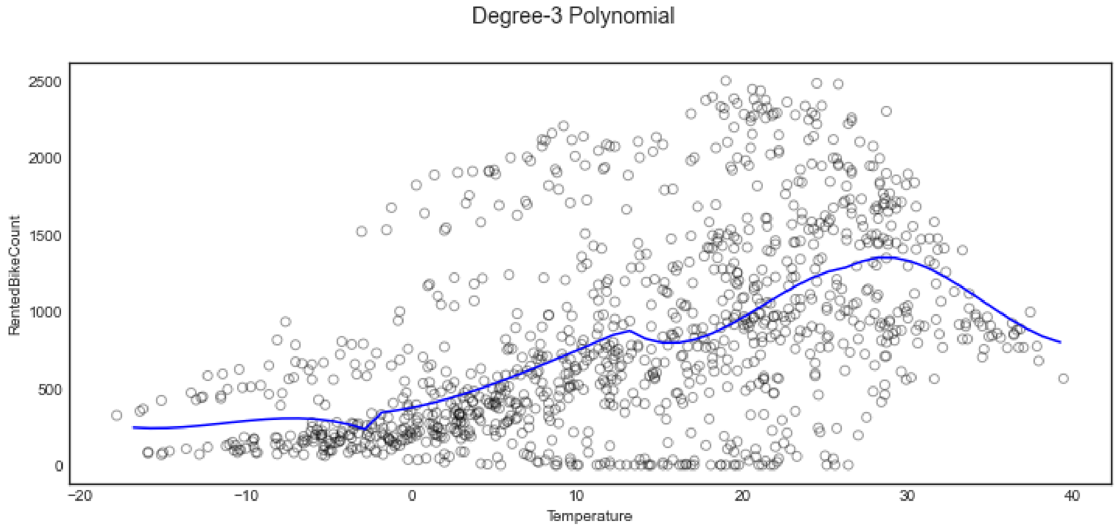
[20]:
```
# 4*anova
print("Subset 1:")
print(sm.stats.anova_lm(fit11, fit12, fit13, typ=1))
print("Subset 2:")
print(sm.stats.anova_lm(fit21, fit22, fit23, typ=1))
print("Subset 3:")
print(sm.stats.anova_lm(fit31, fit32, fit33, typ=1))
print("Subset 4:")
print(sm.stats.anova_lm(fit41, fit42, fit43, typ=1))

#cubic models perform the best

# Create array of test data. Transform to polynomial degree 4 and run␣
↪prediction.
Temp_grid_1 = np.arange(X_subset1.Temperature.min(), X_subset1.Temperature.
↪max()).reshape(-1,1)
Temp_grid_2 = np.arange(X_subset2.Temperature.min(), X_subset2.Temperature.
↪max()).reshape(-1,1)
Temp_grid_3 = np.arange(X_subset3.Temperature.min(), X_subset3.Temperature.
↪max()).reshape(-1,1)
Temp_grid_4 = np.arange(X_subset4.Temperature.min(), X_subset4.Temperature.
↪max()).reshape(-1,1)

X_test_1 = PolynomialFeatures(3).fit_transform(Temp_grid_1)
X_test_2 = PolynomialFeatures(3).fit_transform(Temp_grid_2)
X_test_3 = PolynomialFeatures(3).fit_transform(Temp_grid_3)
X_test_4 = PolynomialFeatures(3).fit_transform(Temp_grid_4)

pred_1 = fit13.predict(X_test_1)
pred_2 = fit23.predict(X_test_2)
pred_3 = fit33.predict(X_test_3)
```

```
pred_4 = fit43.predict(X_test_4)
```

```
Subset 1:
   df_resid           ssr  df_diff        ss_diff         F    Pr(>F)
0     116.0  5.312926e+06      0.0            NaN       NaN       NaN
1     115.0  5.262085e+06      1.0   50841.316789  1.104182  0.295553
2     114.0  5.249055e+06      1.0   13029.465921  0.282976  0.595792
Subset 2:
   df_resid           ssr  df_diff        ss_diff         F    Pr(>F)
0     354.0  9.088829e+07      0.0            NaN       NaN       NaN
1     353.0  9.079297e+07      1.0   95317.553674  0.369562  0.543634
2     352.0  9.078800e+07      1.0    4969.679647  0.019268  0.889680
Subset 3:
   df_resid           ssr  df_diff         ss_diff         F    Pr(>F)
0     424.0  1.770167e+08      0.0             NaN       NaN       NaN
1     423.0  1.766008e+08      1.0   415933.424183  0.997055  0.318595
2     422.0  1.760424e+08      1.0   558425.076552  1.338629  0.247930
Subset 4:
   df_resid           ssr  df_diff         ss_diff         F    Pr(>F)
0     185.0  4.038066e+07      0.0             NaN       NaN       NaN
1     184.0  4.013958e+07      1.0   241082.920897  1.102335  0.295131
2     183.0  4.002246e+07      1.0   117116.926994  0.535509  0.465236
```

[21]:
```
# plot

## combine preds
preds_total = np.append(pred_1, pred_2)
preds_total = np.append(preds_total, pred_3)
preds_total = np.append(preds_total, pred_4)

## creating plots
fig, ax1 = plt.subplots(1,1, figsize=(12,5))
fig.suptitle('Degree-3 Polynomial', fontsize=14)

# Scatter plot with polynomial regression line
ax1.scatter(data.Temperature, data.RentedBikeCount, facecolor='None',␣
 ↪edgecolor='k', alpha=0.4)

# Note: With seaborn, you can just specify order = 4 to fit the quartic␣
 ↪polynomial
# sns.regplot(data.Temperature, data.RentedBikeCount, order = 3, truncate =␣
 ↪True, scatter = False, ax = ax1)
# ax1.scatter(data.Temperature, data.RentedBikeCount, facecolor='None',␣
 ↪edgecolor='k', alpha=0.3)
ax1.plot(Temp_grid[1:, ], preds_total, c='b')
ax1.set_xlabel('Temperature')
ax1.set_ylabel('RentedBikeCount')
```

[21]: Text(0, 0.5, 'RentedBikeCount')



Shown above is the plot of our best fit piecewise polynomial model, based on our ANOVA findings.

[22]:
```
#begin creating error testing dataset
b = Temp_grid[1:, ]
c = [list(x) for x in b]

#create rounded temp values so they can be merged with temp_grid later
mse_test = pd.DataFrame()
mse_test['pred_temp'] = c
mse_test['pred_t_rounded'] = 0.1

for i in range(0,len(mse_test['pred_temp'])):
    mse_test['pred_t_rounded'].loc[i] = (round(mse_test['pred_temp'].loc[i][0],␣
↪1))


mse_test['pred'] = preds_total


mse_test_1 = pd.DataFrame()
mse_test_1['data_temp'] = data.Temperature
mse_test_1['observs'] = data.RentedBikeCount

#merge prediction and observe that match so MSE can be tested
mse_final = pd.merge(mse_test, mse_test_1, left_on='pred_t_rounded',right_on =␣
↪'data_temp')
```

```
#create error array so it's easy to observe squared residuals
error = []
for i in range(0, len(mse_final['pred'])):
    e = mse_final['observs'].loc[i] - mse_final['pred'].loc[i]
    e2 = e*e
    error.append(e2)


#return MSE
np.mean(error)
```

```
C:\Users\Zachary DeBar\anaconda3\lib\site-packages\pandas\core\indexing.py:1732:
SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-
docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  self._setitem_single_block(indexer, value, name)
```

[22]:

```
309395.6542471267
```

We found, through a corrected sampling of our predictions, that our MSE for this model is 309,395.65.

0.3 Splines - Using Temperature as the predictor (OK)

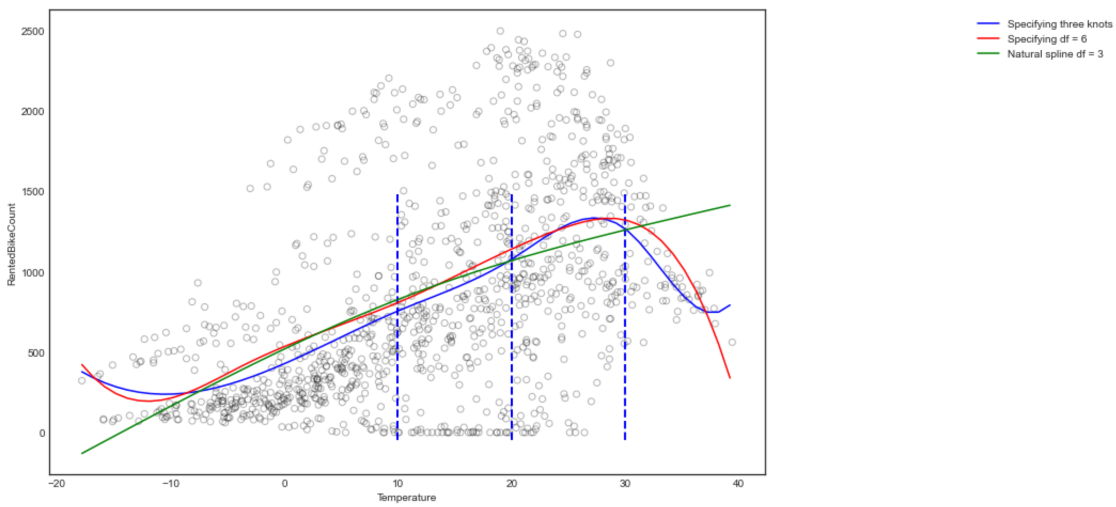
[23]:

```
# Cubic Spline (Basis Functions = BS)
transformed_x = dmatrix("bs(data.Temperature, knots=(10, 20, 30), degree=3,␣
 ↪include_intercept=False)",
                        {"data.Temperature": data.Temperature},␣
 ↪return_type='dataframe')
spline_fit = sm.GLM(data.RentedBikeCount, transformed_x).fit()
spline_pred = spline_fit.predict(dmatrix("bs(Temp_grid, knots = (10, 20, 30),␣
 ↪degree = 3, include_intercept = False)",
                             {"Temp_grid": Temp_grid}, return_type =␣
 ↪'dataframe'))
print(spline_fit.params)
## For a cubic B-spline, we have  d+K = 3 + 3 degrees of freedom +1(intercept)␣
 ↪= 7
## d = boundary knots

# Cubic Spline with Degrees of Freedom Specified
## We could also use the df option to produce a spline with knots at uniform␣
 ↪quantiles of the data.
## Specifying df = 6 degrees of freedom
transformed_x2 = dmatrix("bs(data.Temperature, df = 6, degree = 3,␣
 ↪include_intercept = False)",
```

```
                        {"data.Temperature": data.Temperature}, return_type =␣
↪'dataframe')
spline_fit2 = sm.GLM(data.RentedBikeCount, transformed_x2).fit()
spline_pred2 = spline_fit2.predict(dmatrix("bs(Temp_grid, df = 6, degree = 3,␣
↪include_intercept = False)",
                             {"Temp_grid": Temp_grid}, return_type =␣
↪'dataframe'))
print(spline_fit2.params)
#Nature Spline
transformed_x3 = dmatrix("cr(data.Temperature, df=3)", {"data.Temperature":␣
↪data.Temperature}, return_type='dataframe')
nsspline_fit = sm.GLM(y, transformed_x3).fit()
nsspline_pred = nsspline_fit.predict(dmatrix("cr(Temp_grid, df=3)",␣
↪{"Temp_grid": Temp_grid}, return_type='dataframe'))
print(nsspline_fit.params)

# Plot all the Fits
fig = plt.figure(figsize=(12,8))
plt.scatter(data.Temperature, data.RentedBikeCount,
            facecolor = 'None', edgecolor = 'k', alpha = 0.3)
plt.plot(Temp_grid, spline_pred, color = 'b', label = 'Specifying three knots')
plt.plot(Temp_grid, spline_pred2, color = 'r', label = 'Specifying df = 6')
plt.plot(Temp_grid, nsspline_pred, color = 'g', label = 'Natural spline df =␣
↪3')  # Connected with the unsolved part
[plt.vlines(i , -50, 1500, linestyles='dashed',
            lw = 2, colors = 'b') for i in [10, 20, 30]]
plt.legend(bbox_to_anchor=(1.5, 1.0))
plt.xlabel('Temperature')
plt.ylabel('RentedBikeCount')
plt.show()
```

```
Intercept
377.497538
bs(data.Temperature, knots=(10, 20, 30), degree=3, include_intercept=False)[0]
-378.006791
bs(data.Temperature, knots=(10, 20, 30), degree=3, include_intercept=False)[1]
273.012753
bs(data.Temperature, knots=(10, 20, 30), degree=3, include_intercept=False)[2]
624.334091
bs(data.Temperature, knots=(10, 20, 30), degree=3, include_intercept=False)[3]
1275.108952
bs(data.Temperature, knots=(10, 20, 30), degree=3, include_intercept=False)[4]
206.907531
bs(data.Temperature, knots=(10, 20, 30), degree=3, include_intercept=False)[5]
414.253244
dtype: float64
Intercept                                                                421.966159
```

```
bs(data.Temperature, df=6, degree=3, include_intercept=False)[0]   -412.804073
bs(data.Temperature, df=6, degree=3, include_intercept=False)[1]     70.676116
bs(data.Temperature, df=6, degree=3, include_intercept=False)[2]    356.186296
bs(data.Temperature, df=6, degree=3, include_intercept=False)[3]    943.639852
bs(data.Temperature, df=6, degree=3, include_intercept=False)[4]    994.109847
bs(data.Temperature, df=6, degree=3, include_intercept=False)[5]    -83.334217
dtype: float64
Intercept                          531.915907
cr(data.Temperature, df=3)[0]   -662.045638
cr(data.Temperature, df=3)[1]    313.852131
cr(data.Temperature, df=3)[2]    880.109414
dtype: float64
```



Above is the plot for our initial spline models; shown in red is the best performing model with 6 degrees of freedom specified.

[24]:
```
#MSE Estimate for spline

#begin creating error testing dataset for spline 2
mse_test_2 = pd.DataFrame()
mse_test_2['spline2'] = spline_pred2
mse_test_2['pred_temp_spline'] = Temp_grid

#begin creating error testing dataset for spline 2
mse_test_2 = pd.DataFrame()
mse_test_2['spline2'] = spline_pred2
mse_test_2['pred_temp_spline'] = Temp_grid

#merge prediction and observe that match so MSE can be tested
mse_final2 = pd.merge(mse_test_2, mse_test_1,
```

```
                        left_on='pred_temp_spline',right_on = 'data_temp')
#create error array so it's easy to observe squared residuals
error2 = []
for i in range(0, len(mse_final2['observs'])):
    e = mse_final2['observs'].loc[i] - mse_final2['spline2'].loc[i]
    e2 = e*e
    error2.append(e2)
#return MSE
np.mean(error2)
```

[24]:
```
358896.19034348393
```

We found, through a corrected sampling of our predictions, that our MSE for this model is 358,896.19.

0.4 GAM

[25]:
```
# ! pip install pygam
```

[26]:
```
X_new = data[['Solar Radiation', 'Temperature', 'Snowfall', 'Seasons',
              'Rainfall']]
x_train, x_test, y_train, y_test = train_test_split(X_new, y,
                                                    test_size = 0.5,
                                                    random_state = 5)
from pygam import GAM
gam = GAM().fit(x_train, y_train)
gam.summary()
```

```
GAM
===============================================
==========================================================
Distribution:                      NormalDist Effective DoF:
36.6622
Link Function:                   IdentityLink Log Likelihood:
-7186.8859
Number of Samples:                        547 AIC:
14449.0962
                                              AICc:

14454.825
                                              GCV:

230378.5896
                                              Scale:

202762.4248
                                              Pseudo R-Squared:

0.5337
===============================================================================
==========================
Feature Function                  Lambda               Rank         EDoF
```

```
P > x        Sig. Code
================================= ==================== ============ ============
============ ============
s(0)                              [0.6]                20           12.2
1.84e-10     ***
s(1)                              [0.6]                20           11.6
2.69e-06     ***
s(2)                              [0.6]                20           6.5
9.91e-01
s(3)                              [0.6]                20           3.8
1.42e-05     ***
s(4)                              [0.6]                20           2.6
8.93e-14     ***
intercept                                              1            0.0
3.37e-01
================================================================================
==========================
Significance codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

WARNING: Fitting splines and a linear function to a feature introduces a model
identifiability problem
         which can cause p-values to appear significant when they are not.

WARNING: p-values calculated in this manner behave correctly for un-penalized
models or models with
         known smoothing parameters, but when smoothing parameters have been
estimated, the p-values
         are typically lower than they should be, meaning that the tests reject
the null too readily.
```

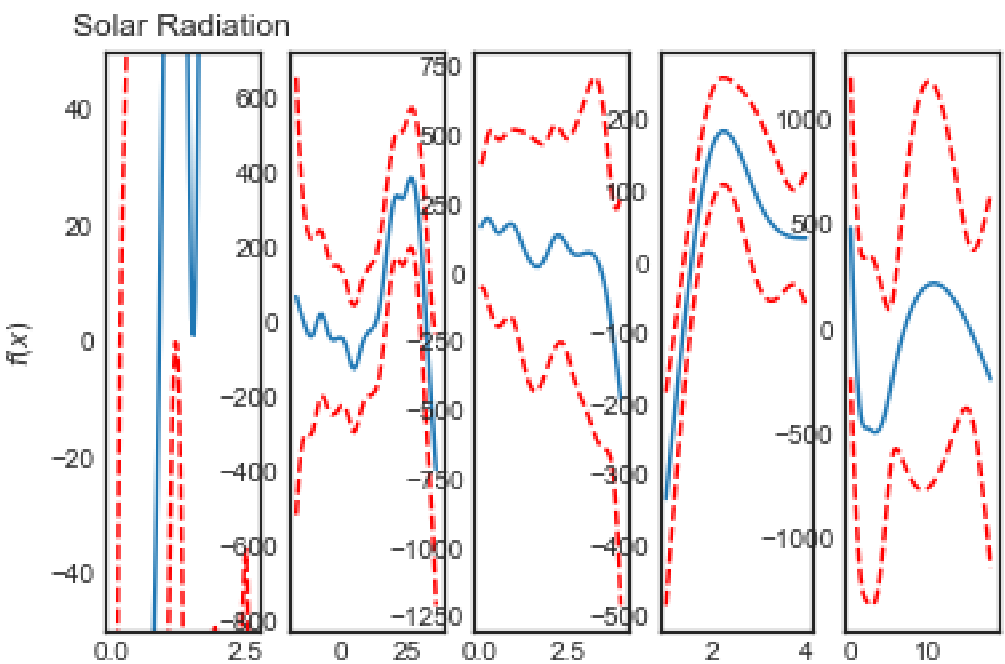
```
C:\Users\ZACHAR~1\AppData\Local\Temp/ipykernel_15056/1365804428.py:8:
UserWarning: KNOWN BUG: p-values computed in this summary are likely much
smaller than they should be.

Please do not make inferences based on these values!

Collaborate on a solution, and stay up to date at:
github.com/dswah/pyGAM/issues/163

  gam.summary()
```

[27]:
```
pred = gam.predict(x_test)
print(mean_squared_error(y_test, pred))
```

```
232378.92039586077
```

We found our GAM to have an MSE of 232,378.92, which was the best performing MSE of the three approaches.

[28]:
```python
## plotting
fig, axs = plt.subplots(1,5);
titles = ['Solar Radiation', 'Temperature', 'Snowfall', 'Seasons', 'Rainfall']
for i, ax in enumerate(axs):
  XX = gam.generate_X_grid(term=i)
  ax.plot(XX[:, i], gam.partial_dependence(term=i, X=XX))
  ax.plot(XX[:, i], gam.partial_dependence(term=i, X=XX, width=.95)[1],
          c='r', ls='--')
  if i == 0:
    ax.set_ylabel('$f(x)$')
    ax.set_ylim(-50,50)
    ax.set_title(titles[i]);
```



0.5 Part 2

[29]:
```python
import pandas as pd
import numpy as np
import matplotlib as mpl
import matplotlib.pyplot as plt
import graphviz
import statsmodels.formula.api as smf
import seaborn as sns
from sklearn import tree
from sklearn.model_selection import train_test_split
```

```python
from sklearn.tree import DecisionTreeRegressor, DecisionTreeClassifier,␣
↪export_graphviz
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor

from sklearn.metrics import confusion_matrix, mean_squared_error
%matplotlib inline
```

[30]:
```python
df = pd.read_csv('Determinants of Wages Data (CPS 1985).csv')
```

[31]:
```python
df['ethnicity'] = df['ethnicity'].map({'cauc':0, 'other':1,'hispanic':2})

df['region'] = df['region'].map({'other':0, 'south':1})

df['gender'] = df['gender'].map({'male':0, 'female':1})

df['sector'] = df['sector'].map({'construction':0, 'manufacturing':1,'other':2})

df['union'] = df['union'].map({'yes':0, 'no':1})

df['married'] = df['married'].map({'yes':0, 'no':1})

df['occupation'] = df['occupation'].map({'services':0, 'sales':1,'worker':
↪2,'office':3,'technical':4,'management':5})
```

[32]:
```python
df
```

[32]:
```
      wage  education  experience  age  ethnicity  region  gender  occupation  \
0     5.10          8          21   35          2       0       1           2
1     4.95          9          42   57          0       0       1           2
2     6.67         12           1   19          0       0       0           2
3     4.00         12           4   22          0       0       0           2
4     7.50         12          17   35          0       0       0           2
..     …          …          …  …        …      …      …           …
529  11.36         18           5   29          0       0       0           4
530   6.10         12          33   51          1       0       1           4
531  23.25         17          25   48          1       0       1           4
532  19.88         12          13   31          0       1       0           4
533  15.38         16          33   55          0       0       0           4

     sector  union  married
0         1      1        0
1         1      1        0
2         1      1        1
3         2      1        1
4         2      1        0
..      …    …      …
529       2      1        1
```

```
530       2     1        0
531       2     0        0
532       2     0        0
533       1     1        0

[534 rows x 11 columns]
```

[33]:
```
df.describe()
```

[33]:
```
             wage   education  experience         age   ethnicity      region  \
count  534.000000  534.000000  534.000000  534.000000  534.000000  534.000000
mean     9.024064   13.018727   17.822097   36.833333    0.226592    0.292135
std      5.139097    2.615373   12.379710   11.726573    0.526203    0.455170
min      1.000000    2.000000    0.000000   18.000000    0.000000    0.000000
25%      5.250000   12.000000    8.000000   28.000000    0.000000    0.000000
50%      7.780000   12.000000   15.000000   35.000000    0.000000    0.000000
75%     11.250000   15.000000   26.000000   44.000000    0.000000    1.000000
max     44.500000   18.000000   55.000000   64.000000    2.000000    1.000000

           gender  occupation      sector       union     married
count  534.000000  534.000000  534.000000  534.000000  534.000000
mean     0.458801    2.501873    1.724719    0.820225    0.344569
std      0.498767    1.529876    0.538453    0.384360    0.475673
min      0.000000    0.000000    0.000000    0.000000    0.000000
25%      0.000000    2.000000    2.000000    1.000000    0.000000
50%      0.000000    2.000000    2.000000    1.000000    0.000000
75%      1.000000    4.000000    2.000000    1.000000    1.000000
max      1.000000    5.000000    2.000000    1.000000    1.000000
```

[34]:
```
df['wage01'] = df.wage.map(lambda x: 1 if x>7.78 else 0)
```

[35]:
```
mod1 = smf.ols(formula =
               'wage01 ~ education + experience + experience + age + ethnicity␣
 ↪+ region + gender + occupation + sector + union +  married',
               data = df)
res1 = mod1.fit()
print(res1.summary())
```

```
                            OLS Regression Results
==============================================================================
Dep. Variable:                 wage01   R-squared:                       0.263
Model:                            OLS   Adj. R-squared:                  0.249
Method:                 Least Squares   F-statistic:                     18.67
Date:                Thu, 19 May 2022   Prob (F-statistic):           2.14e-29
Time:                        10:50:55   Log-Likelihood:                -306.05
No. Observations:                 534   AIC:                             634.1
Df Residuals:                     523   BIC:                             681.2
Df Model:                          10
```

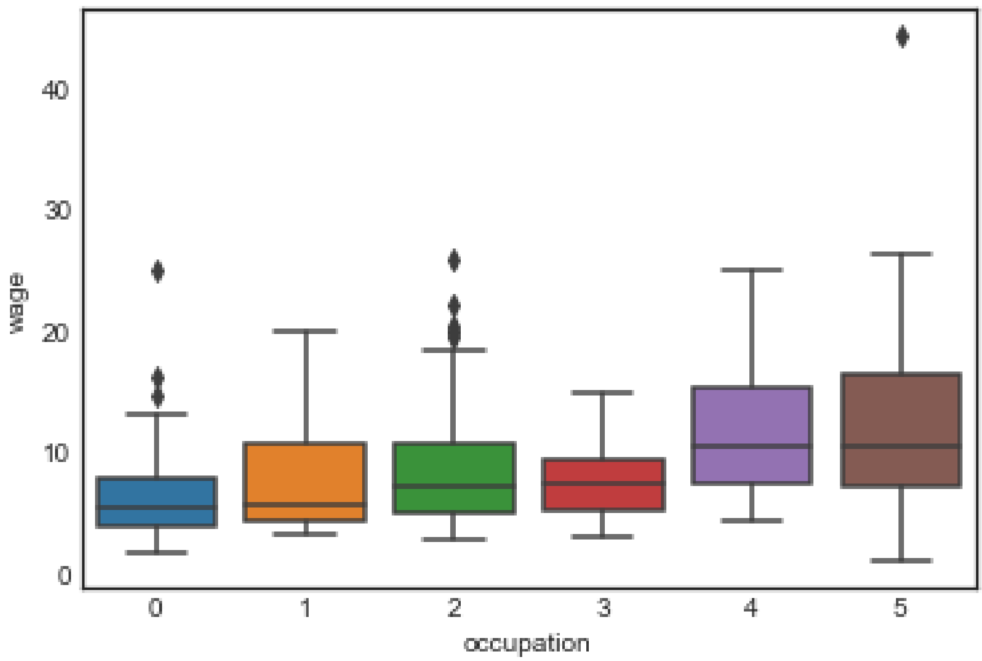
```
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      0.7821      0.671      1.166      0.244      -0.535       2.099
education      0.1920      0.109      1.754      0.080      -0.023       0.407
experience     0.1455      0.109      1.332      0.183      -0.069       0.360
age           -0.1374      0.109     -1.259      0.209      -0.352       0.077
ethnicity     -0.0480      0.037     -1.313      0.190      -0.120       0.024
region        -0.1065      0.042     -2.523      0.012      -0.189      -0.024
gender        -0.1476      0.039     -3.791      0.000      -0.224      -0.071
occupation     0.0553      0.014      3.902      0.000       0.027       0.083
sector        -0.0759      0.036     -2.088      0.037      -0.147      -0.005
union         -0.2322      0.051     -4.586      0.000      -0.332      -0.133
married       -0.0709      0.041     -1.712      0.087      -0.152       0.010
==============================================================================
Omnibus:                      133.966   Durbin-Watson:                   1.994
Prob(Omnibus):                  0.000   Jarque-Bera (JB):               24.207
Skew:                           0.009   Prob(JB):                     5.54e-06
Kurtosis:                       1.957   Cond. No.                     1.69e+03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
[2] The condition number is large, 1.69e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
```

[36]:
```python
df.corr()
```

[36]:
```
                wage  education  experience       age  ethnicity    region  \
wage        1.000000   0.381922    0.087060  0.176967  -0.110107 -0.141031
education   0.381922   1.000000   -0.352676 -0.150019  -0.144870 -0.140143
experience  0.087060  -0.352676    1.000000  0.977961   0.005912 -0.007407
age         0.176967  -0.150019    0.977961  1.000000  -0.025794 -0.038665
ethnicity  -0.110107  -0.144870    0.005912 -0.025794   1.000000  0.122604
region     -0.141031  -0.140143   -0.007407 -0.038665   0.122604  1.000000
gender     -0.205371   0.002031    0.075230  0.079179  -0.010830 -0.021264
occupation  0.365805   0.482023   -0.088791  0.013142  -0.066949 -0.076227
sector     -0.045361   0.188853   -0.111500 -0.075918   0.035155 -0.000430
union      -0.161766   0.023886   -0.117926 -0.119466  -0.057952  0.086275
married    -0.100579   0.035522   -0.270900 -0.278947   0.047276 -0.006523
wage01      0.730236   0.296739    0.120621  0.192879  -0.108813 -0.170553

              gender  occupation    sector     union   married    wage01
wage       -0.205371    0.365805 -0.045361 -0.161766 -0.100579  0.730236
education   0.002031    0.482023  0.188853  0.023886  0.035522  0.296739
```

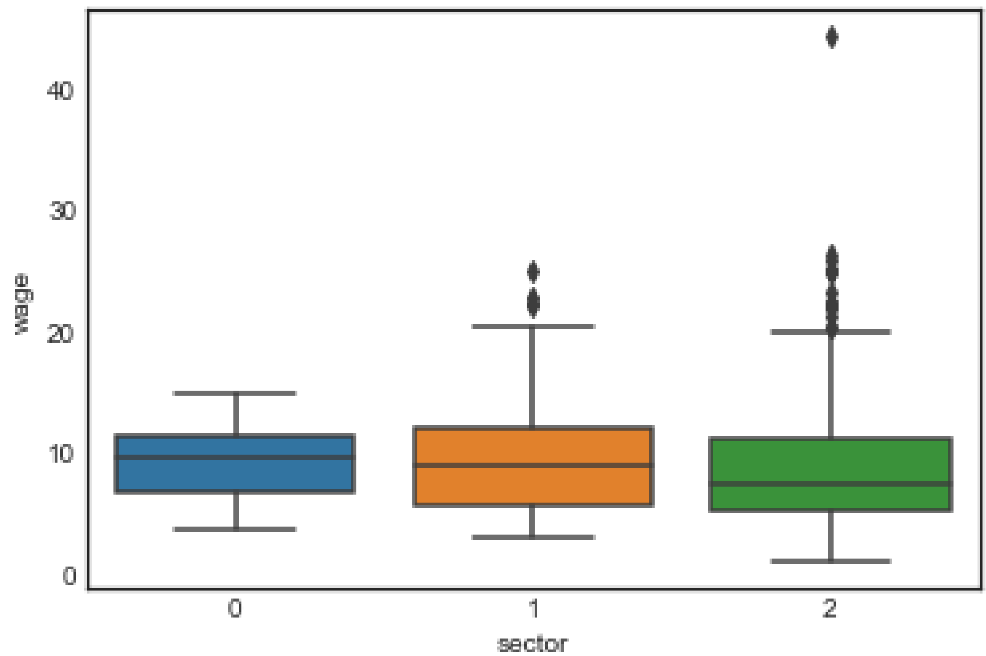
```
experience  0.075230   -0.088791 -0.111500 -0.117926 -0.270900  0.120621
age         0.079179    0.013142 -0.075918 -0.119466 -0.278947  0.192879
ethnicity  -0.010830   -0.066949  0.035155 -0.057952  0.047276 -0.108813
region     -0.021264   -0.076227 -0.000430  0.086275 -0.006523 -0.170553
gender      1.000000    0.012395  0.170764  0.157027 -0.011225 -0.165667
occupation  0.012395    1.000000  0.047317  0.064386 -0.052451  0.287930
sector      0.170764    0.047317  1.000000  0.095849  0.056050 -0.088945
union       0.157027    0.064386  0.095849  1.000000  0.093164 -0.226084
married    -0.011225   -0.052451  0.056050  0.093164  1.000000 -0.139145
wage01     -0.165667    0.287930 -0.088945 -0.226084 -0.139145  1.000000
```

[37]:
```
sns.boxplot(x='occupation', y='wage', data=df);
```



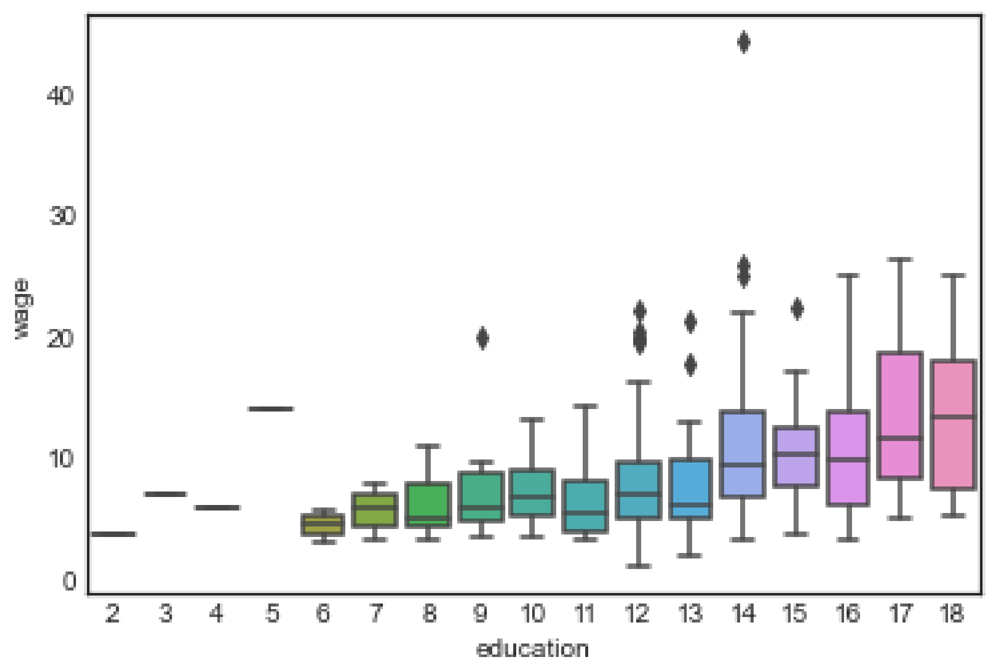
Based on the boxplot, it seems that technical and management occupations earned more than other occupations back in 1985.

[38]:
```
sns.boxplot(x='sector', y='wage', data=df);
```



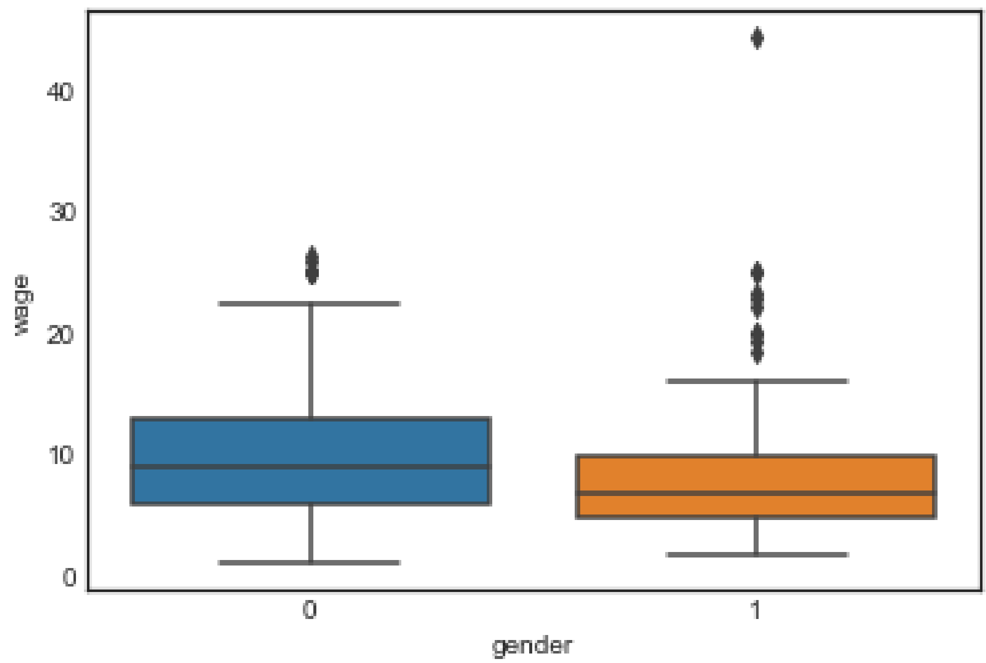
According to the boxplot, there is not much difference in wages across sectors,observations in all sectors seem to have earned around 10$/hr, however, the “other” sector have more outliers indicating that some of our observaions in other sectors were able to earn higher wages back in 1985.

[39]:
```
sns.boxplot(x='education', y='wage', data=df);
```



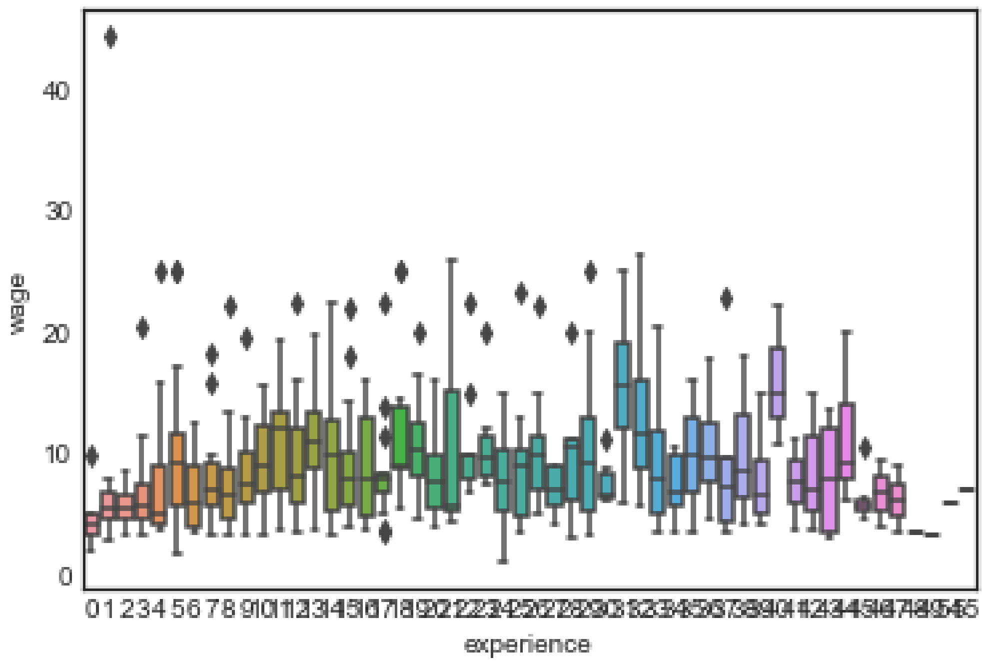
The above boxplot clearly indicates a positive correlation between wage and the level of education.

[40]:
```
sns.boxplot(x='gender', y='wage', data=df);
```



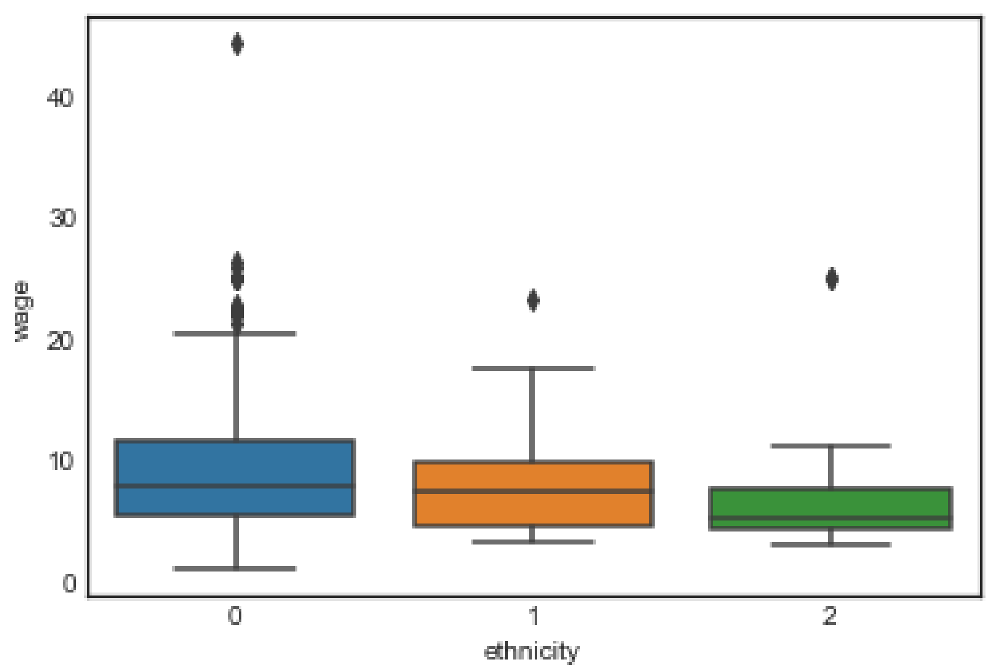
The boxplot clearly indicates that males earned more than females back in 1985, probably due to gender-based wage discrimination.

[41]:
```
sns.boxplot(x='experience', y='wage', data=df);
```



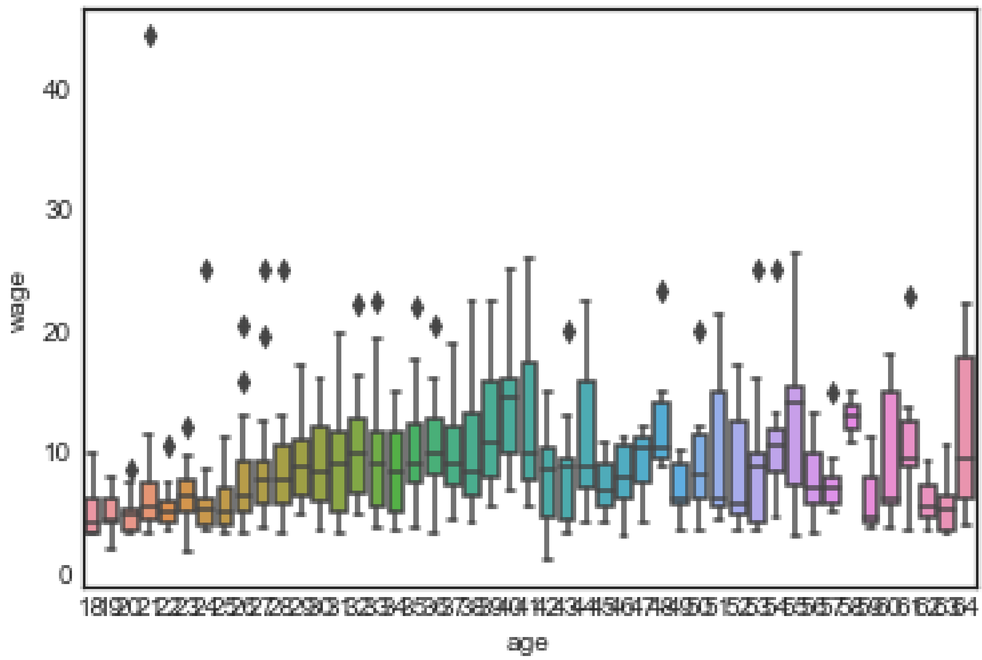
The above box indicates that experience may have a loosely negative exponential curve, increasing as experience increases, but decreasing as some workers retire and others' skills become outpaced by technology.

[42]:
```
sns.boxplot(x='ethnicity', y='wage', data=df);
```



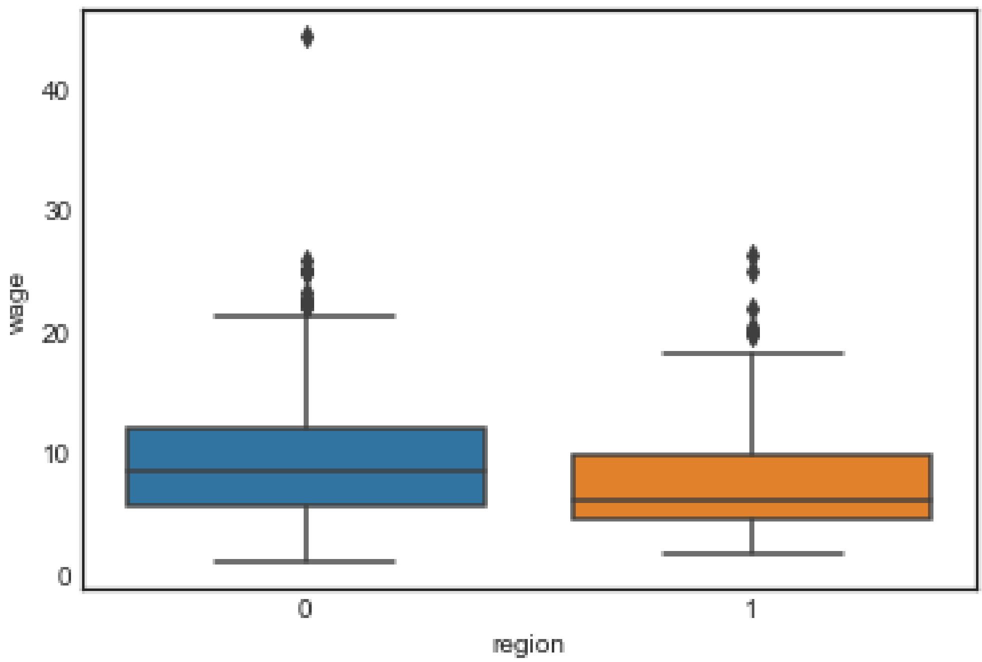
The boxplots indicate that caucasians earned more back in 1985, this porbably due to race-based wage discrmination.

[43]:
```
sns.boxplot(x='age', y='wage', data=df);
```



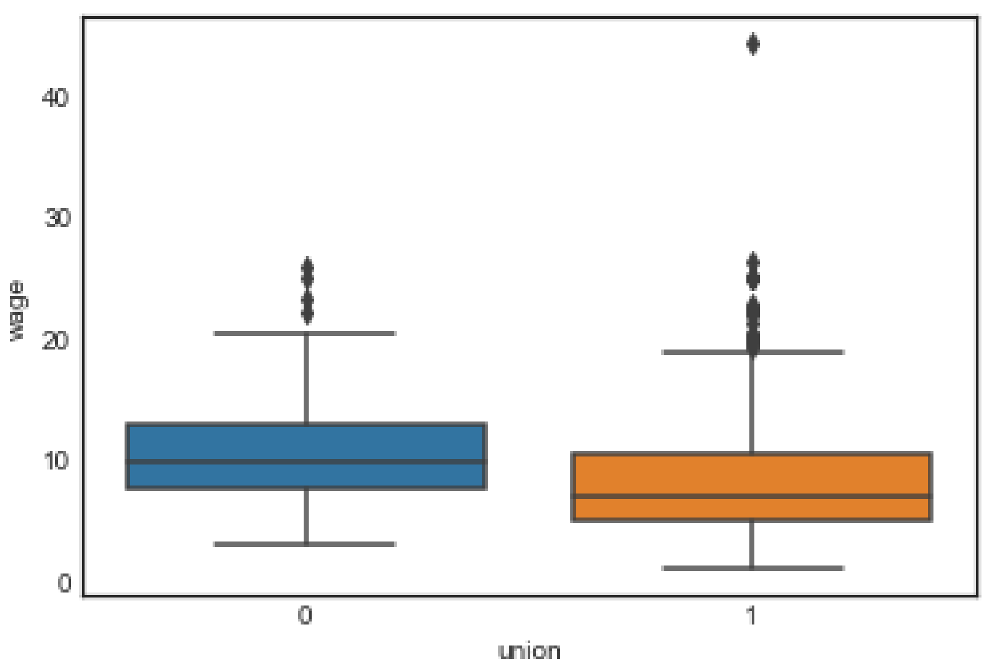
The relationships of age and wage and experience and wage is similar: at younger ages, and lower levels of experience, we observe a positive correlation with wages. However, at a certain age the correlation changes, which could also be due to employees not being up to date with the new technologies in their field, thus their wages would plateau.

[44]:
```
sns.boxplot(x='region', y='wage', data=df);
```



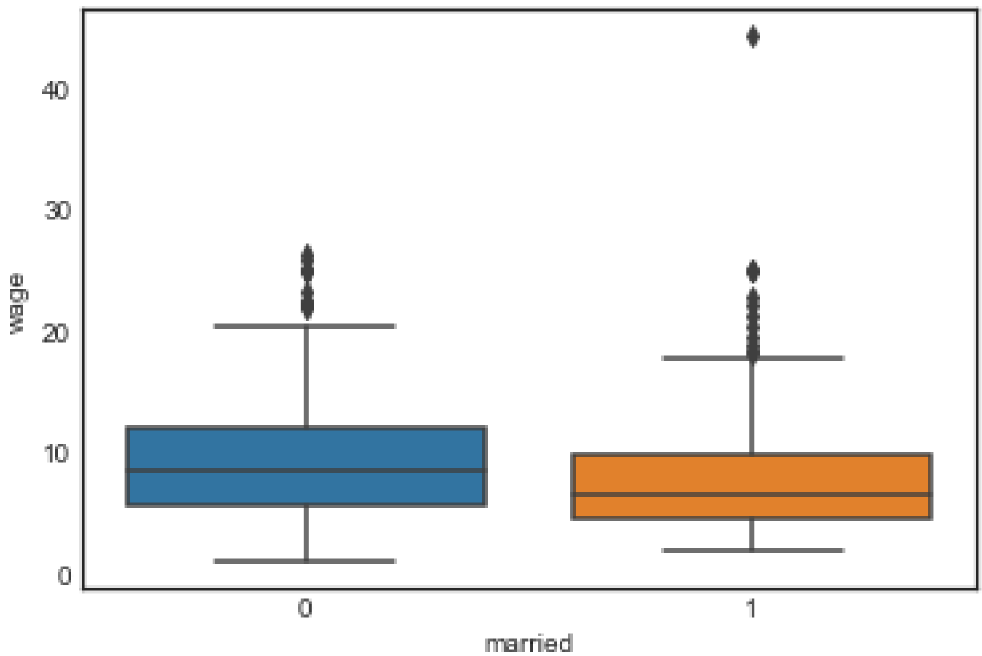
The boxplot indicates that most observations from regions other than the South earned more.

[45]:
```
sns.boxplot(x='union', y='wage', data=df);
```



The boxplot indicated that if an employee was part of the union (0), they earned slightly more wages compared to those not in the union.

[46]:
```
sns.boxplot(x='married', y='wage', data=df);
```



The boxplot indicates that employees who were married earned more than those who were not married.

[47]:
```
# Partition the data into training/testing samples
X = df.drop(['wage', 'wage01'], axis=1)
Y = df.wage01
X_train, X_test, Y_train, Y_test = train_test_split(X, Y,
                                                    train_size=0.5,
                                                    random_state=0)
```

[48]:
```
accuracy = []
for i in np.arange(1, 20):
    clf = DecisionTreeClassifier(max_depth=i)
    clf.fit(X_train, Y_train)
    pred = clf.predict(X_test)
    print(i)
    print("Accuracy score =", clf.score(X_train, Y_train))
    accuracy.append(clf.score(X_train, Y_train))
    print("RMSE =",np.sqrt(mean_squared_error(Y_test, pred)))
```

```
1
```

```
Accuracy score = 0.6329588014981273
RMSE = 0.6089224226894036
2
Accuracy score = 0.6891385767790262
RMSE = 0.5642269578149325
3
Accuracy score = 0.7715355805243446
RMSE = 0.5439486646851283
4
Accuracy score = 0.8052434456928839
RMSE = 0.5805847497871377
5
Accuracy score = 0.8202247191011236
RMSE = 0.5870002456468933
6
Accuracy score = 0.8277153558052435
RMSE = 0.5708263279047433
7
Accuracy score = 0.8726591760299626
RMSE = 0.6150423998051292
8
Accuracy score = 0.8951310861423221
RMSE = 0.6211020771678133
9
Accuracy score = 0.9250936329588015
RMSE = 0.6505543588223597
10
Accuracy score = 0.9438202247191011
RMSE = 0.6562862260377642
11
Accuracy score = 0.9588014981273408
RMSE = 0.6447715386698235
12
Accuracy score = 0.9662921348314607
RMSE = 0.6562862260377642
13
Accuracy score = 0.9737827715355806
RMSE = 0.6418605913487693
14
Accuracy score = 0.9812734082397003
RMSE = 0.6704015231539909
15
Accuracy score = 0.9812734082397003
RMSE = 0.6534265774628634
16
Accuracy score = 0.9812734082397003
RMSE = 0.6534265774628634
17
```

```
Accuracy score = 0.9850187265917603
RMSE = 0.6447715386698235
18
Accuracy score = 0.9850187265917603
RMSE = 0.647669402882915
19
Accuracy score = 0.9850187265917603
RMSE = 0.647669402882915
```

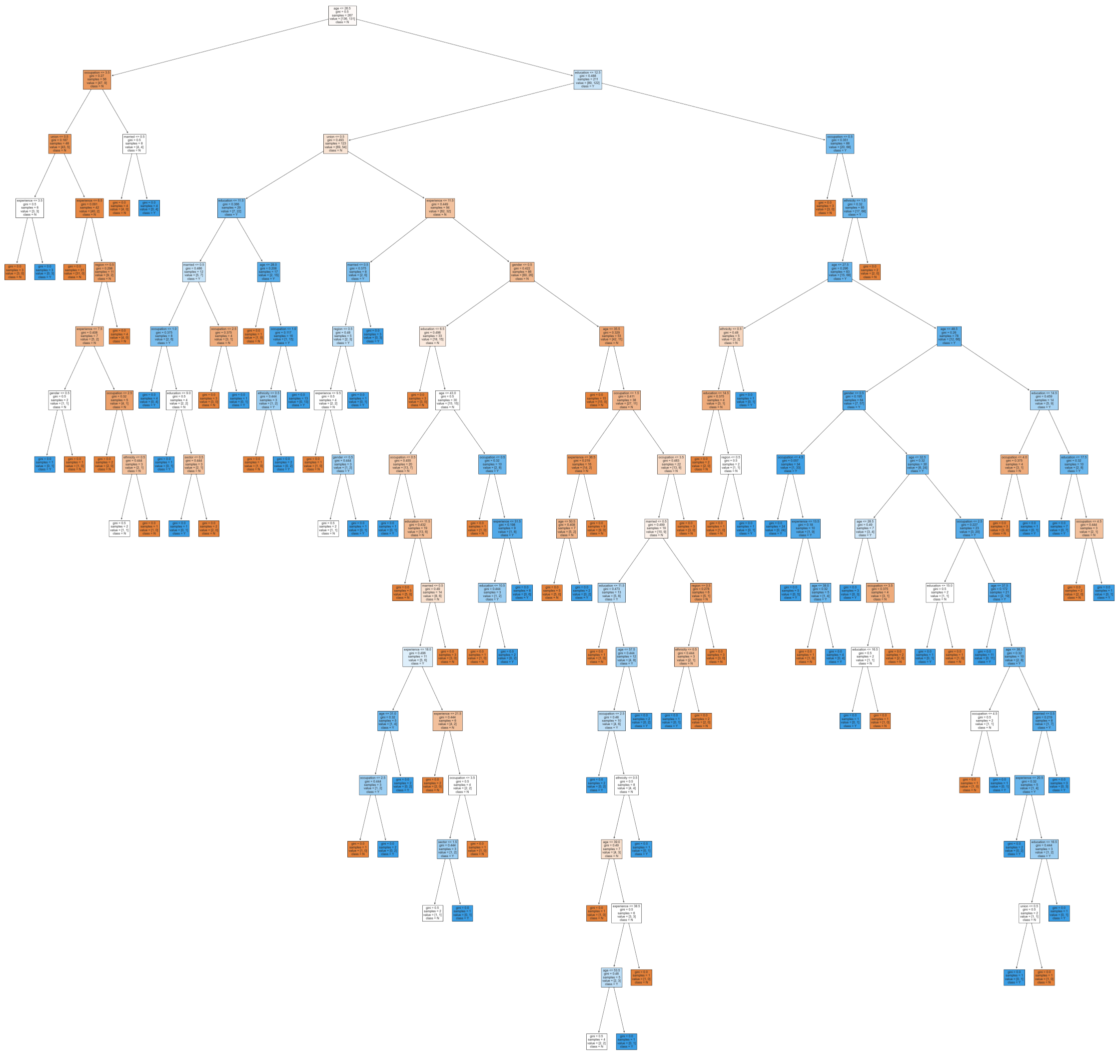
[49]:

```python
clf = DecisionTreeClassifier(max_depth=16)
clf.fit(X_train, Y_train)
print("Training Score =", clf.score(X_train, Y_train))
```

```
Training Score = 0.9812734082397003
```

[50]:

```python
fig, axes = plt.subplots(nrows = 1,ncols = 1,figsize = (60,60))
tree.plot_tree(clf,feature_names = X_train.columns,filled = True,
               class_names = ['N','Y']);
```



[51]:
```
# Evalaute the testing error
pred = clf.predict(X_test)
cm = pd.DataFrame(confusion_matrix(Y_test, pred).T,
                  index=['No', 'Yes'], columns=['No', 'Yes'])
print(cm)
print('Accuracy =',(81+73)/267)
```

```
     No  Yes
No   84   63
Yes  48   72
Accuracy = 0.5767790262172284
```

Even though the training score was highest for a depth of 16, the accuracy of the test is pretty low – so we will check the accuracy for other depths.

[52]:
```
clf1 = DecisionTreeClassifier(max_depth=3)
clf1.fit(X_train, Y_train)
print("Training Score =", clf1.score(X_train, Y_train))
```

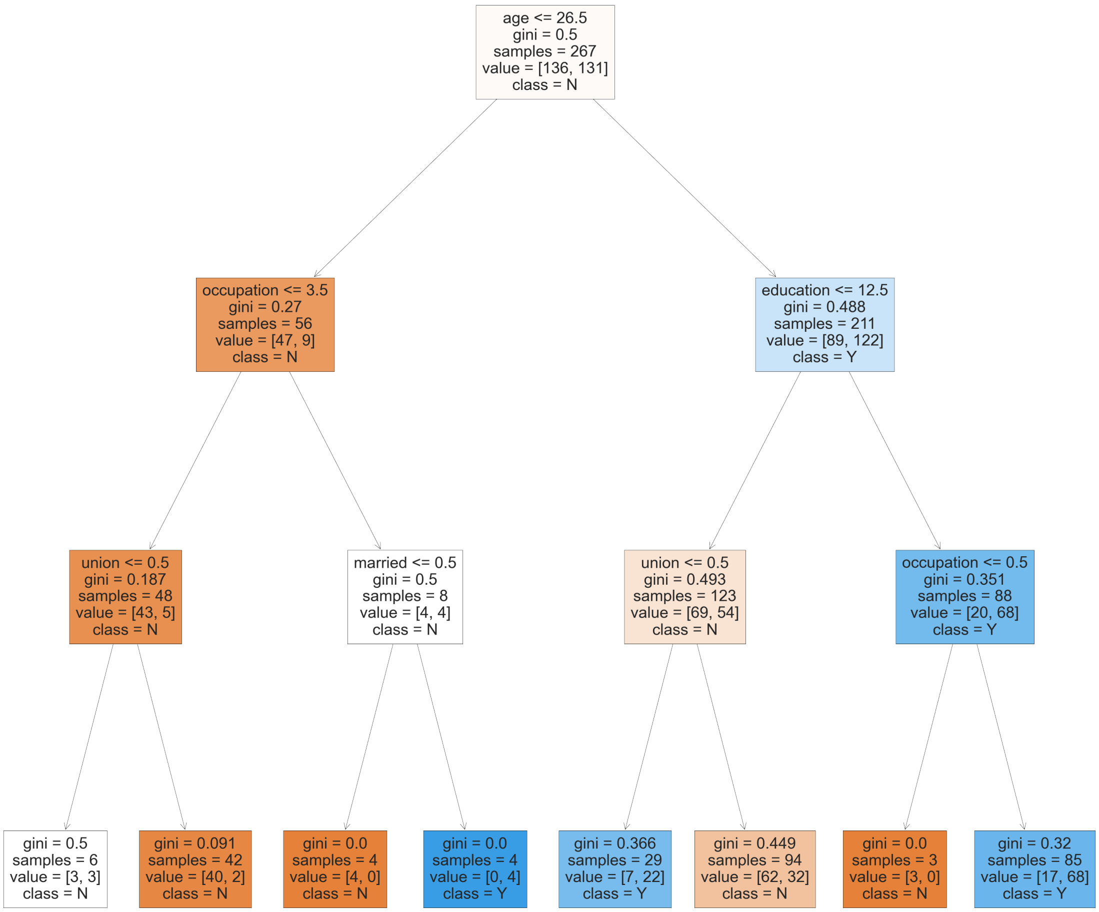
```
Training Score = 0.7715355805243446
```

[53]:
```
pred1 = clf1.predict(X_test)
cm1 = pd.DataFrame(confusion_matrix(Y_test, pred1).T,
                   index=['No', 'Yes'], columns=['No', 'Yes'])
print(cm1)
print('Accuracy =',(102+86)/267)
```

```
      No  Yes
No   102   49
Yes   30   86
Accuracy = 0.704119850187266
```

After predicting the test score for each depth, the model with depth = 3 achieved the highest accuracy.

[54]:
```
fig, axes = plt.subplots(nrows = 1,ncols = 1,figsize = (60,60))
tree.plot_tree(clf1,feature_names = X_train.columns,filled = True,
               class_names = ['N','Y']);
```



0.6 Regr tree

Use the original wage values as wages to run a continuous regression rather than using "greater than median" as our outcome variable.

[55]:
```
# Partition the data into training/testing samples
X = df.drop(['wage', 'wage01'], axis=1)
Y1 = df.wage
X_train, X_test, Y1_train, Y1_test = train_test_split(X, Y1,
                                                      train_size=0.5,
                                                      random_state=0)
```

[56]:
```
accuracy = []
for i in np.arange(1, 11):
```

```
    regr0 = RandomForestRegressor(max_features=i)
    regr0.fit(X_train, Y1_train)
    pred = regr0.predict(X_test)
    print(i)
    print("Accuracy score =", regr0.score(X_train, Y1_train))
    accuracy.append(regr0.score(X_train, Y1_train))
    print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred)))
```

```
1
Accuracy score = 0.8771712492562963
RMSE = 4.797250993545244
2
Accuracy score = 0.8845151098389485
RMSE = 4.769743099801751
3
Accuracy score = 0.8771412965675475
RMSE = 4.786395039657306
4
Accuracy score = 0.8862808468014931
RMSE = 4.807679933854359
5
Accuracy score = 0.8767966174742365
RMSE = 4.811754092456208
6
Accuracy score = 0.8707005434699369
RMSE = 4.871385190105981
7
Accuracy score = 0.876566371810634
RMSE = 4.876114305746892
8
Accuracy score = 0.8757264336661115
RMSE = 4.899257848981701
9
Accuracy score = 0.8768832374628507
RMSE = 4.9512892039403305
10
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
```

[57]:
```
pred = regr0.predict(X_test)
for i in np.arange(1, 20):
    print(i)
    print("Accuracy score =", regr0.score(X_train, Y1_train))
    accuracy.append(regr0.score(X_train, Y1_train))
    print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred)))
```

```
1
Accuracy score = 0.8748415681883012
```

```
RMSE = 4.960918649503328
2
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
3
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
4
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
5
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
6
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
7
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
8
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
9
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
10
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
11
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
12
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
13
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
14
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
15
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
16
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
17
Accuracy score = 0.8748415681883012
```

```
RMSE = 4.960918649503328
18
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
19
Accuracy score = 0.8748415681883012
RMSE = 4.960918649503328
```

[58]:

```
accuracy = []
error = []
for i in np.arange(1, 20):
    regr = DecisionTreeRegressor(max_depth=i)
    regr.fit(X_train, Y1_train)
    pred = regr.predict(X_test)
    print(i)
    print("Accuracy score =", regr.score(X_train, Y1_train))
    accuracy.append(regr.score(X_train, Y1_train))
    print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred)))
    error.append(np.sqrt(mean_squared_error(Y1_test, pred)))
```

```
1
Accuracy score = 0.21298870685346072
RMSE = 4.962352645654619
2
Accuracy score = 0.2960296838036227
RMSE = 4.9214452205606385
3
Accuracy score = 0.3878286543093107
RMSE = 5.336321932939656
4
Accuracy score = 0.4851408021708673
RMSE = 5.365029076769744
5
Accuracy score = 0.5551015252438707
RMSE = 5.568189726657421
6
Accuracy score = 0.6540924159213188
RMSE = 5.508642093997996
7
Accuracy score = 0.7381476988037567
RMSE = 5.875095044183314
8
Accuracy score = 0.823252043277398
RMSE = 6.20007829997982
9
Accuracy score = 0.8785757074707911
RMSE = 6.056760522303408
10
```

```
Accuracy score = 0.9169878064220927
RMSE = 6.256312811576195
11
Accuracy score = 0.9438158038819826
RMSE = 6.291264510662417
12
Accuracy score = 0.95939305427095
RMSE = 6.568565725291481
13
Accuracy score = 0.9700384268657609
RMSE = 6.336231157059932
14
Accuracy score = 0.9734912478687461
RMSE = 6.634667076812633
15
Accuracy score = 0.9750629883613565
RMSE = 6.660677475918064
16
Accuracy score = 0.9750647088091416
RMSE = 6.4201659411272605
17
Accuracy score = 0.9750647088091416
RMSE = 6.347060955397828
18
Accuracy score = 0.9750647088091416
RMSE = 6.712276834251211
19
Accuracy score = 0.9750647088091416
RMSE = 6.576907991343026
```

[59]:

```
regr1 = DecisionTreeRegressor(max_depth=1)
regr1.fit(X_train, Y1_train)
pred2 = regr1.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred2)))
```

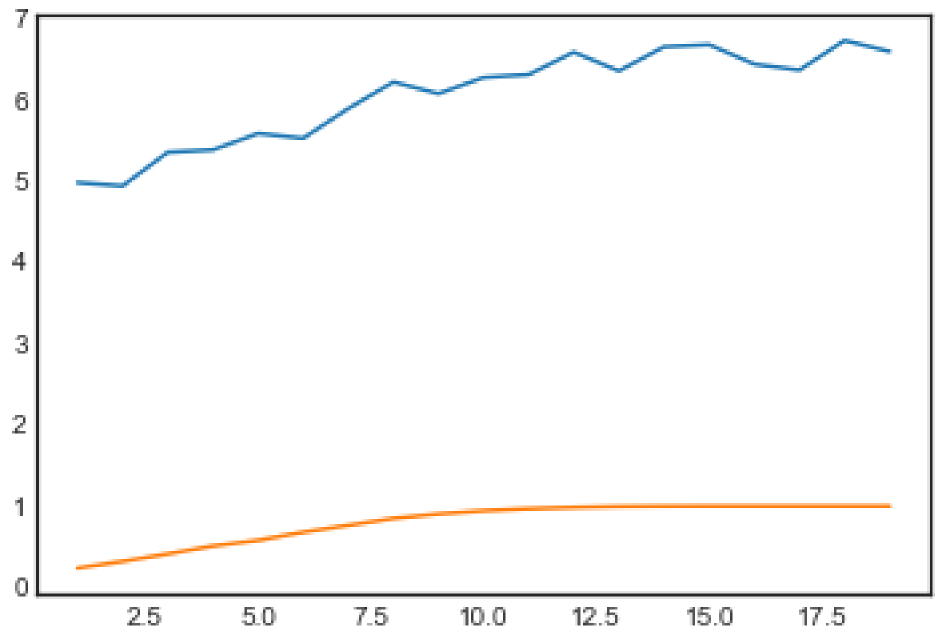
```
RMSE = 4.962352645654619
```

[60]:

```
plt.plot(np.arange(1,20),error)
plt.plot(np.arange(1,20),accuracy)
```

[60]:

```
[<matplotlib.lines.Line2D at 0x286100d1730>]
```



[61]:

```
regr2 = DecisionTreeRegressor(max_depth=2)
regr2.fit(X_train, Y1_train)
pred3 = regr2.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred3)))

regr3 = DecisionTreeRegressor(max_depth=3)
regr3.fit(X_train, Y1_train)
pred4 = regr3.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred4)))

regr4 = DecisionTreeRegressor(max_depth=4)
regr4.fit(X_train, Y1_train)
pred5 = regr4.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred5)))

regr5 = DecisionTreeRegressor(max_depth=5)
regr5.fit(X_train, Y1_train)
pred6 = regr5.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred6)))

regr6 = DecisionTreeRegressor(max_depth=6)
regr6.fit(X_train, Y1_train)
pred7 = regr6.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred7)))

regr7 = DecisionTreeRegressor(max_depth=7)
```

```
regr7.fit(X_train, Y1_train)
pred8 = regr7.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred8)))

regr8 = DecisionTreeRegressor(max_depth=8)
regr8.fit(X_train, Y1_train)
pred9 = regr8.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred9)))

regr9 = DecisionTreeRegressor(max_depth=9)
regr9.fit(X_train, Y1_train)
pred10 = regr9.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred10)))

regr10 = DecisionTreeRegressor(max_depth=10)
regr10.fit(X_train, Y1_train)
pred11 = regr10.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred11)))

regr11 = DecisionTreeRegressor(max_depth=11)
regr11.fit(X_train, Y1_train)
pred12 = regr11.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred12)))

regr12 = DecisionTreeRegressor(max_depth=12)
regr12.fit(X_train, Y1_train)
pred13 = regr12.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred13)))

regr13 = DecisionTreeRegressor(max_depth=13)
regr13.fit(X_train, Y1_train)
pred14 = regr13.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred14)))

regr14 = DecisionTreeRegressor(max_depth=14)
regr14.fit(X_train, Y1_train)
pred15 = regr14.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred15)))

regr15 = DecisionTreeRegressor(max_depth=15)
regr15.fit(X_train, Y1_train)
pred16 = regr15.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred16)))

regr16 = DecisionTreeRegressor(max_depth=16)
regr16.fit(X_train, Y1_train)
pred17 = regr16.predict(X_test)
```

```
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred17)))

regr17 = DecisionTreeRegressor(max_depth=17)
regr17.fit(X_train, Y1_train)
pred18 = regr17.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred18)))

regr18 = DecisionTreeRegressor(max_depth=18)
regr18.fit(X_train, Y1_train)
pred19 = regr18.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred19)))

regr19 = DecisionTreeRegressor(max_depth=19)
regr19.fit(X_train, Y1_train)
pred20 = regr19.predict(X_test)
print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred20)))
```

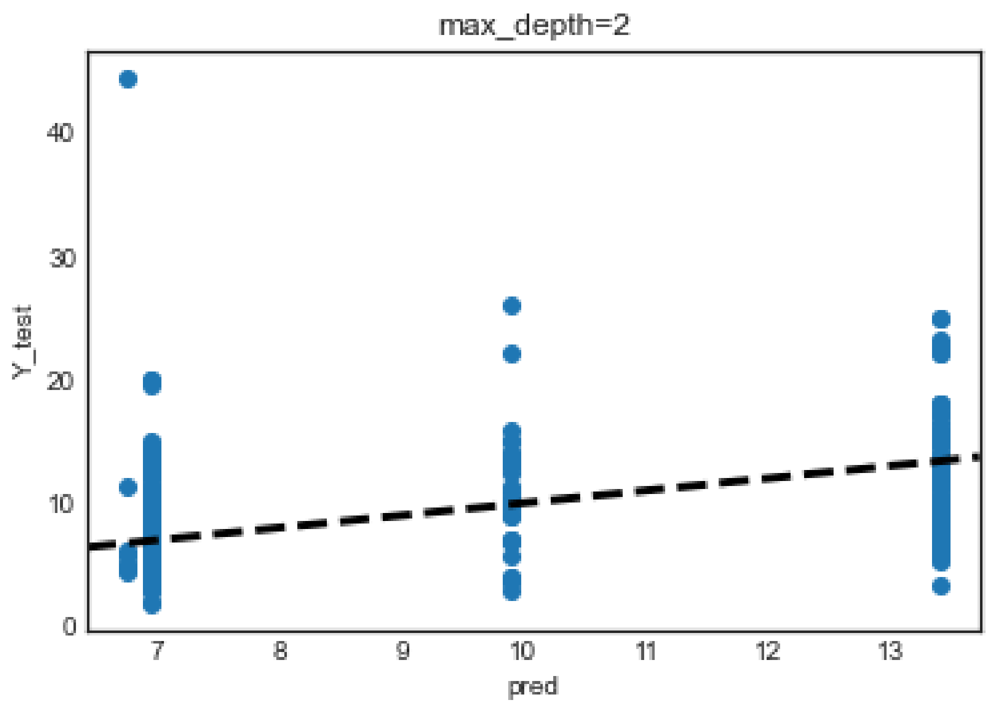
```
RMSE = 4.9214452205606385
RMSE = 5.336321932939656
RMSE = 5.365029076769743
RMSE = 5.590418831554847
RMSE = 5.599915165219513
RMSE = 6.021249015841448
RMSE = 5.863034263948096
RMSE = 6.394331114574081
RMSE = 6.558515707372336
RMSE = 6.3548644012953375
RMSE = 6.313467174708813
RMSE = 6.380678914121804
RMSE = 6.582954914138586
RMSE = 6.632315260270013
RMSE = 6.5457620735553395
RMSE = 6.376811749604
RMSE = 6.322434528786085
RMSE = 6.432710299535711
```

[62]:

```
plt.scatter(pred3, Y1_test, label='wages')
xpoints = ypoints = plt.xlim()
plt.plot(xpoints, ypoints, linestyle='--', color='k', lw=3, scalex=False,␣
 ↪scaley=False)
plt.xlabel('pred')
plt.ylabel('Y_test')
plt.title('max_depth=2')
```

[62]:

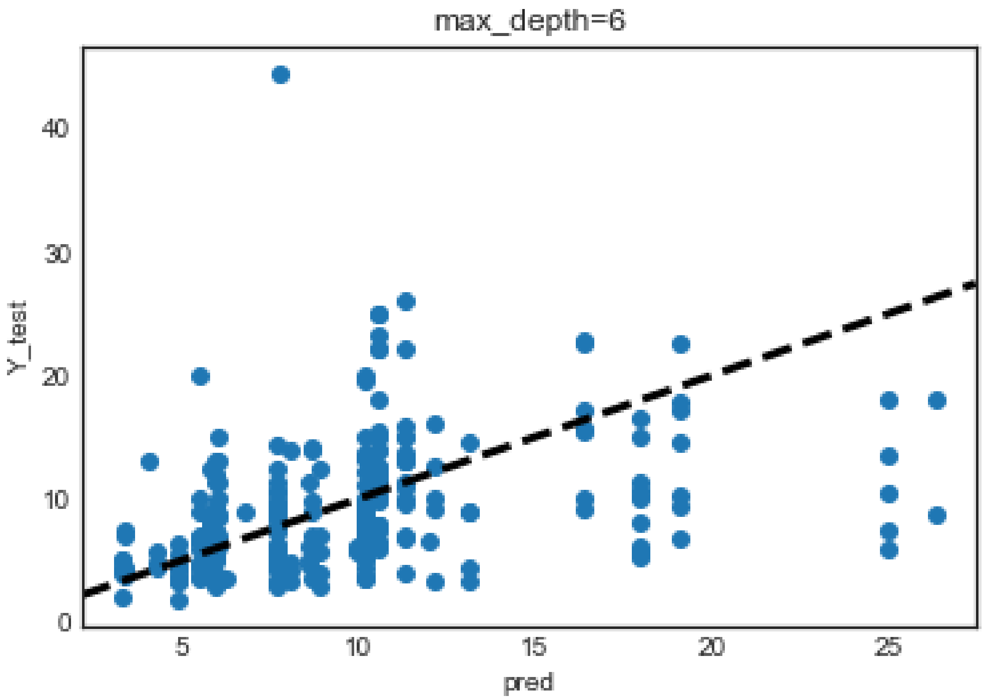
```
Text(0.5, 1.0, 'max_depth=2')
```



RMSE is lowest with depth=2, but has a very low accuracy and the graph shows that the predictions are inaccurate, so we will make our decision based on the plots, RMSEs and accuracies of other depths.

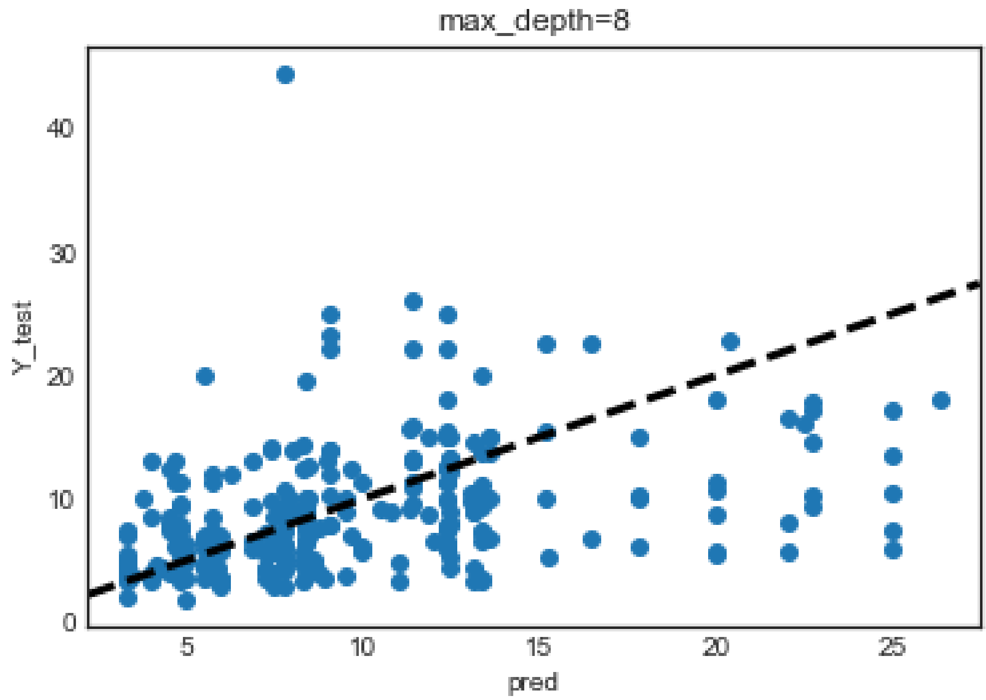
[63]:
```
plt.scatter(pred7, Y1_test, label='wages')
xpoints = ypoints = plt.xlim()
plt.plot(xpoints, ypoints, linestyle='--', color='k', lw=3, scalex=False,␣
 ↪scaley=False)
plt.xlabel('pred')
plt.ylabel('Y_test')
plt.title('max_depth=6')
```

[63]:
```
Text(0.5, 1.0, 'max_depth=6')
```



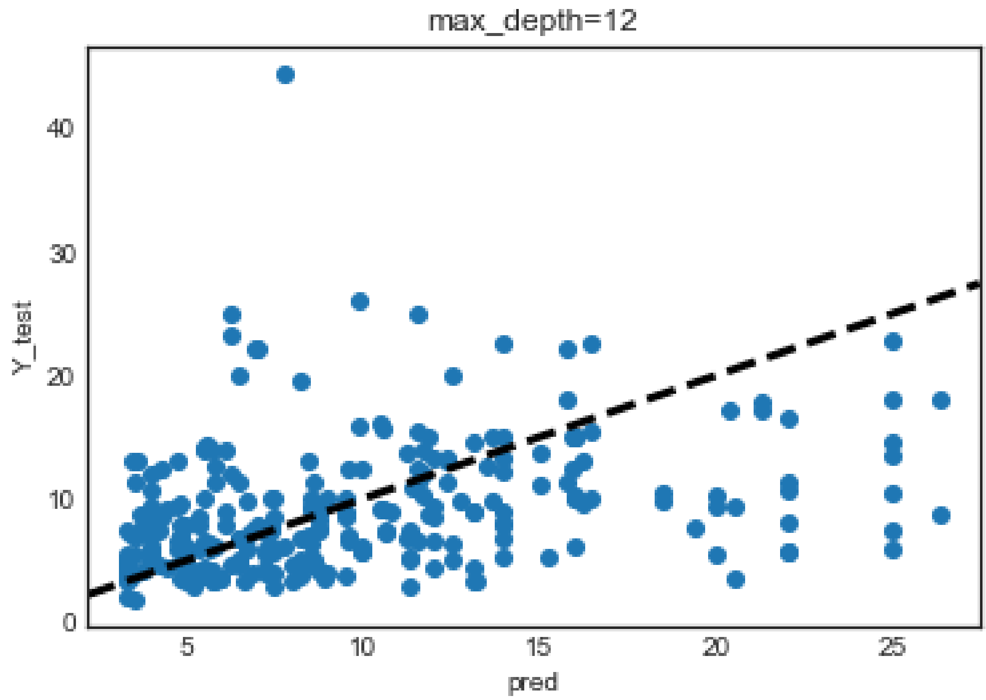
[64]:
```
plt.scatter(pred9, Y1_test, label='wages')
xpoints = ypoints = plt.xlim()
plt.plot(xpoints, ypoints, linestyle='--', color='k', lw=3, scalex=False,␣
 ↪scaley=False)
plt.xlabel('pred')
plt.ylabel('Y_test')
plt.title('max_depth=8')
```

[64]:
```
Text(0.5, 1.0, 'max_depth=8')
```



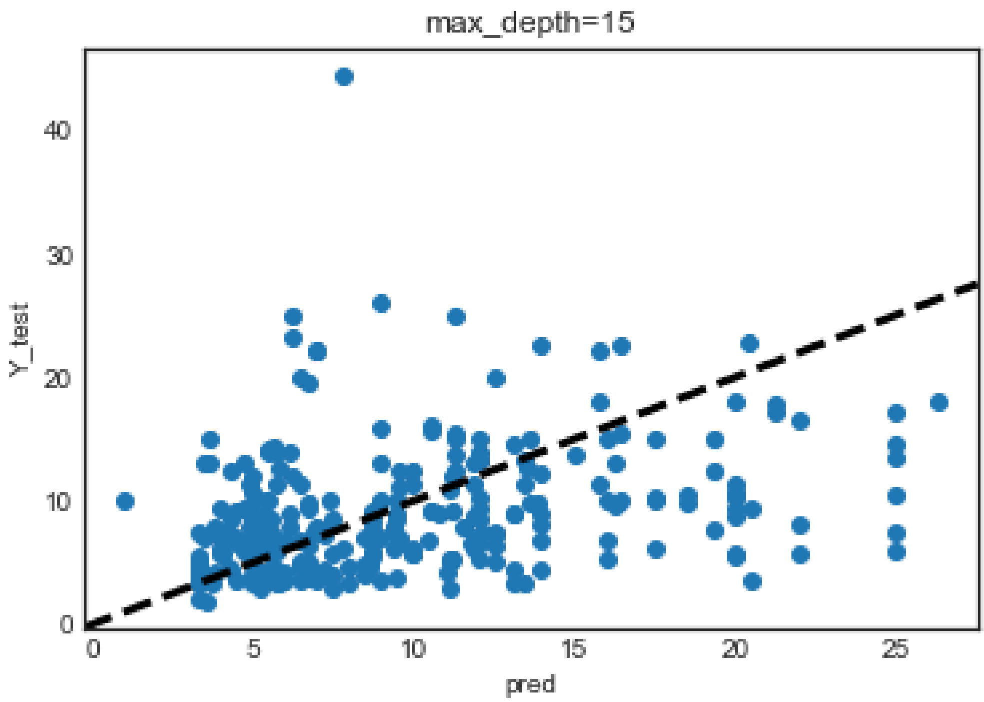
[65]:
```
plt.scatter(pred13, Y1_test, label='wages')
xpoints = ypoints = plt.xlim()
plt.plot(xpoints, ypoints, linestyle='--', color='k', lw=3, scalex=False,␣
 ↪scaley=False)
plt.xlabel('pred')
plt.ylabel('Y_test')
plt.title('max_depth=12')
```

[65]:
```
Text(0.5, 1.0, 'max_depth=12')
```



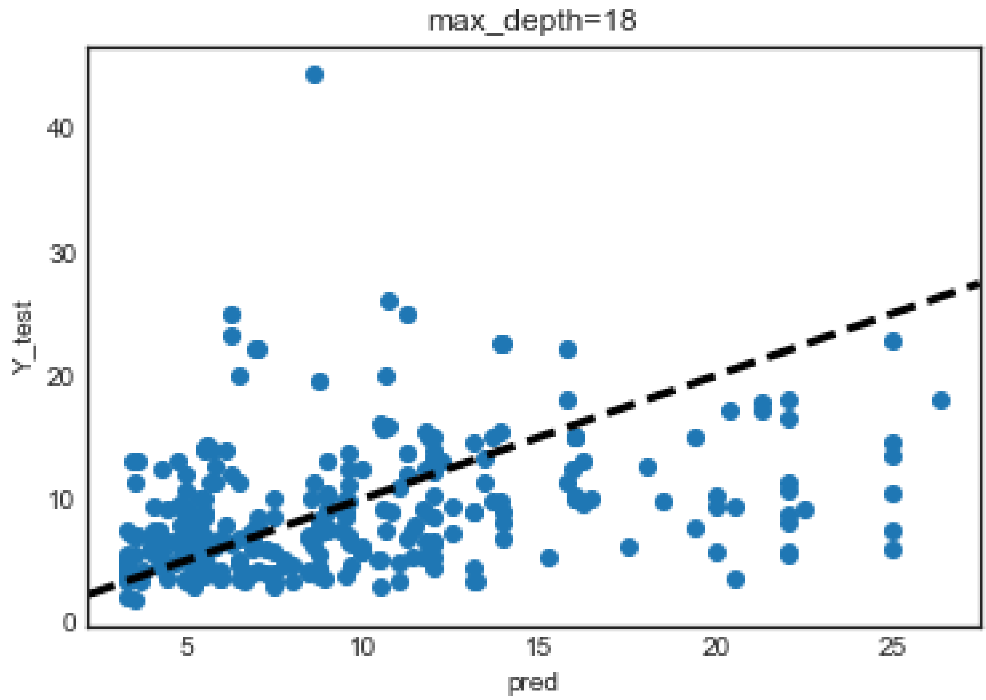

```
[66]: plt.scatter(pred14, Y1_test, label='wages')
      xpoints = ypoints = plt.xlim()
      plt.plot(xpoints, ypoints, linestyle='--', color='k', lw=3, scalex=False,␣
       ↪scaley=False)
      plt.xlabel('pred')
      plt.ylabel('Y_test')
      plt.title('max_depth=15')
```

```
[66]: Text(0.5, 1.0, 'max_depth=15')
```



[67]:
```
plt.scatter(pred19, Y1_test, label='wages')
xpoints = ypoints = plt.xlim()
plt.plot(xpoints, ypoints, linestyle='--', color='k', lw=3, scalex=False,␣
 ↪scaley=False)
plt.xlabel('pred')
plt.ylabel('Y_test')
plt.title('max_depth=18')
```

[67]: Text(0.5, 1.0, 'max_depth=18')



Based on the RMSEs depth=8 has the lowest, but the plot of depth=18 seems to offer slightly better predictions. We decided to choose depth=8 as the optimal to avoid overfitting the model.

0.7 Random forest & Bagging

[68]:
```
accuracy = []
for i in np.arange(1, 11):
    regr0 = RandomForestRegressor(max_features=i)
    regr0.fit(X_train, Y1_train)
    pred = regr0.predict(X_test)
    print(i)
    print("Accuracy score =", regr0.score(X_train, Y1_train))
    accuracy.append(regr0.score(X_train, Y1_train))
    print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred)))
```

```
1
Accuracy score = 0.8840197444667879
RMSE = 4.8530706705504665
2
Accuracy score = 0.88172907990898
RMSE = 4.796649612256762
3
Accuracy score = 0.8758224911565393
RMSE = 4.823876914308358
4
```

```
Accuracy score = 0.8811731837768809
RMSE = 4.848083767714674
5
Accuracy score = 0.8841648045858205
RMSE = 4.823423102064894
6
Accuracy score = 0.8723231499880906
RMSE = 4.855597939851262
7
Accuracy score = 0.8806130101207063
RMSE = 4.874763860486386
8
Accuracy score = 0.8754811938642615
RMSE = 4.8910405830261645
9
Accuracy score = 0.8782300333184291
RMSE = 4.8649861202157405
10
Accuracy score = 0.8769940217572525
RMSE = 4.97151827631581
```

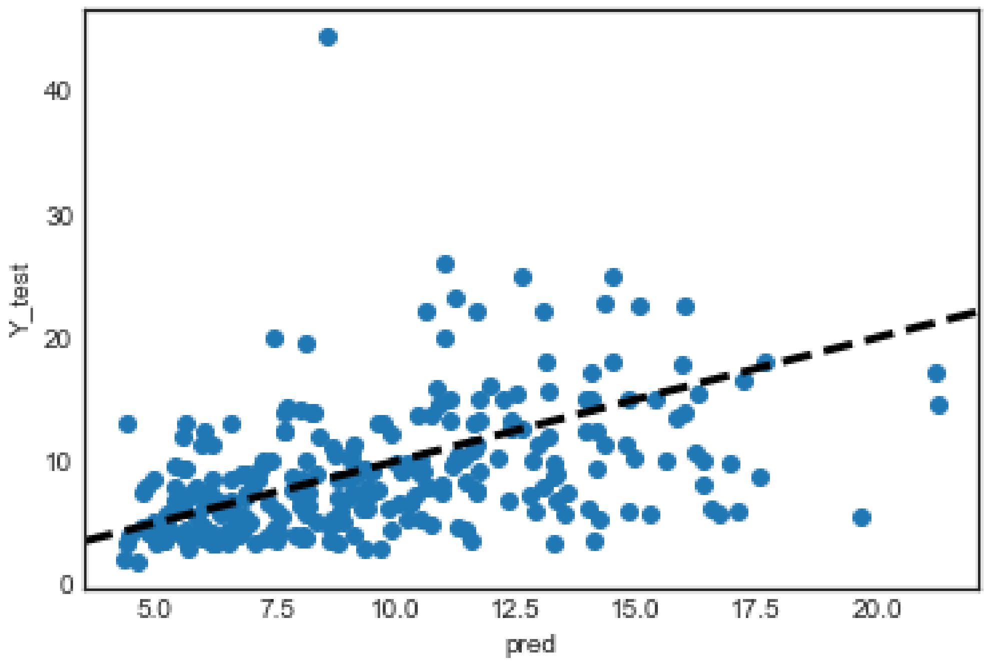
According to the training score & the test RMSE, 2 features has the highest training score and lowest test RMSE, so we condlude that its optimal.

[69]:
```
regr20 = RandomForestRegressor(max_features=10, random_state=1) #BAGGING
regr20.fit(X_train, Y1_train)
```

[69]:
```
RandomForestRegressor(max_features=10, random_state=1)
```

[70]:
```
pred21 = regr20.predict(X_test)
plt.scatter(pred21, Y1_test, label='medv')
xpoints = ypoints = plt.xlim()
plt.plot(xpoints, ypoints, linestyle='--', color='k', lw=3, scalex=False,␣
 ↪scaley=False)
plt.xlabel('pred')
plt.ylabel('Y_test')
print("Bagging RMSE =",np.sqrt(mean_squared_error(Y1_test, pred21)))
```

```
Bagging RMSE = 4.9780619996926525
```

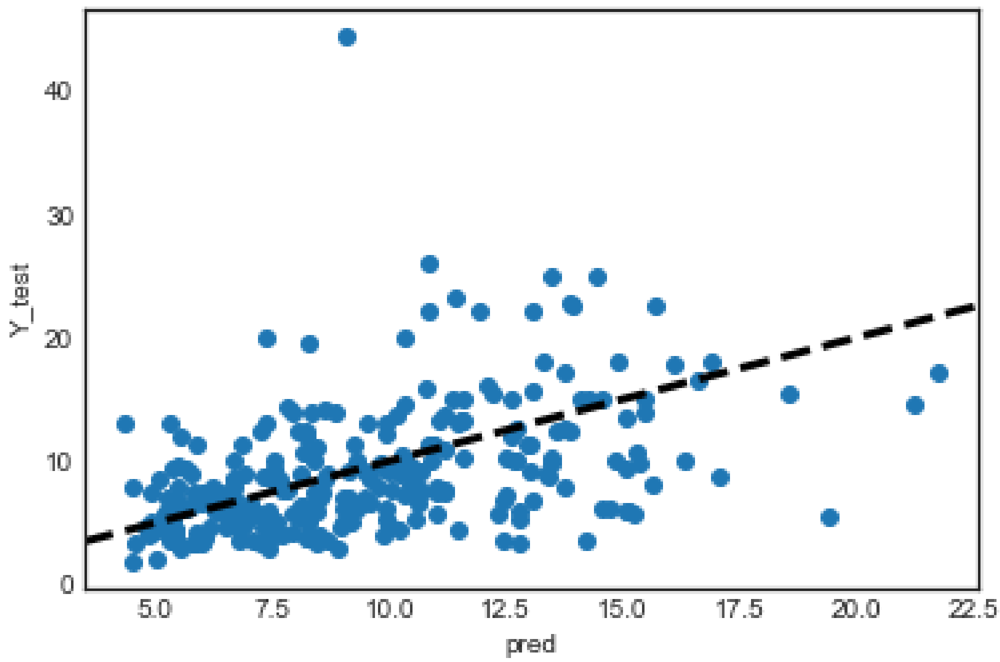


[71]:
```
regr26 = RandomForestRegressor(max_features=4, random_state=1)
regr26.fit(X_train, Y1_train)

pred27 = regr26.predict(X_test)
print("RF RMSE =",np.sqrt(mean_squared_error(Y1_test, pred27)))
```

```
RF RMSE = 4.835128056781248
```

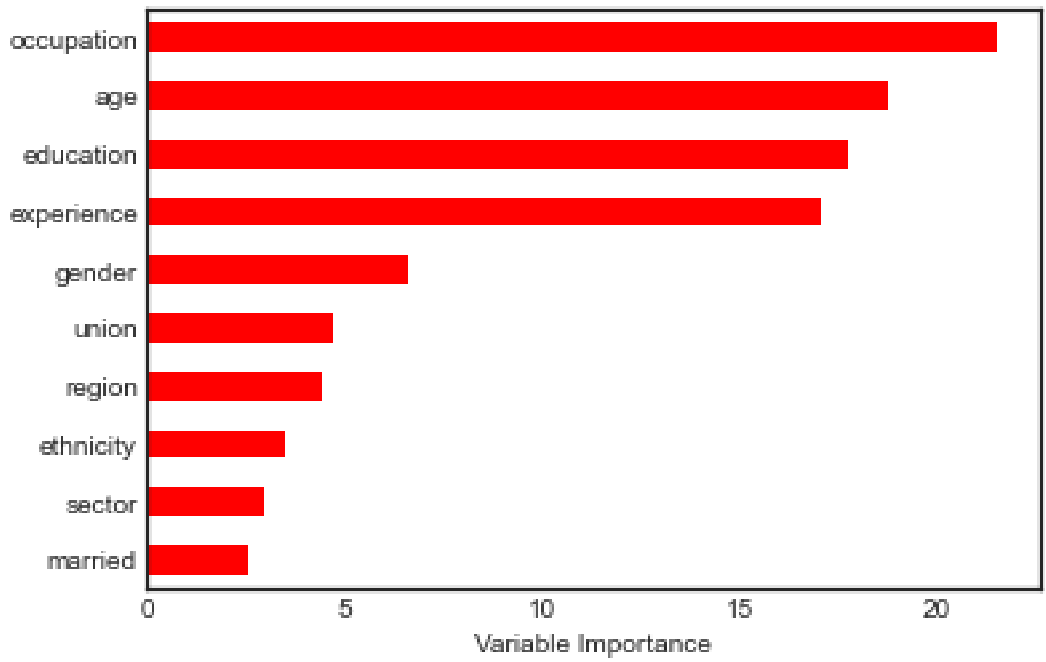
[72]:
```
plt.scatter(pred27, Y1_test, label='medv')
xpoints = ypoints = plt.xlim()
plt.plot(xpoints, ypoints, linestyle='--', color='k', lw=3, scalex=False,␣
 ↪scaley=False)
plt.xlabel('pred')
plt.ylabel('Y_test')
print("Bagging RMSE =",np.sqrt(mean_squared_error(Y1_test, pred21)))
```

```
Bagging RMSE = 4.9780619996926525
```



[73]:

```
Importance = pd.DataFrame({'Importance':regr26.feature_importances_*100},␣
 ↪index=X.columns)
Importance.sort_values(by='Importance', axis=0, ascending=True).
 ↪plot(kind='barh', color='r', )
plt.xlabel('Variable Importance')
plt.gca().legend_ = None
```

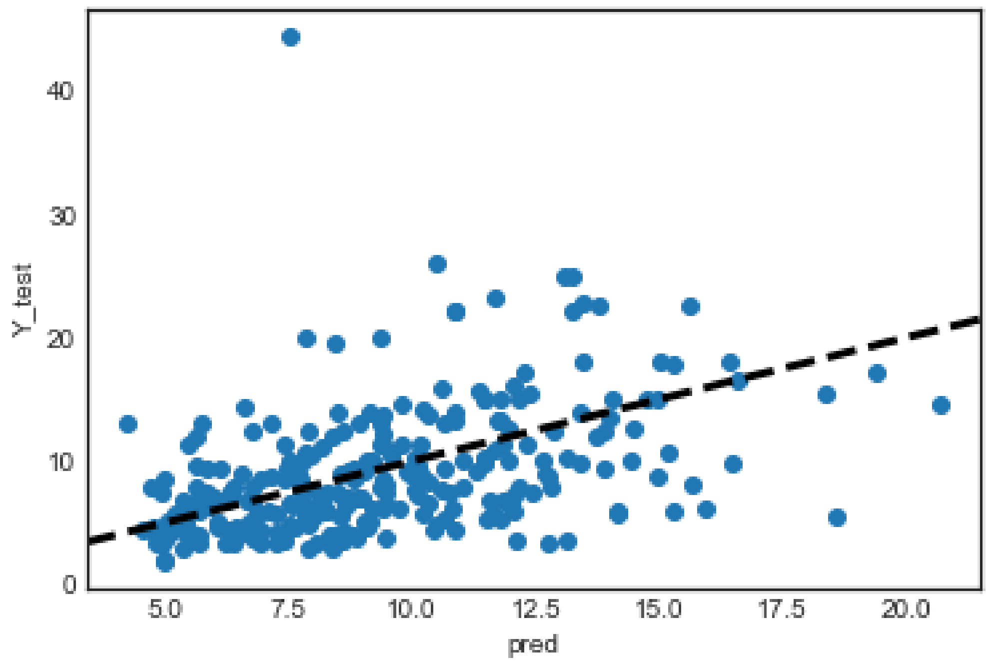


[74]:
```
regr28 = RandomForestRegressor(max_features=2, random_state=1)
regr28.fit(X_train, Y1_train)

pred29 = regr28.predict(X_test)
print("RF RMSE =",np.sqrt(mean_squared_error(Y1_test, pred29)))
```

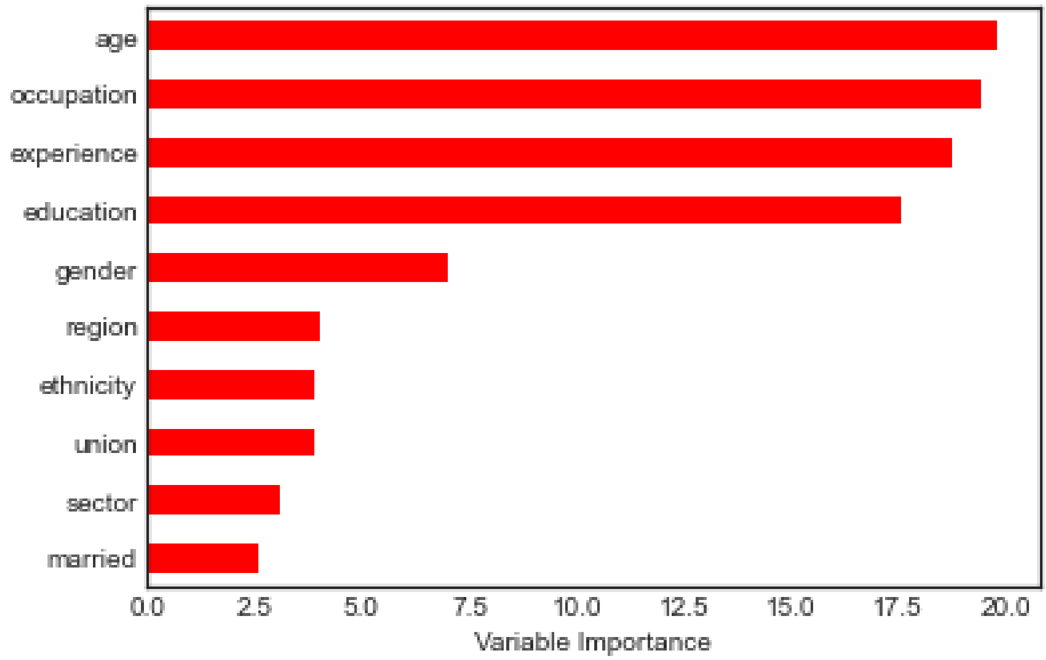
```
RF RMSE = 4.825932756362598
```

[75]:
```
plt.scatter(pred29, Y1_test, label='medv')
xpoints = ypoints = plt.xlim()
plt.plot(xpoints, ypoints, linestyle='--', color='k', lw=3, scalex=False,␣
 ↪scaley=False)
plt.xlabel('pred')
plt.ylabel('Y_test')
```

[75]:
```
Text(0, 0.5, 'Y_test')
```



[76]:
```
Importance = pd.DataFrame({'Importance':regr28.feature_importances_*100},␣
 ↪index=X.columns)
Importance.sort_values(by='Importance', axis=0, ascending=True).
 ↪plot(kind='barh', color='r', )
plt.xlabel('Variable Importance')
plt.gca().legend_ = None
```



Based on the scatter plots above, we can observe that using max_features = 2 performs well in predicting our Y_test. The importance of the features are also pretty consistent with our results from the classification tree.

0.8 BOOSTING

[77]:
```
accuracy = []
for i in np.arange(1, 20):
    regr = GradientBoostingRegressor(max_depth=i)
    regr.fit(X_train, Y1_train)
    pred = regr.predict(X_test)

    print(i)
    print("Accuracy score =", regr.score(X_train, Y1_train))
    accuracy.append(regr.score(X_train, Y1_train))
    print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred)))
```

```
1
Accuracy score = 0.4355657655728188
RMSE = 4.602711528188237
2
Accuracy score = 0.6155862663156659
RMSE = 4.676651034626421
3
Accuracy score = 0.778478491908961
RMSE = 5.030492558205867
```

```
4
Accuracy score = 0.89245710022546
RMSE = 5.108732751974454
5
Accuracy score = 0.9421229000266884
RMSE = 5.2633915228452
6
Accuracy score = 0.9669602293352692
RMSE = 5.366823124082164
7
Accuracy score = 0.9736973631455613
RMSE = 5.524954365336468
8
Accuracy score = 0.974911267762452
RMSE = 5.45647247355314
9
Accuracy score = 0.9750452332803955
RMSE = 5.641303909732654
10
Accuracy score = 0.9750632606186433
RMSE = 5.606385705525622
11
Accuracy score = 0.9750646625865518
RMSE = 5.620555900707193
12
Accuracy score = 0.9750647047612724
RMSE = 5.667244592230365
13
Accuracy score = 0.9750647070161222
RMSE = 6.217520061332713
14
Accuracy score = 0.9750647078563979
RMSE = 6.336732757460862
15
Accuracy score = 0.975064708120891
RMSE = 6.343788116898644
16
Accuracy score = 0.9750647081212257
RMSE = 6.365507761179617
17
Accuracy score = 0.9750647081212257
RMSE = 6.361501075625979
18
Accuracy score = 0.9750647081212257
RMSE = 6.38393298617441
19
Accuracy score = 0.9750647081212257
RMSE = 6.355049378348475
```

Accuracies are increasing as the depth increases, so it's not the sole metric we will utilize to decide our depth. Therefore, we decided to choose a depth that offers a good accuracy, not necessarily the highest, but still provides a relatively low test RMSE, keeping in mind that we want to avoid overfitting our model by simply increasing depths. We will try fitting different combinations of depths with the optimal learning rate we are about to obtain.

[78]:
```
for i in np.arange(1, 11):
    regr = GradientBoostingRegressor(learning_rate=i/100)
    regr.fit(X_train, Y1_train)
    pred = regr.predict(X_test)
    print(i/100)
    print("RMSE =",np.sqrt(mean_squared_error(Y1_test, pred)))
```

```
0.01
RMSE = 4.790962015613647
0.02
RMSE = 4.744301433906062
0.03
RMSE = 4.747178614467223
0.04
RMSE = 4.789766874948777
0.05
RMSE = 4.798844366963399
0.06
RMSE = 4.861875204384497
0.07
RMSE = 4.837302413744835
0.08
RMSE = 4.8590144748876645
0.09
RMSE = 4.8568766874483
0.1
RMSE = 5.0213672819940305
```

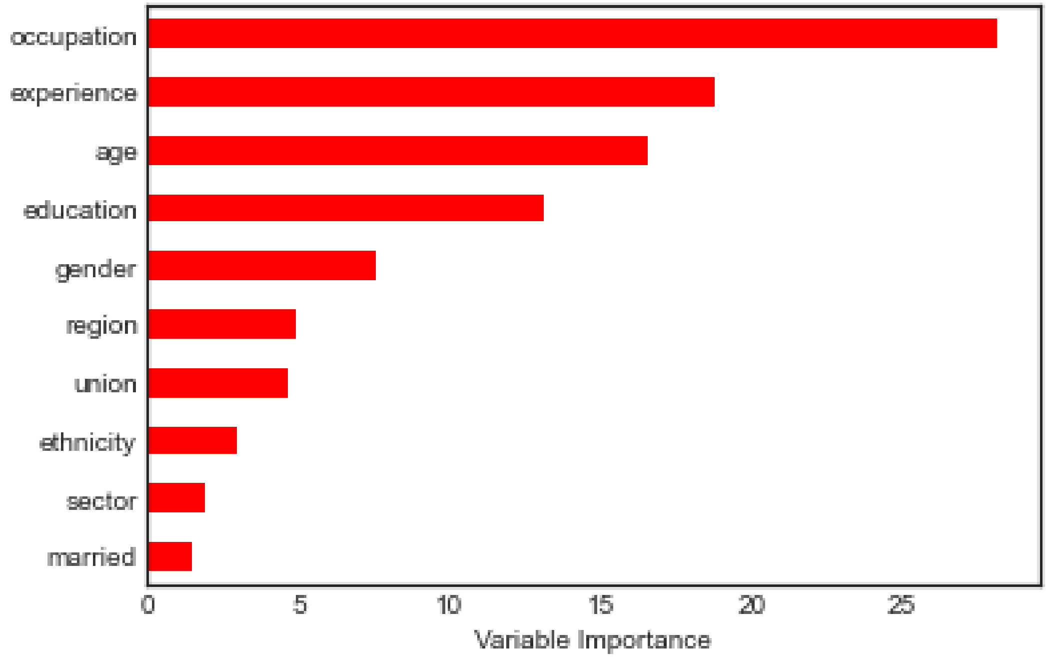
According to the RMSE of each learning rate, learning rate 0.02 has the lowest.

[79]:
```
regr30 = GradientBoostingRegressor(n_estimators=500, learning_rate=0.02,␣
 ↪max_depth=5, random_state=1)
regr30.fit(X_train, Y1_train)
print("Boosting RMSE =",np.sqrt(mean_squared_error(Y1_test, regr30.
 ↪predict(X_test))))
```

```
Boosting RMSE = 5.266461707713858
```

[80]:
```
feature_importance = regr30.feature_importances_*100
rel_imp = pd.Series(feature_importance, index=X.columns).
 ↪sort_values(inplace=False)
rel_imp.T.plot(kind='barh', color='r', )
plt.xlabel('Variable Importance')
```

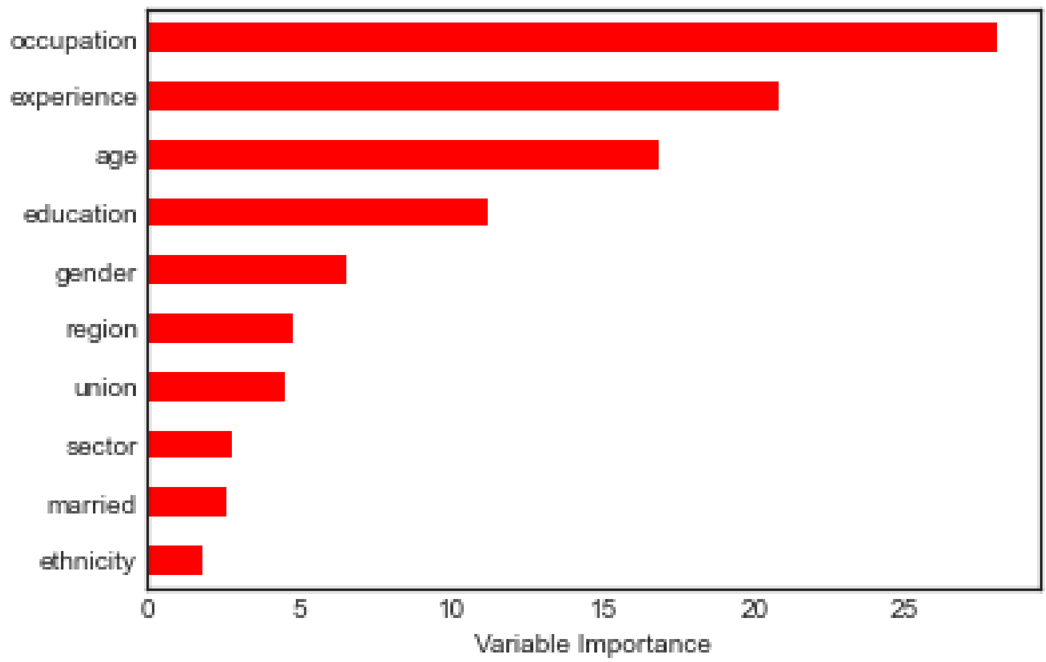
```
plt.gca().legend_ = None
```




```
[81]: regr31 = GradientBoostingRegressor(n_estimators=500, learning_rate=0.02,␣
      ↪max_depth=6, random_state=1)
      regr31.fit(X_train, Y1_train)
      print("Boosting RMSE =",np.sqrt(mean_squared_error(Y1_test, regr31.
      ↪predict(X_test))))
```

Boosting RMSE = 5.396882685425097

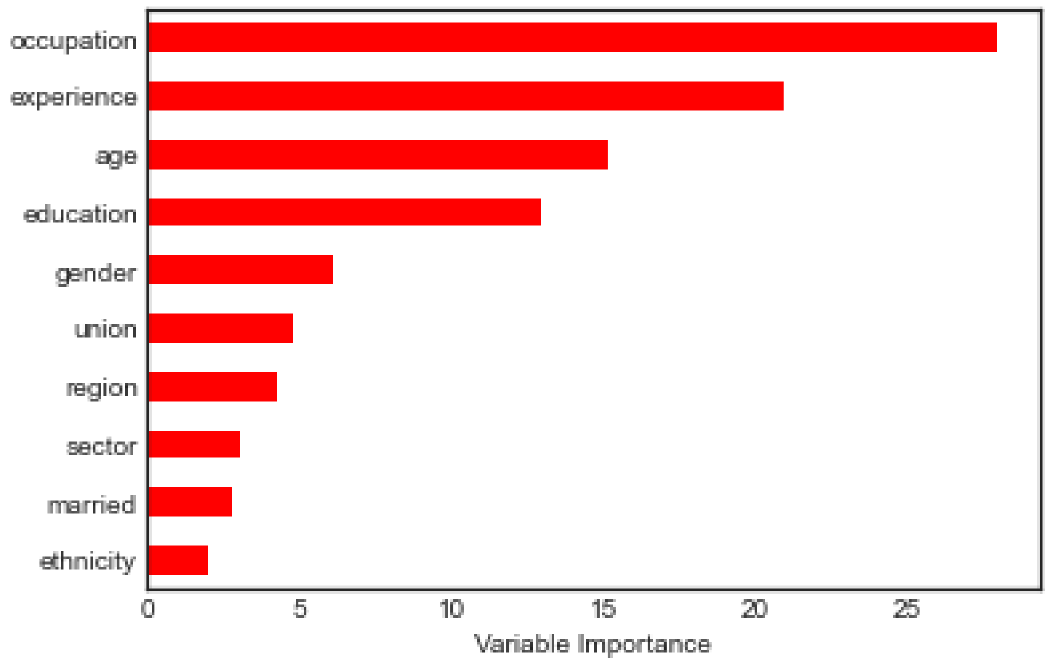
```
[82]: feature_importance = regr31.feature_importances_*100
      rel_imp = pd.Series(feature_importance, index=X.columns).
      ↪sort_values(inplace=False)
      rel_imp.T.plot(kind='barh', color='r', )
      plt.xlabel('Variable Importance')
      plt.gca().legend_ = None
```



[83]:
```
regr32 = GradientBoostingRegressor(n_estimators=500, learning_rate=0.02,␣
 ↪max_depth=7, random_state=1)
regr32.fit(X_train, Y1_train)
print("Boosting RMSE =",np.sqrt(mean_squared_error(Y1_test, regr32.
 ↪predict(X_test))))
```

```
Boosting RMSE = 5.4962543923389555
```

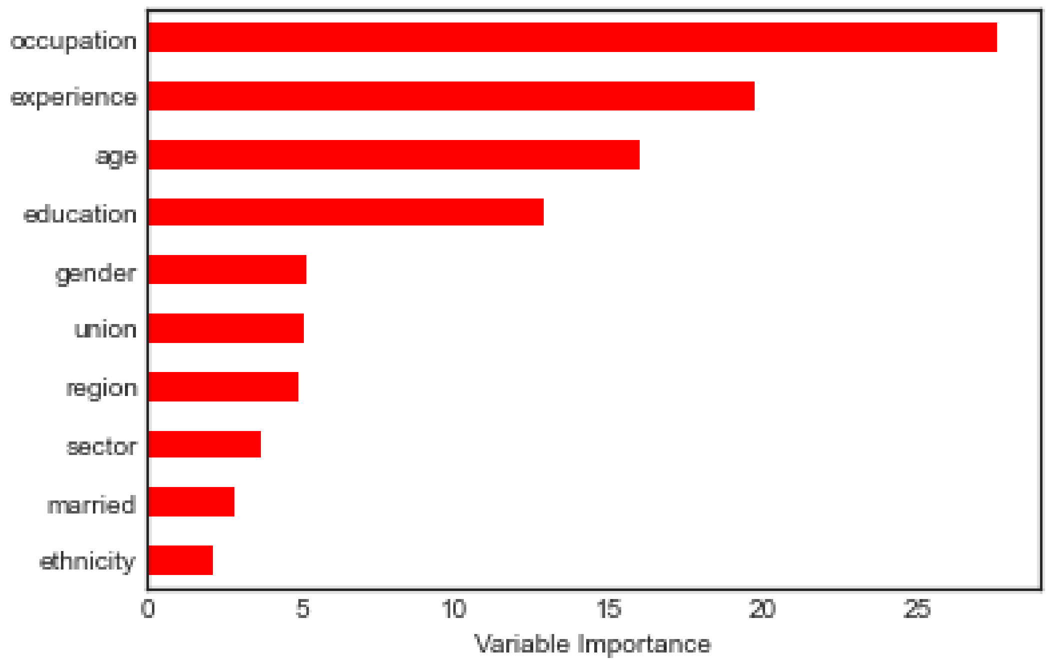
[84]:
```
feature_importance = regr32.feature_importances_*100
rel_imp = pd.Series(feature_importance, index=X.columns).
 ↪sort_values(inplace=False)
rel_imp.T.plot(kind='barh', color='r', )
plt.xlabel('Variable Importance')
plt.gca().legend_ = None
```



[85]:
```
regr33 = GradientBoostingRegressor(n_estimators=500, learning_rate=0.02,␣
↪max_depth=8, random_state=1)
regr33.fit(X_train, Y1_train)
print("Boosting RMSE =",np.sqrt(mean_squared_error(Y1_test, regr33.
↪predict(X_test))))
```

```
Boosting RMSE = 5.571458569377225
```

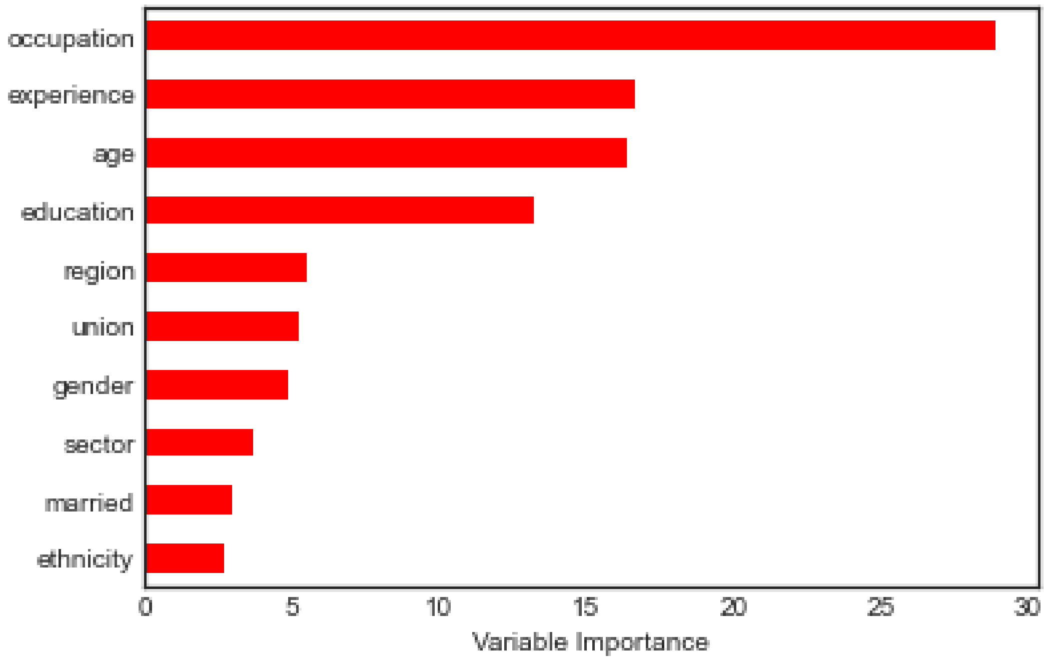
[86]:
```
feature_importance = regr33.feature_importances_*100
rel_imp = pd.Series(feature_importance, index=X.columns).
↪sort_values(inplace=False)
rel_imp.T.plot(kind='barh', color='r', )
plt.xlabel('Variable Importance')
plt.gca().legend_ = None
```



[87]:
```
regr34 = GradientBoostingRegressor(n_estimators=500, learning_rate=0.02,␣
↪max_depth=9, random_state=1)
regr34.fit(X_train, Y1_train)
print("Boosting RMSE =",np.sqrt(mean_squared_error(Y1_test, regr34.
↪predict(X_test))))
```

```
Boosting RMSE = 5.566906492971657
```

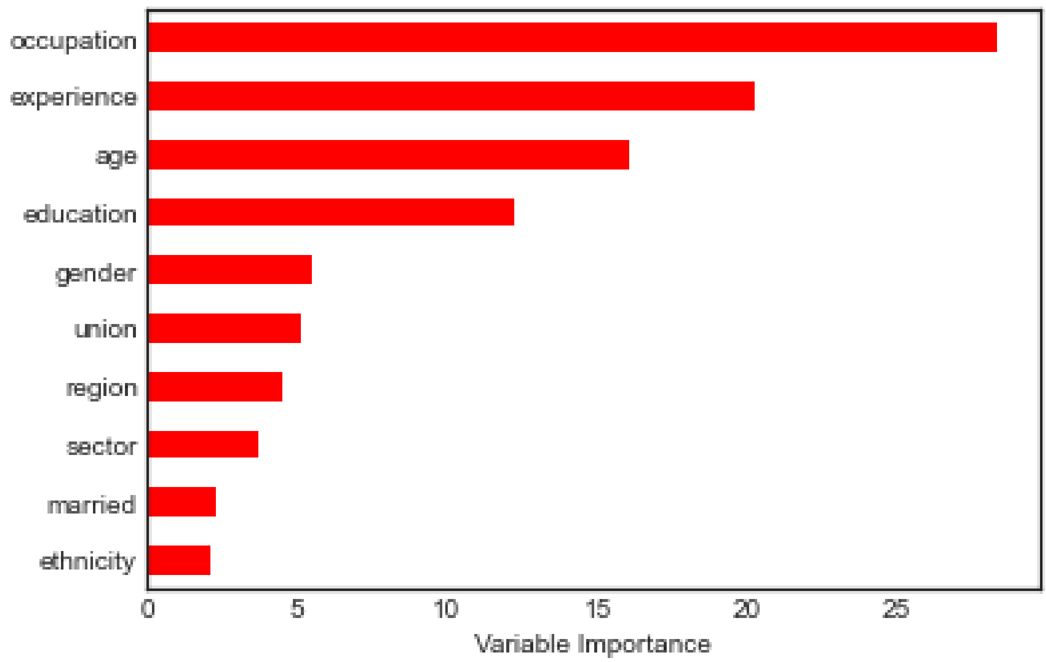
[88]:
```
feature_importance = regr34.feature_importances_*100
rel_imp = pd.Series(feature_importance, index=X.columns).
↪sort_values(inplace=False)
rel_imp.T.plot(kind='barh', color='r', )
plt.xlabel('Variable Importance')
plt.gca().legend_ = None
```



[89]:
```
regr35 = GradientBoostingRegressor(n_estimators=500, learning_rate=0.02,␣
↪max_depth=10, random_state=1)
regr35.fit(X_train, Y1_train)
print("Boosting RMSE =",np.sqrt(mean_squared_error(Y1_test, regr35.
↪predict(X_test))))
```

```
Boosting RMSE = 5.620244208951375
```

[90]:
```
feature_importance = regr35.feature_importances_*100
rel_imp = pd.Series(feature_importance, index=X.columns).
↪sort_values(inplace=False)
rel_imp.T.plot(kind='barh', color='r', )
plt.xlabel('Variable Importance')
plt.gca().legend_ = None
```



Based on the accuracies, RMSEs & feature importance, max_depth = 5 provides a relatively high accuracy, low RMSE, and importance features that are consistent with the previous models.

(RF RMSE = 4.825932756362598)

(Boosting RMSE = 5.266461707713858)

(Bagging RMSE = 4.9780619996926525)

(Regression tree RMSE = 6.056711949391955)

Comparing the performance of the regression models based on RMSEs, Random Forest performs best indicating that our predictions of wages are typically off by about $4.83/hr.

```
[ ]:
```