Cross-Validation with Linear Regression

This notebook demonstrates how to do cross-validation (CV) with linear regression as an example (it is heavily used in almost all modelling techniques such as decision trees, SVM etc.). We will mainly use `sklearn` to do cross-validation.

This notebook is divided into the following parts:

0. Experiments to understand overfitting
1. Building a linear regression model without cross-validation
2. Problems in the current approach
3. Cross-validation: A quick recap
4. Cross-validation in `sklearn` :

    - 4.1 K-fold CV
    - 4.2 Hyperparameter tuning using CV
    - 4.3 Other CV schemes

0. Experiments to Understand Overfitting

In this section, let's quickly go through some experiments to understand what overfitting looks like. We'll run some experiments using polynomial regression.

In [565]:

```
# import all libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import re

import sklearn
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler
from sklearn.preprocessing import PolynomialFeatures
from sklearn.preprocessing import scale
from sklearn.feature_selection import RFE
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import KFold
from sklearn.model_selection import GridSearchCV
from sklearn.pipeline import make_pipeline

import warnings # supress warnings
warnings.filterwarnings('ignore')
```

In [566]:

```
# import Housing.csv
housing = pd.read_csv('Housing.csv')
housing.head()
```

Out[566]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterh
0	13300000	7420	4	2	3	yes	no	no	
1	12250000	8960	4	4	4	yes	no	no	
2	12250000	9960	3	2	2	yes	no	yes	
3	12215000	7500	4	2	2	yes	no	yes	
4	11410000	7420	4	1	2	yes	yes	yes	

In [567]:

```
# number of observations
len(housing.index)
```

Out[567]:

```
545
```

For the first experiment, we'll do regression with only one feature. Let's filter the data so it only contains `area` and `price`.

In [568]:

```
# filter only area and price
df = housing.loc[:, ['area', 'price']]
df.head()
```

Out[568]:

	area	price
0	7420	13300000
1	8960	12250000
2	9960	12250000
3	7500	12215000
4	7420	11410000

In [569]:

```
# recaling the variables (both)
df_columns = df.columns
scaler = MinMaxScaler()
df = scaler.fit_transform(df)

# rename columns (since now its an np array)
df = pd.DataFrame(df)
df.columns = df_columns

df.head()
```

Out[569]:

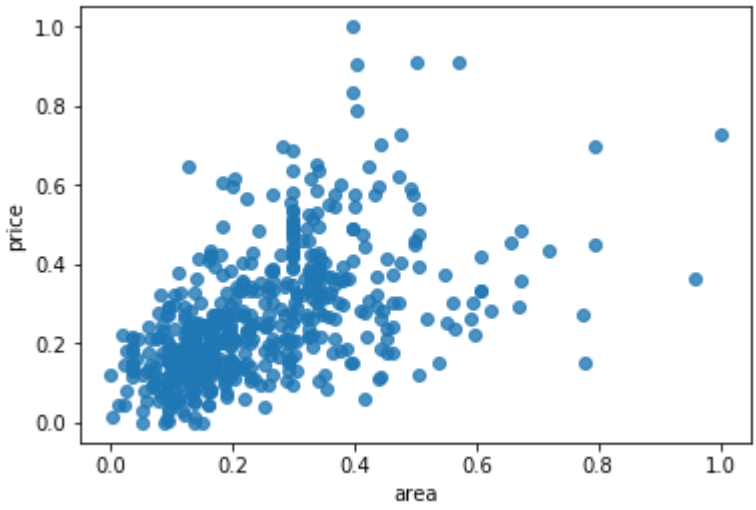
	area	price
0	0.396564	1.000000
1	0.502405	0.909091
2	0.571134	0.909091
3	0.402062	0.906061
4	0.396564	0.836364

In [570]:

```
# visualise area-price relationship
sns.regplot(x="area", y="price", data=df, fit_reg=False)
```

Out[570]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x1c22068978>
```



In [571]:

```
# split into train and test
df_train, df_test = train_test_split(df,
                                     train_size = 0.7,
                                     test_size = 0.3,
                                     random_state = 10)
print(len(df_train))
print(len(df_test))
```

```
381
164
```

In [572]:

```
# split into X and y for both train and test sets
# reshaping is required since sklearn requires the data to be in shape
# (n, 1), not as a series of shape (n, )
X_train = df_train['area']
X_train = X_train.values.reshape(-1, 1)
y_train = df_train['price']

X_test = df_test['area']
X_test = X_test.values.reshape(-1, 1)
y_test = df_test['price']
```

Polynomial Regression

You already know simple linear regression:

$y = \beta_0 + \beta_1 x_1$

In polynomial regression of degree n, we fit a curve of the form:

$y = \beta_0 + \beta_1 x_1 + \beta_2 x_1^2 + \beta_3 x_1^3 \ldots + \beta_n x_1^n$

In the experiment below, we have fitted polynomials of various degrees on the housing data and compared their performance on train and test sets.

In sklearn, polynomial features can be generated using the `PolynomialFeatures` class. Also, to perform `LinearRegression` and `PolynomialFeatures` in tandem, we will use the module `sklearn_pipeline` - it basically creates the features and feeds the output to the model (in that sequence).

In [573]:

```
len(X_train)
```

Out[573]:

```
381
```

Let's now predict the y labels (for both train and test sets) and store the predictions in a table. Each row of the table is one data point, each column is a value of n (degree).

	degree-1	degree-2	degree-3	...	degree-n
x1					

x2

x3

...

xn

In [574]:

```
# fit multiple polynomial features
degrees = [1, 2, 3, 6, 10, 20]

# initialise y_train_pred and y_test_pred matrices to store the train and test predictions
# each row is a data point, each column a prediction using a polynomial of some degree
y_train_pred = np.zeros((len(X_train), len(degrees)))
y_test_pred = np.zeros((len(X_test), len(degrees)))

for i, degree in enumerate(degrees):

    # make pipeline: create features, then feed them to linear_reg model
    model = make_pipeline(PolynomialFeatures(degree), LinearRegression())
    model.fit(X_train, y_train)

    # predict on test and train data
    # store the predictions of each degree in the corresponding column
    y_train_pred[:, i] = model.predict(X_train)
    y_test_pred[:, i] = model.predict(X_test)
```

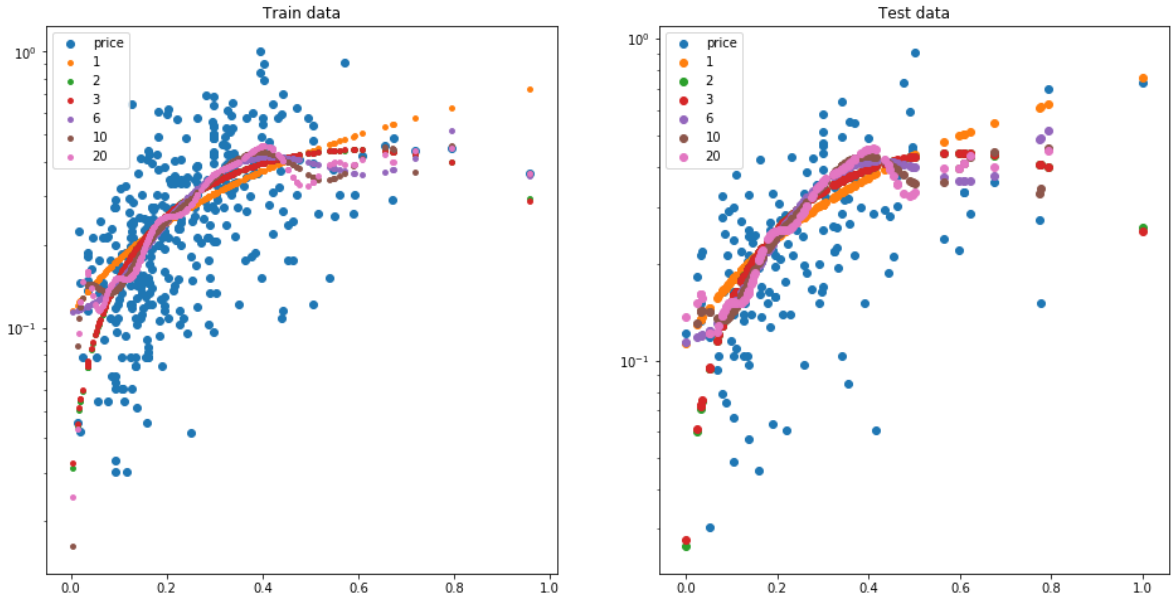
In [575]:

```
# visualise train and test predictions
# note that the y axis is on a log scale

plt.figure(figsize=(16, 8))

# train data
plt.subplot(121)
plt.scatter(X_train, y_train)
plt.yscale('log')
plt.title("Train data")
for i, degree in enumerate(degrees):
    plt.scatter(X_train, y_train_pred[:, i], s=15, label=str(degree))
    plt.legend(loc='upper left')

# test data
plt.subplot(122)
plt.scatter(X_test, y_test)
plt.yscale('log')
plt.title("Test data")
for i, degree in enumerate(degrees):
    plt.scatter(X_test, y_test_pred[:, i], label=str(degree))
    plt.legend(loc='upper left')
```



In [576]:

```
# compare r2 for train and test sets (for all polynomial fits)
print("R-squared values: \n")

for i, degree in enumerate(degrees):
    train_r2 = round(sklearn.metrics.r2_score(y_train, y_train_pred[:, i]), 2)
    test_r2 = round(sklearn.metrics.r2_score(y_test, y_test_pred[:, i]), 2)
    print("Polynomial degree {0}: train score={1}, test score={2}".format(degree,
                                                                          train_r2,
                                                                          test_r2))
```

```
R-squared values:

Polynomial degree 1: train score=0.29, test score=0.25
Polynomial degree 2: train score=0.34, test score=0.22
Polynomial degree 3: train score=0.34, test score=0.22
Polynomial degree 6: train score=0.36, test score=0.11
Polynomial degree 10: train score=0.37, test score=-108.76
Polynomial degree 20: train score=0.38, test score=-299928623752.57
```

1. Building a Model Without Cross-Validation

Let's now build a multiple regression model. First, let's build a vanilla MLR model without any cross-validation etc.

In [577]:

```
# data preparation

# list of all the "yes-no" binary categorical variables
# we'll map yes to 1 and no to 0
binary_vars_list =  ['mainroad', 'guestroom', 'basement', 'hotwaterheating', 'airconditioni

# defining the map function
def binary_map(x):
    return x.map({'yes': 1, "no": 0})

# applying the function to the housing variables list
housing[binary_vars_list] = housing[binary_vars_list].apply(binary_map)
housing.head()
```

Out[577]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterh
0	13300000	7420	4	2	3	1	0	0	
1	12250000	8960	4	4	4	1	0	0	
2	12250000	9960	3	2	2	1	0	1	
3	12215000	7500	4	2	2	1	0	1	
4	11410000	7420	4	1	2	1	1	1	

In [578]:

```
# 'dummy' variables
# get dummy variables for 'furnishingstatus'
# also, drop the first column of the resulting df (since n-1 dummy vars suffice)
status = pd.get_dummies(housing['furnishingstatus'], drop_first = True)
status.head()
```

Out[578]:

	semi-furnished	unfurnished
0	0	0
1	0	0
2	1	0
3	0	0
4	0	0

In [579]:

```
# concat the dummy variable df with the main df
housing = pd.concat([housing, status], axis = 1)
housing.head()
```

Out[579]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterh
0	13300000	7420	4	2	3	1	0	0	
1	12250000	8960	4	4	4	1	0	0	
2	12250000	9960	3	2	2	1	0	1	
3	12215000	7500	4	2	2	1	0	1	
4	11410000	7420	4	1	2	1	1	1	

In [580]:

```
# 'furnishingstatus' since we alreday have the dummy vars
housing.drop(['furnishingstatus'], axis = 1, inplace = True)
housing.head()
```

Out[580]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hotwaterh
0	13300000	7420	4	2	3	1	0	0	
1	12250000	8960	4	4	4	1	0	0	
2	12250000	9960	3	2	2	1	0	1	
3	12215000	7500	4	2	2	1	0	1	
4	11410000	7420	4	1	2	1	1	1	

Splitting Into Train and Test

In [581]:

```
# train-test 70-30 split
df_train, df_test = train_test_split(housing,
                                     train_size = 0.7,
                                     test_size = 0.3,
                                     random_state = 100)

# rescale the features
scaler = MinMaxScaler()

# apply scaler() to all the numeric columns
numeric_vars = ['area', 'bedrooms', 'bathrooms', 'stories', 'parking','price']
df_train[numeric_vars] = scaler.fit_transform(df_train[numeric_vars])
df_train.head()
```

Out[581]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hc
359	0.169697	0.155227	0.4	0.0	0.000000	1	0	0	
19	0.615152	0.403379	0.4	0.5	0.333333	1	0	0	
159	0.321212	0.115628	0.4	0.5	0.000000	1	1	1	
35	0.548133	0.454417	0.4	0.5	1.000000	1	0	0	
28	0.575758	0.538015	0.8	0.5	0.333333	1	0	1	

In [582]:

```
# apply rescaling to the test set also
df_test[numeric_vars] = scaler.fit_transform(df_test[numeric_vars])
df_test.head()
```

Out[582]:

	price	area	bedrooms	bathrooms	stories	mainroad	guestroom	basement	hc
265	0.247651	0.084536	0.333333	0.000000	0.333333	1	0	0	
54	0.530201	0.298969	0.333333	0.333333	0.333333	1	1	0	
171	0.328859	0.592371	0.333333	0.000000	0.000000	1	0	0	
244	0.261745	0.252234	0.333333	0.000000	0.333333	1	1	1	
268	0.245638	0.226804	0.666667	0.000000	0.333333	1	0	0	

In [583]:

```
# divide into X_train, y_train, X_test, y_test
y_train = df_train.pop('price')
X_train = df_train

y_test = df_test.pop('price')
X_test = df_test
```

Note that we haven't rescaled the test set yet, which we'll need to do later while making predictions.

Using RFE

Now, we have 13 predictor features. To build the model using RFE, we need to tell RFE how many features we want in the final model. It then runs a feature elimination algorithm.

Note that the number of features to be used in the model is a **hyperparameter**.

In [584]:

```
# num of max features
len(X_train.columns)
```

Out[584]:

```
13
```

In [585]:

```
# first model with an arbitrary choice of n_features
# running RFE with number of features=10

lm = LinearRegression()
lm.fit(X_train, y_train)

rfe = RFE(lm, n_features_to_select=10)
rfe = rfe.fit(X_train, y_train)
```

In [586]:

```
# tuples of (feature name, whether selected, ranking)
# note that the 'rank' is > 1 for non-selected features
list(zip(X_train.columns,rfe.support_,rfe.ranking_))
```

Out[586]:

```
[('area', True, 1),
 ('bedrooms', True, 1),
 ('bathrooms', True, 1),
 ('stories', True, 1),
 ('mainroad', True, 1),
 ('guestroom', True, 1),
 ('basement', False, 3),
 ('hotwaterheating', True, 1),
 ('airconditioning', True, 1),
 ('parking', True, 1),
 ('prefarea', True, 1),
 ('semi-furnished', False, 4),
 ('unfurnished', False, 2)]
```

In [587]:

```
# predict prices of X_test
y_pred = rfe.predict(X_test)

# evaluate the model on test set
r2 = sklearn.metrics.r2_score(y_test, y_pred)
print(r2)
```

```
0.5812051458999572
```

In [588]:

```
# try with another value of RFE
lm = LinearRegression()
lm.fit(X_train, y_train)

rfe = RFE(lm, n_features_to_select=6)
rfe = rfe.fit(X_train, y_train)

# predict prices of X_test
y_pred = rfe.predict(X_test)
r2 = sklearn.metrics.r2_score(y_test, y_pred)
print(r2)
```

```
0.5350445027578821
```

2. Problems in the Current Approach

In train-test split, we have three options:

1. **Simply split into train and test**: But that way tuning a hyperparameter makes the model 'see' the test data (i.e. knowledge of test data leaks into the model)
2. **Split into train, validation, test sets**: Then the validation data would eat into the training set
3. **Cross-validation**: Split into train and test, and train multiple models by sampling the train set. Finally, just test once on the test set.

3. Cross-Validation: A Quick Recap

The following figure illustrates k-fold cross-validation with k=4. There are some other schemes to divide the training set, we'll look at them briefly later.

4. Cross-Validation in sklearn

Let's now experiment with k-fold CV.

4.1 K-Fold CV

In [589]:

```
# k-fold CV (using all the 13 variables)
lm = LinearRegression()
scores = cross_val_score(lm, X_train, y_train, scoring='r2', cv=5)
scores
```

Out[589]:

```
array([0.6829775 , 0.69324306, 0.6762109 , 0.61782891, 0.59266171])
```

In [590]:

```
# the other way of doing the same thing (more explicit)

# create a KFold object with 5 splits
folds = KFold(n_splits = 5, shuffle = True, random_state = 100)
scores = cross_val_score(lm, X_train, y_train, scoring='r2', cv=folds)
scores
```

Out[590]:

```
array([0.59930574, 0.71307628, 0.61325733, 0.62739077, 0.6212937 ])
```

In [591]:

```
# can tune other metrics, such as MSE
scores = cross_val_score(lm, X_train, y_train, scoring='mean_squared_error', cv=5)
scores
```

Out[591]:

```
array([-0.00806336, -0.00658776, -0.0064797 , -0.0070743 , -0.01523682])
```

4.2 Hyperparameter Tuning Using Grid Search Cross-Validation

A common use of cross-validation is for tuning hyperparameters of a model. The most common technique is what is called **grid search** cross-validation.

In [592]:

```
# number of features in X_train
len(X_train.columns)
```

Out[592]:

```
13
```

In [593]:

```
# step-1: create a cross-validation scheme
folds = KFold(n_splits = 5, shuffle = True, random_state = 100)

# step-2: specify range of hyperparameters to tune
hyper_params = [{'n_features_to_select': list(range(1, 14))}]


# step-3: perform grid search
# 3.1 specify model
lm = LinearRegression()
lm.fit(X_train, y_train)
rfe = RFE(lm)

# 3.2 call GridSearchCV()
model_cv = GridSearchCV(estimator = rfe,
                        param_grid = hyper_params,
                        scoring= 'r2',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)

# fit the model
model_cv.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 13 candidates, totalling 65 fits

[Parallel(n_jobs=1)]: Done  65 out of  65 | elapsed:    0.6s finished
```

Out[593]:

```
GridSearchCV(cv=KFold(n_splits=5, random_state=100, shuffle=True),
       error_score='raise',
       estimator=RFE(estimator=LinearRegression(copy_X=True, fit_intercept=T
rue, n_jobs=1, normalize=False),
  n_features_to_select=None, step=1, verbose=0),
       fit_params=None, iid=True, n_jobs=1,
       param_grid=[{'n_features_to_select': [1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13]}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring='r2', verbose=1)
```

In [594]:

```
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[594]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_n_features_to_select
0	0.021255	0.004623	0.000754	0.000177	1
1	0.011884	0.002191	0.000645	0.000187	2
2	0.010464	0.001464	0.000561	0.000027	3
3	0.009211	0.000434	0.000540	0.000012	4
4	0.008505	0.000537	0.000547	0.000034	5
5	0.007733	0.000240	0.000545	0.000022	6
6	0.006756	0.000089	0.000537	0.000007	7
7	0.006009	0.000059	0.000532	0.000009	8
8	0.005458	0.000290	0.000535	0.000011	9
9	0.004963	0.000692	0.000544	0.000022	10
10	0.004004	0.000257	0.000592	0.000131	11
11	0.003051	0.000100	0.000520	0.000006	12
12	0.002276	0.000162	0.000509	0.000005	13

13 rows × 21 columns

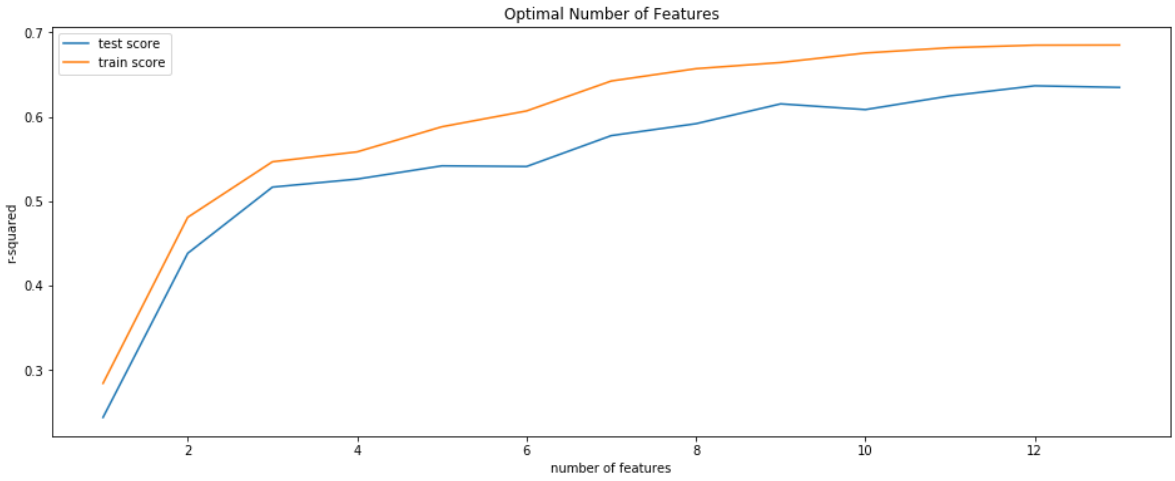
In [595]:

```
# plotting cv results
plt.figure(figsize=(16,6))

plt.plot(cv_results["param_n_features_to_select"], cv_results["mean_test_score"])
plt.plot(cv_results["param_n_features_to_select"], cv_results["mean_train_score"])
plt.xlabel('number of features')
plt.ylabel('r-squared')
plt.title("Optimal Number of Features")
plt.legend(['test score', 'train score'], loc='upper left')
```

Out[595]:

```
<matplotlib.legend.Legend at 0x1c2417a978>
```



Now we can choose the optimal value of number of features and build a final model.

In [596]:

```
# final model
n_features_optimal = 10

lm = LinearRegression()
lm.fit(X_train, y_train)

rfe = RFE(lm, n_features_to_select=n_features_optimal)
rfe = rfe.fit(X_train, y_train)

# predict prices of X_test
y_pred = lm.predict(X_test)
r2 = sklearn.metrics.r2_score(y_test, y_pred)
print(r2)
```

```
0.599557533872853
```

Notice that the test score is very close to the 'mean test score' on the k-folds (about 60%). In general, the mean score estimated by CV will usually be a good estimate of the test score.

Another Example: Car Price Prediction

In [597]:

```
# reading the dataset
cars = pd.read_csv("CarPrice_Assignment.csv")
```

In [598]:

```
# All data preparation steps in this cell

# converting symboling to categorical
cars['symboling'] = cars['symboling'].astype('object')


# create new column: car_company
p = re.compile(r'\w+-?\w+')
cars['car_company'] = cars['CarName'].apply(lambda x: re.findall(p, x)[0])


# replacing misspelled car_company names
# volkswagen
cars.loc[(cars['car_company'] == "vw") |
         (cars['car_company'] == "vokswagen")
         , 'car_company'] = 'volkswagen'
# porsche
cars.loc[cars['car_company'] == "porcshce", 'car_company'] = 'porsche'
# toyota
cars.loc[cars['car_company'] == "toyouta", 'car_company'] = 'toyota'
# nissan
cars.loc[cars['car_company'] == "Nissan", 'car_company'] = 'nissan'
# mazda
cars.loc[cars['car_company'] == "maxda", 'car_company'] = 'mazda'


# drop carname variable
cars = cars.drop('CarName', axis=1)


# split into X and y
X = cars.loc[:, ['symboling', 'fueltype', 'aspiration', 'doornumber',
       'carbody', 'drivewheel', 'enginelocation', 'wheelbase', 'carlength',
       'carwidth', 'carheight', 'curbweight', 'enginetype', 'cylindernumber',
       'enginesize', 'fuelsystem', 'boreratio', 'stroke', 'compressionratio',
       'horsepower', 'peakrpm', 'citympg', 'highwaympg',
       'car_company']]
y = cars['price']


# creating dummy variables for categorical variables
cars_categorical = X.select_dtypes(include=['object'])
cars_categorical.head()


# convert into dummies
cars_dummies = pd.get_dummies(cars_categorical, drop_first=True)
cars_dummies.head()


# drop categorical variables
X = X.drop(list(cars_categorical.columns), axis=1)


# concat dummy variables with X
X = pd.concat([X, cars_dummies], axis=1)


# rescale the features
```

```
cols = X.columns
X = pd.DataFrame(scale(X))
X.columns = cols


# split into train and test
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                                    train_size=0.7,
                                                    test_size = 0.3, random_state=40)
```

In [599]:

```
# number of features
len(X_train.columns)
```

Out[599]:

```
68
```

In [600]:

```
# creating a KFold object with 5 splits
folds = KFold(n_splits = 5, shuffle = True, random_state = 100)

# specify range of hyperparameters
hyper_params = [{'n_features_to_select': list(range(2, 40))}]

# specify model
lm = LinearRegression()
lm.fit(X_train, y_train)
rfe = RFE(lm)

# set up GridSearchCV()
model_cv = GridSearchCV(estimator = rfe,
                        param_grid = hyper_params,
                        scoring= 'r2',
                        cv = folds,
                        verbose = 1,
                        return_train_score=True)

# fit the model
model_cv.fit(X_train, y_train)
```

```
Fitting 5 folds for each of 38 candidates, totalling 190 fits

[Parallel(n_jobs=1)]: Done 190 out of 190 | elapsed:   12.4s finished
```

Out[600]:

```
GridSearchCV(cv=KFold(n_splits=5, random_state=100, shuffle=True),
       error_score='raise',
       estimator=RFE(estimator=LinearRegression(copy_X=True, fit_intercept=T
rue, n_jobs=1, normalize=False),
  n_features_to_select=None, step=1, verbose=0),
       fit_params=None, iid=True, n_jobs=1,
       param_grid=[{'n_features_to_select': [2, 3, 4, 5, 6, 7, 8, 9, 10, 11,
12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
31, 32, 33, 34, 35, 36, 37, 38, 39]}],
       pre_dispatch='2*n_jobs', refit=True, return_train_score=True,
       scoring='r2', verbose=1)
```

In [601]:

```
# cv results
cv_results = pd.DataFrame(model_cv.cv_results_)
cv_results
```

Out[601]:

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_n_features_to_select
0	0.080977	0.014933	0.000463	1.605575e-05	2
1	0.072639	0.000502	0.000510	1.230776e-04	3
2	0.075410	0.004731	0.000449	1.406468e-06	4
3	0.072320	0.000901	0.000450	1.512408e-06	5
4	0.070245	0.000234	0.000481	6.004450e-05	6
5	0.073988	0.004944	0.000533	9.688582e-05	7
6	0.071237	0.002765	0.000455	4.665700e-06	8
7	0.068408	0.000527	0.000455	4.505039e-06	9
8	0.071769	0.003848	0.000458	8.098655e-06	10
9	0.068091	0.002635	0.000458	6.324881e-06	11
10	0.065974	0.000170	0.000462	1.365754e-05	12
11	0.071652	0.011210	0.000466	1.353999e-05	13
12	0.065336	0.000592	0.000460	5.230884e-06	14
13	0.066241	0.004866	0.000483	4.721194e-05	15
14	0.064967	0.001730	0.000460	1.549537e-06	16
15	0.068873	0.004905	0.000462	3.749772e-06	17
16	0.069526	0.004974	0.000518	7.734713e-05	18
17	0.060862	0.000792	0.000461	2.301696e-06	19
18	0.059948	0.000405	0.000467	4.291534e-06	20
19	0.059856	0.001099	0.000529	1.302623e-04	21
20	0.060118	0.002698	0.000524	1.172173e-04	22

	mean_fit_time	std_fit_time	mean_score_time	std_score_time	param_n_features_to_select	
21	0.057417	0.000536	0.000459	9.956652e-07	23	{
22	0.056351	0.000323	0.000463	4.341575e-06	24	{
23	0.059238	0.003907	0.000523	1.207523e-04	25	{
24	0.055106	0.001007	0.000462	1.741598e-06	26	{
25	0.056810	0.002534	0.000524	7.280936e-05	27	{
26	0.089414	0.016823	0.000632	1.292317e-04	28	{
27	0.056095	0.003312	0.000469	5.823273e-06	29	{
28	0.069551	0.012552	0.000668	1.074558e-04	30	{
29	0.072766	0.024778	0.000593	1.596130e-04	31	{
30	0.049456	0.000566	0.000582	2.108430e-04	32	{
31	0.064888	0.009813	0.000585	1.793140e-04	33	{
32	0.050363	0.002295	0.000485	1.368448e-05	34	{
33	0.046479	0.000808	0.000470	1.886974e-06	35	{
34	0.045449	0.000894	0.000470	4.022426e-06	36	{
35	0.044488	0.000677	0.000477	8.201599e-06	37	{
36	0.052455	0.006477	0.000611	2.521245e-04	38	{
37	0.048690	0.001671	0.000575	9.897910e-05	39	{

38 rows × 21 columns

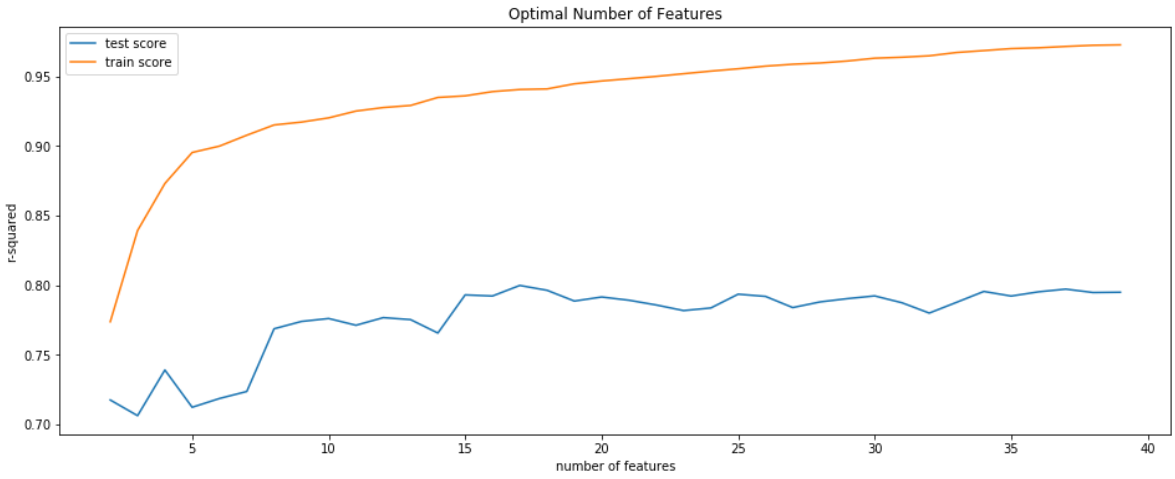
In [602]:

```
# plotting cv results
plt.figure(figsize=(16,6))

plt.plot(cv_results["param_n_features_to_select"], cv_results["mean_test_score"])
plt.plot(cv_results["param_n_features_to_select"], cv_results["mean_train_score"])
plt.xlabel('number of features')
plt.ylabel('r-squared')
plt.title("Optimal Number of Features")
plt.legend(['test score', 'train score'], loc='upper left')
```

Out[602]:

```
<matplotlib.legend.Legend at 0x1c23886048>
```



4.3 Types of Cross-Validation Schemes

1. **K-Fold** cross-validation: Most common
2. **Leave One Out (LOO)**: Takes each data point as the 'test sample' once, and trains the model on the rest n-1 data points. Thus, it trains n total models.
    - Advantage: Utilises the data well since each model is trained on n-1 samples
    - Disadvantage: Computationally expensive
3. **Leave P-Out (LPO)**: Creat all possible splits after leaving p samples out. For n data points, there are (nCp) possibile train-test splits.
4. (**For classification problems**) **Stratified K-Fold**: Ensures that the relative class proportion is approximately preserved in each train and validation fold. Important when ther eis huge class imbalance (e.g. 98% good customers, 2% bad).

Additional Reading

The sklearn documentation enlists all CV schemes here. (http://scikit-learn.org/stable/modules/cross_validation.html)