Assignment 3

Due: Midnight on Tuesday October 8, 2019

NetID: sa857

In this assignment you will gain some experience with decision trees and random forests using two data sets. One is a diabetes data set, where the task is to predict the progression of the disease. The other a data set of real estate listings, where the task is to forecast the sale price of the house.

Submission Instructions:

Please fill out this *starter* Jupyter Notebook, and submit **both** this `.ipynb` file as well as a pdf file (via html).

- In the notebook interface, choose `File -> Download as -> Notebook (ipynb)`.
- In the notebook interface, choose `File -> Download as -> HTML`. Then open the html file, and print to pdf.

Notes:

- We are using the markdown cell-type for texts (and latex), and the code cell-type for the python code. Make sure you don't mix these up. You can change the type from the dropdown at the toolbar on the top.

In [4]:
```
import numpy as np
import matplotlib.pyplot as plt
import ipdb
import pprint
import pandas as pd
import seaborn as sns
```

Question 1: Regression trees vs. random forests (20 pts)

This problem is based on the `diabetes` dataset from the `sklearn` package. Please read about the dataset at https://scikit-learn.org/stable/datasets/index.html#diabetes-dataset (https://scikit-learn.org/stable/datasets/index.html#diabetes-dataset). We will seek to predict the response, which is a quantitative measure of diabetes progression one year after baseline, using regression trees and random forests.

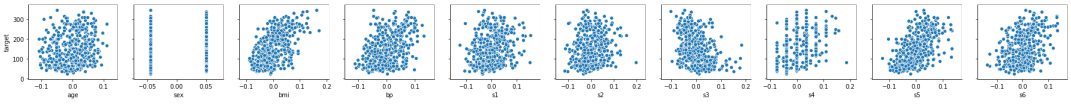
The following cell imports the dataset as `diabetes` and names the predictor variables `diabetes_x` and the response `diabetes_y`. The names of the six predictor variables are also printed. For a more detailed description, use the `.DESCR` aspect of `diabetes`.

In [5]:
```
from sklearn import datasets
diabetes = datasets.load_diabetes()
diabetes_x = diabetes.data
diabetes_y = diabetes.target
print(diabetes.feature_names)
```

```
['age', 'sex', 'bmi', 'bp', 's1', 's2', 's3', 's4', 's5', 's6']
```

In [6]:
```
df = pd.DataFrame(diabetes_x, columns=diabetes.feature_names)
df['target'] = diabetes_y
sns.pairplot(df, y_vars=["target"], x_vars=diabetes.feature_name
s)
```

Out[6]: `<seaborn.axisgrid.PairGrid at 0x11a5ca5d0>`



Part (a): Building a Simple Regression Tree

To start we will manually build a regression tree using only two of the predictor variables: `bmi` and `s5`. To keep things simple, build a tree that has exactly three nodes and four leaves. (i.e. the data is split into two parts initially and then each of those parts is again split one more time.) At each node you will need to evaluate each possible splitting point for both `bmi` and `s5` and pick the one that minimizes the RSS.

When you have built the regression tree, create a scatter plot of `s5` versus `bmi`, color-coded by the response variable. In this plot, use vertical and horizontal lines to indicate the regions that your tree splits the data into. You may find the functions `plt.hlines()` and `plt.vlines()` to be useful.

In [7]:
```
bmi = diabetes_x[:,2] # all rows in the bmi column
s5 = diabetes_x[:,8]  # all rows in the s5  column
```

In [8]:
```
# we put a wrapper on the np.mean function to avoid warnings fro
m taking the average of an empty list
def average(x):
    if len(x) == 0:
        return(0.0)
    else:
        return(np.mean(x))
```

In [9]:
```
def splitter(bmi_dat, s5_dat, labels_dat):

    rss_bmi = []
    rss_s5 = []


    for i in range(len(bmi_dat)): # go through all the rows

        # for bmi (body mass index)
        left = np.where(bmi_dat <= bmi_dat[i])[0] # all entries' indices such that entries <= current val
        right = np.where(bmi_dat > bmi_dat[i])[0] # all entries' indices such that entries > current val

        # actually the TSS: actual value - average
        # how good is this partition?
        # this is done by using the indices from the previous steps
        rss_bmi.append(np.sum((labels_dat[left] - average(labels_dat[left]))**2) +
                       np.sum((labels_dat[right] - average(labels_dat[right]))**2))

        # do the same for s5
        left = np.where(s5_dat <= s5_dat[i])[0]
        right = np.where(s5_dat > s5_dat[i])[0]

        # how good is this partition?
        rss_s5.append(np.sum((labels_dat[left] - average(labels_dat[left]))**2) +
                      np.sum((labels_dat[right] - average(labels_dat[right]))**2))

    # return the indices of rss arrays which produced the best splits (lowest RSS)
    return {
        "best_bmi_cut" : np.argmin(rss_bmi),
        "best_s5_cut" : np.argmin(rss_s5),
        "best_bmi_cut_rss" : rss_bmi[np.argmin(rss_bmi)],
        "best_s5_cut_rss" : rss_s5[np.argmin(rss_s5)]
    }
```

In [10]:
```
def createNewDataFromSplit(bmi_dat, s5_dat, labels_dat, split_dat):

    # figure out best cut and indices between the two predictors
    if split_dat["best_s5_cut_rss"] < split_dat["best_bmi_cut_rss"]:
        best_overall_cut = split_dat["best_s5_cut"]
        best_overall_cut_location = s5_dat[best_overall_cut]
        best_overall_cut_type = "s5"
        which_indices_left = np.where(s5_dat <= s5_dat[best_overall_cut])
        which_indices_right = np.where(s5_dat > s5_dat[best_overall_cut])
    else:
        best_overall_cut = split_dat["best_bmi_cut"]
        best_overall_cut_location = bmi_dat[best_overall_cut]
        best_overall_cut_type = "bmi"
        which_indices_left = np.where(bmi_dat <= bmi_dat[best_overall_cut])
        which_indices_right = np.where(bmi_dat > bmi_dat[best_overall_cut])

    # left partition
    s5_left = s5_dat[which_indices_left]
    bmi_left = bmi_dat[which_indices_left]
    labels_left = labels_dat[which_indices_left]

    # right partition
    s5_right = s5_dat[which_indices_right]
    bmi_right = bmi_dat[which_indices_right]
    labels_right = labels_dat[which_indices_right]

    # tests
    assert(len(s5_left) + len(s5_right) == len(s5_dat))
    assert(len(bmi_left) + len(bmi_right) == len(bmi_dat))
    assert(len(labels_left) + len(labels_right) == len(labels_dat))
    assert(len(s5_left) == len(bmi_left))
    assert(len(s5_right) == len(bmi_right))

    # return all data
    return {
        "s5_left" : s5_left,
        "bmi_left" : bmi_left,
        "labels_left" : labels_left,

        "s5_left_len" : len(s5_left),
        "bmi_left_len" : len(bmi_left),
        "labels_left_len" : len(labels_left),

        "s5_right" : s5_right,
        "bmi_right" : bmi_right,
        "labels_right" : labels_right,
```

In [11]:
```
def splitPipeline(bmi_dat, s5_dat, labels_dat):
    split_results = splitter(bmi_dat, s5_dat, labels_dat)
    new_dat = createNewDataFromSplit(bmi_dat, s5_dat, labels_da
t, split_results)
    return new_dat
```

In [12]:
```
# first split
first_split_dat = splitPipeline(bmi, s5, diabetes_y)
```

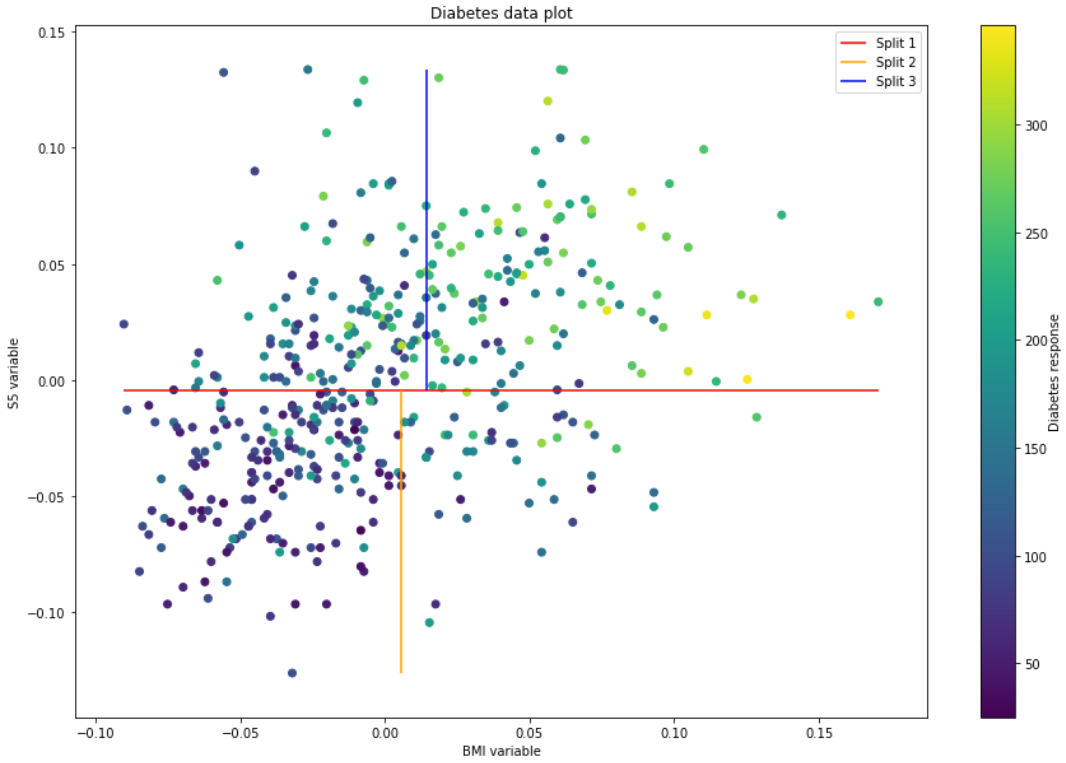
In [13]:
```
# left split
left_split_dat = splitPipeline(first_split_dat["bmi_left"],
                               first_split_dat["s5_left"],
                               first_split_dat["labels_left"])

left_leaf_one = {
    "bmi": left_split_dat["bmi_left"],
    "s5": left_split_dat["s5_left"],
    "labels" : left_split_dat["labels_left"]
}

left_leaf_two = {
    "bmi": left_split_dat["bmi_right"],
    "s5": left_split_dat["s5_right"],
    "labels" : left_split_dat["labels_right"]
}
```

In [14]:
```
# right split
right_split_dat = splitPipeline(first_split_dat["bmi_right"],
                               first_split_dat["s5_right"],
                               first_split_dat["labels_right"])

right_leaf_one = {
    "s5": right_split_dat["s5_left"],
    "bmi": right_split_dat["bmi_left"],
    "labels": right_split_dat["labels_left"]
}

right_leaf_two = {
    "s5": right_split_dat["s5_right"],
    "bmi": right_split_dat["bmi_right"],
    "labels" :right_split_dat["labels_right"]
}
```

In [15]:
```
# You should feel free to rewrite the above code in any way that
suits you.
# Now complete the code to make the best split, and then split e
ach child node,
# and then visualize the tree, showing the four leaf rectangles
in the space s5 vs. bmi
```

In [16]:
```
# draw the main figure
plt.figure(figsize=(15,10))
plt.scatter(bmi, s5, c=diabetes_y)
plt.title("Diabetes data plot")
plt.xlabel("BMI variable")
plt.ylabel("S5 variable")
cbar = plt.colorbar()
cbar.set_label("Diabetes response", labelpad=+1)

# draw the cuts

# the first one was an s5 cut at -0.00421985970694603
plt.hlines(y=first_split_dat["best_overall_cut_location"],
           xmin=min(bmi),
           xmax=max(bmi),
           label="Split 1",
           color="red")

# the second one was a bmi cut at 0.00564997867688165
plt.vlines(left_split_dat["best_overall_cut_location"],
           min(s5),
           first_split_dat["best_overall_cut_location"],
           label="Split 2",
           color="orange")

# the third one was a bmi cut at 0.0142724752679289
plt.vlines(right_split_dat["best_overall_cut_location"],
           first_split_dat["best_overall_cut_location"],
           max(s5),
           label="Split 3",
           color="blue")

# add legend
plt.legend()
```

Out[16]: <matplotlib.legend.Legend at 0x11ce26b90>



Part (b) Fitting a Full Regression Tree

Now build a tree that uses all the predictor variables, has a more flexible structure, and is validated with a test set. Split the full dataset into a training set and a test set (50/50). Fit a regression tree to the training set using the function `DecisionTreeRegressor` from `sklearn.tree` . For now, use your best judgment to choose parameters for tree complexity; we will use analytical methods to choose parameters in later parts of this problem set. Some starter code is provided:

In [17]:
```
from sklearn import tree
from sklearn.model_selection import train_test_split
```

In [18]:
```
# split into training and test
X_train, X_test, y_train, y_test = train_test_split(diabetes_x,
                                                    diabetes_y,
                                                    test_size=0.
5,
                                                    random_stat
e=0)
```

In [19]:
```
#tree parameters go inside the first set of parentheses
regr = tree.DecisionTreeRegressor(random_state=0)

# the training data goes in the second set of parentheses
regr.fit(X_train, y_train)
```

Out[19]:
```
DecisionTreeRegressor(criterion='mse', max_depth=None, max_feat
ures=None,
                      max_leaf_nodes=None, min_impurity_decreas
e=0.0,
                      min_impurity_split=None, min_samples_lea
f=1,
                      min_samples_split=2, min_weight_fraction_
leaf=0.0,
                      presort=False, random_state=0, splitter='
best')
```

Part (c) Plotting the Tree

Plot your regression tree. To do so, we suggest that you use GraphViz in conjunction with `sklearn.tree.export_graphviz` . Once you install GraphViz, the following cell will plot the tree.

Instructions for using GraphViz (Windows):

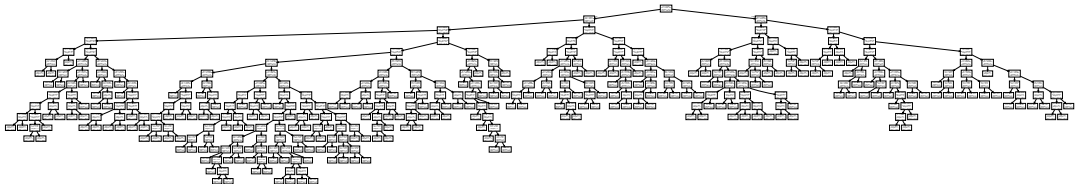
1. Install GraphViz to your computer from the link https://graphviz.gitlab.io/download/ (https://graphviz.gitlab.io/download/).
2. Install the Python package using `pip install graphviz` or `conda install graphviz` .
3. Set a path to your computer's GraphViz installation (NOT the Python package). You can do so locally in this notebook by running something like `import os; os.environ["PATH"] += os.pathsep + 'C:/Program Files (x86)/Graphviz2.38/bin/'` (substituting in the location of your own GraphViz installation).
4. You can now use the functions in the `graphviz` package with `sklearn.tree.export_graphviz` !

Instructions for using GraphViz (Mac OS):

1. Make sure you have the package manager Homebrew.
2. Install GraphViz to your computer using `brew install graphviz` .
3. Install the Python package using `pip install graphviz` or `conda install graphviz` .
4. You can now use the functions in the `graphviz` package with `sklearn.tree.export_graphviz` ! *Note: If you get an ExecutableNotFound error, you might have to set a path to your computer's GraphViz installation (NOT the Python package). You can do so locally in this notebook by running something like `import os; os.environ["PATH"] += os.pathsep + 'C:/Program Files (x86)/Graphviz2.38/bin/'` (substituting in the location of your own GraphViz installation).*

In [20]:
```
import graphviz
#import os; os.environ["PATH"] += os.pathsep + 'C:/Program Files (x86)/Graphviz2.38/bin/'

graphviz.Source(tree.export_graphviz(regr, out_file=None, feature_names=diabetes.feature_names))
```

Out[20]:



Part (d) Evaluation

Interpret your regression tree. What are some examples of variables that seem to correspond with higher or lower measures of diabetes progression? Find the MSE of the model using the test set. The `.predict` method for your model can help with this.

In [21]:
```
pprint.pprint(sorted(list(zip(list(regr.feature_importances_), diabetes.feature_names))))
```

```
[(0.013413759400397468, 'sex'),
 (0.026684977242390943, 's4'),
 (0.0408694277407698, 's1'),
 (0.04642432653642173, 'age'),
 (0.053840420384133045, 's6'),
 (0.05405326763781456, 's3'),
 (0.060981196265229215, 's2'),
 (0.13013451791509534, 'bmi'),
 (0.14893407472936745, 'bp'),
 (0.4246640321483804, 's5')]
```

It appears that the variables that correspond to higher measures of diabetes progression are the ones that most reduce the MSE from one level to the next. This reduction is given by the `.feature_imporances_` property. Based on this we can see that `s5`, `bp` and `bmi` are some of the more important features, whereas `sex`, `s4` and `s1` are less important.

In [22]:
```
from sklearn.metrics import mean_squared_error
y_pred = regr.predict(X_test)
mean_squared_error(y_test, y_pred)
```

Out[22]: 6874.162895927602

Part (e) Random Forest

Now use random forests to analyze the data with the `RandomForestRegressor` function from `sklearn.ensemble`. (Again, you may use your best judgment to choose the initial parameters for tree complexity.)

(i) What test MSE do you obtain, and how does it compare to the test MSE of the regression tree above?

(ii) According to the model, which variables are most important in predicting diabetes progression? (The `.feature_importances_` method of the model may help with this.)

(iii) Plot the MSE of the prediction against m, the number of variables considered at each split.

(iv) Comment on the plot you created and if it makes sense.

In [23]:
```
from sklearn import ensemble
```

part i) the default configuration

In [24]:
```
# dtr = ensemble.RandomForestRegressor(min_samples_leaf = 15, ma
x_features = m)
dtr = ensemble.RandomForestRegressor(random_state=0, min_samples
_leaf = 15)
regr = dtr.fit(X_train, y_train)
y_pred = regr.predict(X_test)
mean_squared_error(y_test, y_pred)
```

```
/Users/sarimabbas/Developer/dataScience/sds355_container/sds355
_env/lib/python3.7/site-packages/sklearn/ensemble/forest.py:24
5: FutureWarning: The default value of n_estimators will change
from 10 in version 0.20 to 100 in 0.22.
  "10 in version 0.20 to 100 in 0.22.", FutureWarning)
```

Out[24]:
```
3663.9641166378487
```

The MSE is lower (from 6874 it dropped to 3663, which is almost a 3211 difference

part ii) most important variables

In [25]:
```
pprint.pprint(sorted(list(zip(list(regr.feature_importances_), d
iabetes.feature_names))))
```

```
[(0.0, 's4'),
 (0.0013215904633095913, 's1'),
 (0.004595800056369401, 's6'),
 (0.005496902335978503, 'age'),
 (0.00668524154559176, 's2'),
 (0.008024265196098236, 'sex'),
 (0.046931600530010366, 's3'),
 (0.13088492712290042, 'bp'),
 (0.17683413186561717, 'bmi'),
 (0.6192255408841246, 's5')]
```

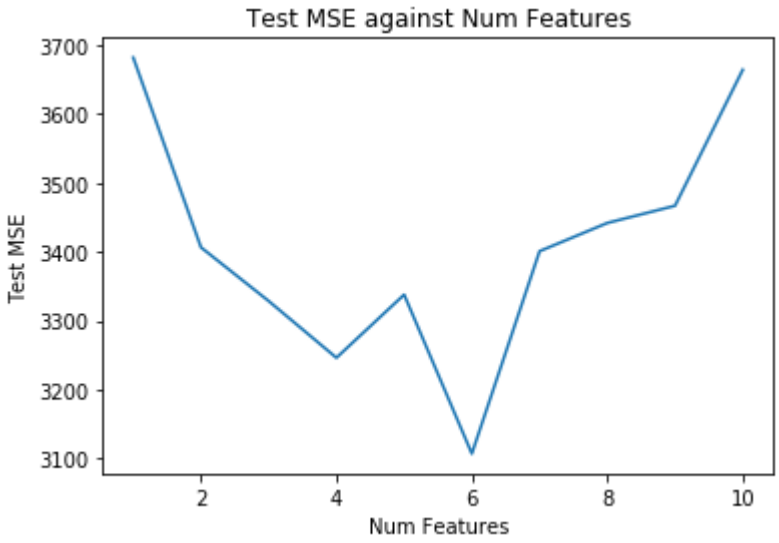
The random forest regressor shows the most important features as beingn `s5`, `bmi`, and `bp`, same as the single decision tree. However, the order of the top 2nd and 3rd features is switched (`bmi` takes second place).

part iii) plot MSE against m

In [ ]:
```
ftrs = diabetes.feature_names
mse_arr = []
ftrs_range = range(1, len(ftrs) + 1)
for i in ftrs_range:
    dtr = ensemble.RandomForestRegressor(random_state=0,
                                         min_samples_leaf=15,
                                         max_features = i)
    regr = dtr.fit(X_train, y_train)
    y_pred = regr.predict(X_test)
    mse_arr.append(mean_squared_error(y_test, y_pred))
```

In [27]:
```
plt.plot(ftrs_range, mse_arr)
plt.xlabel("Num Features")
plt.ylabel("Test MSE")
plt.title("Test MSE against Num Features")
```

Out[27]:
```
Text(0.5, 1.0, 'Test MSE against Num Features')
```



part iv) comment on plot

The plot does make sense and shows a bias-variance tradeoff. In other words, when the number of features is too low, variance is low but bias is high, so test MSE is high (under fitting). The optimal number is 6 features when test MSE is minimum. After that, test MSE increases again (over fitting) since bias is low but variance is high.

Question 2: Analyzing Real Estate Data (40 pts)

In this problem, you will train random forests on data from the website Zillow to forecast the sale price of real estate listings. Random forests are nonparametric methods for classification and regression. As discussed in class, the method is based on the following idea: a good predictor will have low bias and low variance. A deep decision tree has low bias, but high variance. To reduce the variance, multiple trees are fit and averaged together. By introducing randomness in the construction of the trees, the correlation between them is reduced, to facilitate the variance reduction.

Read in the training and test sets as follows:

```
import pandas as pd
train = pd.read_csv("zillow_train.csv")
test = pd.read_csv("zillow_test.csv")
```

Use the following variables: `Lat`, `Long`, `ListPrice`, `SaleYear`, `Bathroom`, `Bedroom`, `BuildDecade`, `MajorRenov`, `FinishSqFt`, `LotSqFt`, `MSA`, `City`, `HighSchool`, `SalePrice`. You will build regression models to predict `SalePrice`.

Import data

In [28]:
```
train = pd.read_csv("zillow_training.csv")
test = pd.read_csv("zillow_testing.csv")

feature_names = ["Lat",
                 "Long",
                 "ListPrice",
                 "SaleYear",
                 "Bathroom",
                 "Bedroom",
                 "BuildDecade",
                 "MajorRenov",
                 "FinishSqFt",
                 "LotSqFt",
                 "MSA",
                 "City",
                 "HighSchool",
                 "SalePrice"]
```

In [29]:
```
def dropAndSort(in_dat, feature_names_dat):
    one = in_dat[feature_names_dat]
    two = one.reindex(sorted(one.columns), axis=1)
    return two
```

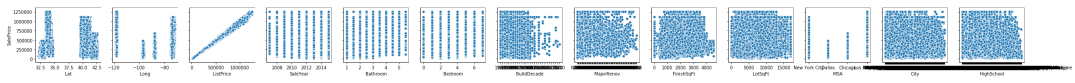
In [30]:
```
train = dropAndSort(train, feature_names)
test = dropAndSort(test, feature_names)
```

(a) Explore the data

Get an idea of what kind of data you're working with. As usual, you might ask yourself what n (sample size) and p (number of predictor variables) are here. Make plots of the distributions of the variables. Include a plot of the response, `SalePrice` . Does it appear that the data are "raw", or have they been pre-processed in different ways? If so, how?

In [31]:
```
x_vars = feature_names[:]
x_vars.remove("SalePrice")
sns.pairplot(train, y_vars=["SalePrice"], x_vars=x_vars)
```

Out[31]: `<seaborn.axisgrid.PairGrid at 0x11f39f750>`



- There are 13 predictor variables i.e. p
- The sample size is 77728 for training and 18682 for test i.e. n
- It does appear that the data has been pre-processed, because I am not seeing very many outliers, and it tends to look homogenous, across all relevant predictor variables.

Part (b) Preliminary steps

(i) Some of the variables in the data are categorical; how many values do they take? (You may find the `.nunique` method of pandas to be useful here.) Why might factor variables with many categories present a problem when fitting decision trees? Describe a couple different ways of handling factor variables when fitting decision trees.

How many unique values do the categorical variables take?

- `City`, `HighSchool`, `MSA` and `MajorRenov` are categorical, not ordinal.
- They take 1289, 654, 4 and 113 values respectively.

In [32]:
```
train.nunique()
```

Out[32]:
```
Bathroom          10
Bedroom            8
BuildDecade       29
City            1289
FinishSqFt      6222
HighSchool       654
Lat            72960
ListPrice       4774
Long           73090
LotSqFt        10374
MSA                4
MajorRenov       113
SalePrice       5052
SaleYear           9
dtype: int64
```

Hopefully, all the unique values of the test set are contained within the train set (although this can be confirmed with a computationally expensive search).

Here is an example of unique values for `MajorRenov`:

In [33]:
```
# print the last ten values
sorted(train.MajorRenov.unique())[-10:]
```

Out[33]:
```
['2007',
 '2008',
 '2009',
 '2010',
 '2011',
 '2012',
 '2013',
 '2014',
 '2015',
 'NONE']
```

`NONE` can be found in `MajorRenov` and in `HighSchool` but not the other columns:

In [34]:
```
assert('NONE' in train.MajorRenov.unique())
assert('NONE' in train.HighSchool.unique())
assert('NONE' not in train.City.unique())
assert('NONE' not in train.MSA.unique())
```

Finally, `BuildDecade` is numerical but might also give us trouble:

In [35]:
```
assert('UNKNOWN' in train.BuildDecade.unique())
```

Why might factor variables present a problem?

- From what I've read, sklearn specifically has trouble with categorical variables. And if you do numeric encoding, it assumes they are ordered somehow. That might be why one-hot encoding is preferred.
- Secondly, and perhaps more relevant, if a categorical variable has high cardinality/many choices, then the number of partitions is exponential in the number of categories. This might become very computationally expensive.

How to handle factor variables?

- One-hot encoding: convert factor variables to their numeric representations by introducing more columns, with the new columns indicating whether the factor is present or not
- Numeric encoding: convert each category to a number, counting upwards e.g. 1, 2, 3...
- Drop redudant factors: If a factor variable is already adequately represented by a numeric variable, you can probably drop it

(ii) Use your best judgement to modify the Zillow dataset to handle factor variables. In addition to `pandas` and `numpy`, it might be helpful to look at functions in `sklearn.preprocessing`.

In [36]:
```
# first step: fill in the missing data for BuildDecade
```

In [37]:
```
from sklearn.impute import SimpleImputer
```

In [38]:
```
imputer = SimpleImputer()
```

In [39]:
```
train["BuildDecade"] = pd.to_numeric(train["BuildDecade"], error
s="coerce")
train["BuildDecade"] = imputer.fit_transform(train["BuildDecad
e"].values.reshape(-1, 1))
```

In [40]:
```
test["BuildDecade"] = pd.to_numeric(test["BuildDecade"], error
s="coerce")
test["BuildDecade"] = imputer.transform(test["BuildDecade"].valu
es.reshape(-1, 1))
```

In [41]:
```
# next step: drop redudant columns
# City is adequately represented by Lat, Lng, so we can probably
drop it
```

In [42]:
```
train = train.drop(columns=["City"])
test = test.drop(columns=["City"])
```

In [43]:
```
# next step: numerically encode the rest of the columns
```

In [44]:
```
from sklearn.preprocessing import LabelEncoder
```

With `HighSchool` and `MSA` numeric encoding is hard to justify because it imposes an ordering on the categories, however I chose to do it for performance.

In [45]:
```
train_HighSchool_labels = set(train.HighSchool.unique())
test_HighSchool_labels = [l if l in train_HighSchool_labels else
"NONE" for l in test["HighSchool"]]
test["HighSchool"] = test_HighSchool_labels
```

In [46]:
```
le1 = LabelEncoder()
train["HighSchool"] = le1.fit_transform(train["HighSchool"])
test["HighSchool"] = le1.transform(test["HighSchool"])
```

In [47]:
```
le2 = LabelEncoder()
train["MSA"] = le2.fit_transform(train["MSA"])
test["MSA"] = le2.transform(test["MSA"])
```

With `MajorRenov` , numeric encoding makes the most sense because the categories are already ordered, with only `NONE` posing a problem.

In [48]:
```
train_MajorRenov_labels = set(train.MajorRenov.unique())
test_MajorRenov_labels = [l if l in train_MajorRenov_labels else
"NONE" for l in test["MajorRenov"]]
test["MajorRenov"] = test_MajorRenov_labels
```

In [49]:
```
le3 = LabelEncoder()
train["MajorRenov"] = le3.fit_transform(train["MajorRenov"])
test["MajorRenov"] = le3.transform(test["MajorRenov"])
```

(iii) We will soon use a few methods to predict `SalePrice` . Throughout, we will evaluate the predictions in terms of the absolute relative error:

$$\frac{1}{n}\sum_{i=1}^{n}\frac{|Y_i-\hat{Y}_i|}{Y_i}$$

Explain why this is a more appropriate choice of accuracy, compared with squared error.

- squared error exaggerates the effect of bad predictions, since differences are squared
- absolute error is on the same scale as that of the predictions, and it does not exaggerate the effect of bad predictions

Part (c) Build models using random forests

Build random forest models to predict `SalePrice` from the other variables, using the appropriate method from `sklearn.ensemble`. As in Question 1, one parameter to vary is `max_features`, or the number of variables allowed in each split; this regulates the correlation between the trees in the random forest by introducing randomness. Two more relevant parameters are `n_estimators` and `min_samples_leaf`, or number of trees and minimum node size, which regulate variance and bias.

Train several random forest models, each time using different values of the parameters. Evaluate each model using 5-fold cross-validation (`sklearn.model_selection.KFold` may be a useful resource to perform k-fold cross-validation). For the sake of time, you may keep `n_estimators` low and constant. First vary `max_features` and create a plot of the cross-validation error versus the value of this parameter. Next vary `min_samples_leaf` a create a similar plot with the values of this parameter.

Comment on how cross-validation error relates to `max_features` and `min_samples_leaf`, and how do you imagine it would relate to `n_estimators`? Does this make sense to you?

Now find a combination of values for `max_features` and `min_samples_leaf` that approximately minimizes the cross-validation error.

Note: Use mean absolute error (`mae`) rather than mean squared error (`mse`) as the criterion for growing the trees. But then when you evaluate different models, compute the relative absolute error, as described above.

In [50]:
```
from sklearn import model_selection
```

In [51]:
```
# the two things that can vary are: 1) max_features, 2) min_samp
les_leaf
# n_estimators can be kept constant
def modelCreator(p_maxFeatures='auto', p_minSamplesLeaf=1):
    return ensemble.RandomForestRegressor(n_estimators = 15,
                                          min_samples_leaf = p_m
inSamplesLeaf,
                                          max_features = p_maxFe
atures,
                                          criterion = 'mse',
                                          random_state=0)
```

In [52]:
```
# separate the train and test data into features and labels
y_train = train[["SalePrice"]]
X_train = train.drop(columns=["SalePrice"])

y_test = test[["SalePrice"]]
X_test = test.drop(columns=["SalePrice"])
```

5-fold validation, varying `max_features`

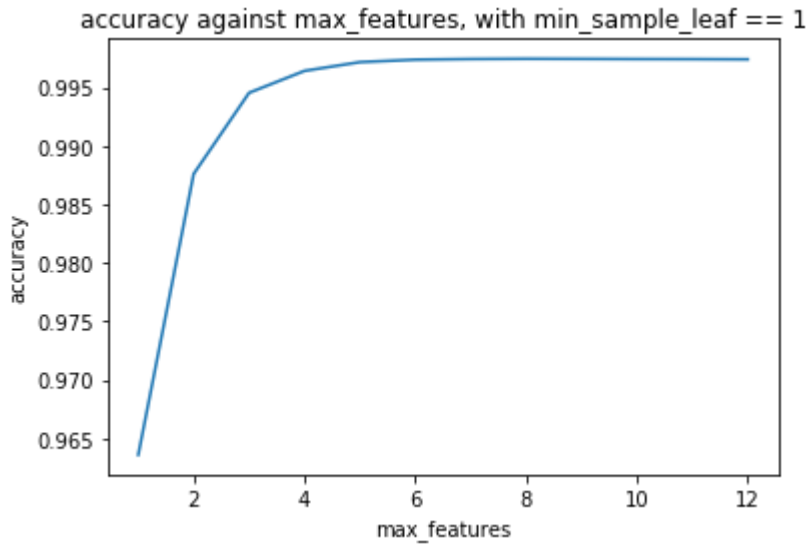
In [ ]:
```
max_features_range = range(1, 13) # from 1 to 12 (all remaining
features)
accuracies = []
for mf in max_features_range:
    cv_accuracies = model_selection.cross_val_score(
                          estimator=modelCreator(p_maxFeatures=m
f),
                          X=X_train,
                          y=y_train,
                          cv=5)
    mean_accuracy = np.mean(cv_accuracies)
    accuracies.append(mean_accuracy)
```

In [54]:
```
accuracies
```

Out[54]:
```
[0.963615767424707,
 0.9876147016299933,
 0.9945438762066164,
 0.9964176585984701,
 0.997165651112347,
 0.9973752868371477,
 0.9974291791325243,
 0.9974498812663798,
 0.9974408828573086,
 0.9974272453202613,
 0.9974200422071714,
 0.9974039268100453]
```

In [55]:
```
plt.plot(max_features_range, accuracies)
plt.xlabel("max_features")
plt.ylabel("accuracy")
plt.title("accuracy against max_features, with min_sample_leaf
== 1")
```

Out[55]:
```
Text(0.5, 1.0, 'accuracy against max_features, with min_sample_
leaf == 1')
```



- `max_features == 5` seems pretty optimal
- the trend makes sense, as the more features we include, complexity increases and over-fit to training

5-fold validation, varying `min_samples_leaf`

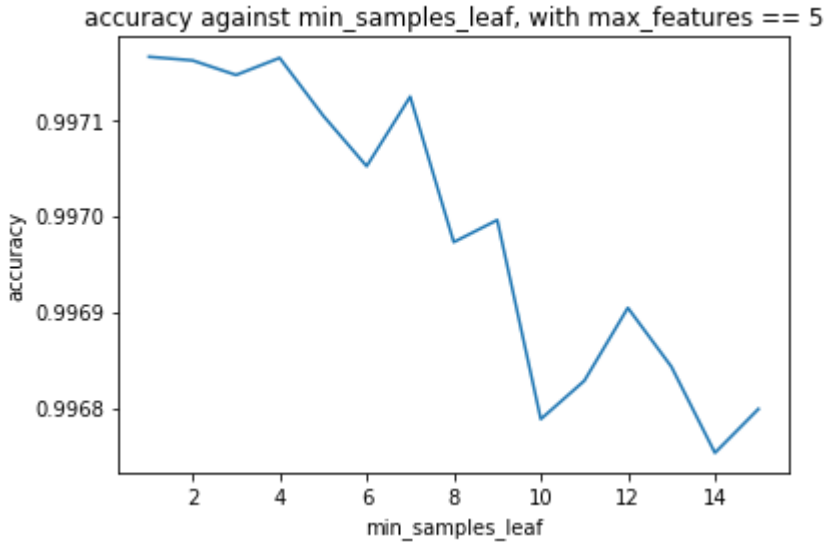
In [ ]:
```
min_samples_leaf_range = range(1, 16) # from 1 to 15 (all remain
ing features)
accuracies = []
for msl in min_samples_leaf_range:
    cv_accuracies = model_selection.cross_val_score(
                          estimator=modelCreator(p_minSamplesLeaf=
msl,
                                                 p_maxFeatures=5),
                          X=X_train,
                          y=y_train,
                          cv=5)
    mean_accuracy = np.mean(cv_accuracies)
    accuracies.append(mean_accuracy)
```

In [57]:
```
accuracies
```

Out[57]:
```
[0.997165651112347,
 0.9971618716080117,
 0.9971466883323015,
 0.9971644947138776,
 0.9971043857870436,
 0.9970517343744547,
 0.9971241567834083,
 0.9969726935130364,
 0.9969956905839098,
 0.9967883808128191,
 0.9968283271520066,
 0.9969042906110959,
 0.9968429975068748,
 0.996753304063809,
 0.9967989472011455]
```

In [58]:
```
plt.plot(min_samples_leaf_range, accuracies)
plt.xlabel("min_samples_leaf")
plt.ylabel("accuracy")
plt.title("accuracy against min_samples_leaf, with max_features
== 5")
```

Out[58]:
```
Text(0.5, 1.0, 'accuracy against min_samples_leaf, with max_fea
tures == 5')
```



- `min_samples_leaf == 4` seems optimal, with `max_features == 5`
- trend makes sense, as more samples in the leaves reduces complexity and over-fit

Part (d) Comparison to Least-Squares Regression

Now build a least-squares linear model for the response variable as a function of the predictor variables using the training set. You may wish to use the `sklearn.linear_model.LinearRegression` function, described here (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html).

Experiment with different subsets of the predictor variables included in the linear model.

Using the random forest model from Part (c) with the best combination of values for `max_features` and `min_samples_leaf` that you found, compare both the mean squared error and the relative absolute error on the test set from the random forest and linear models.

Which model does a better job at prediction? Do you think the model with the higher MSE has higher variance or higher bias, or both?

In [59]:
```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error
```

In [60]:
```
lr = LinearRegression()
lr.fit(X_train, y_train)
lr_pred = lr.predict(X_test)
```

In [61]:
```
print(mean_squared_error(y_test, lr_pred))
print(mean_absolute_error(y_test, lr_pred))
```

```
205817978.5303931
9716.564219157435
```

In [62]:
```
dtr = modelCreator(p_maxFeatures=5, p_minSamplesLeaf=4)
regr = dtr.fit(X_train, y_train)
tree_pred = regr.predict(X_test)
```

```
/Users/sarimabbas/Developer/dataScience/sds355_container/sds355
_env/lib/python3.7/site-packages/ipykernel_launcher.py:2: DataC
onversionWarning: A column-vector y was passed when a 1d array
was expected. Please change the shape of y to (n_samples,), for
example using ravel().
```

In [63]:
```
print(mean_squared_error(y_test, tree_pred))
print(mean_absolute_error(y_test, tree_pred))
```

```
195324718.8487553
9372.984191348974
```

- The MSE of the random forest is much lower than the linear regression, by about 10493260
- The MAE of both is much lower, since the differences aren't exaggerated due to squaring. But random forests are still better by about 344
- Random forests is a better model overall
- The model with the higher MSE is probably with higher variance, not higher bias, because the Linear Model has sufficiently high complexity to reduce bias

Part (e) Predicting SalePrice

Read in the file "zillow_part_e.csv" which has 7000 houses with all the same variables as the training and testing set, except that the SalePrice variable is missing.

Construct the best model you can on the training data. You can use random forests, or you may try to use gradient tree boosting, which is also available in sklearn.ensemble.

Using your best model, predict the sale prices for these 7000 houses. Students will be assigned extra credit according to which decile they are in for the predictive accuracy (relative absolute error). (The top 10% will receive 10 points extra credit, the next 10% 9 points, and so on.)

Save your predictions in a file called "zillow_predictions.csv" and submit this file with your homework. Your csv file should only contain a single column of predictions, without a header, where the i-th row corresponds to the predicted sale price for the i-th row of the dataset read in from "zillow_part_e.csv", excluding the header.

In [64]:
```
# Your Code Here
newdata = pd.read_csv("zillow_part_e.csv")
```

In [65]:
```
# do processing
newdata = dropAndSort(newdata, feature_names[:-1])
newdata = newdata.drop(columns=["City"])
```

In [66]:
```
# imputation
newdata["BuildDecade"] = pd.to_numeric(newdata["BuildDecade"], e
rrors="coerce")
newdata["BuildDecade"] = imputer.transform(newdata["BuildDecad
e"].values.reshape(-1, 1))
```

In [67]:
```
# encode categorical variables
newdata_HighSchool_labels = [l if l in train_HighSchool_labels e
lse "NONE" for l in newdata["HighSchool"]]
newdata["HighSchool"] = newdata_HighSchool_labels
```

In [68]:
```
newdata_MajorRenov_labels = [l if l in train_MajorRenov_labels e
lse "NONE" for l in newdata["MajorRenov"]]
newdata["MajorRenov"] = newdata_MajorRenov_labels
```

In [69]:
```
newdata["HighSchool"] = le1.transform(newdata["HighSchool"])
newdata["MSA"] = le2.transform(newdata["MSA"])
newdata["MajorRenov"] = le3.transform(newdata["MajorRenov"])
```

In [70]:
```
dtr = modelCreator(p_maxFeatures=5, p_minSamplesLeaf=4)
regr = dtr.fit(X_train, y_train)
y_pred = regr.predict(newdata)
```

```
/Users/sarimabbas/Developer/dataScience/sds355_container/sds355
_env/lib/python3.7/site-packages/ipykernel_launcher.py:2: DataC
onversionWarning: A column-vector y was passed when a 1d array
was expected. Please change the shape of y to (n_samples,), for
example using ravel().
```

In [71]:
```
exportFrame = pd.DataFrame(y_pred, columns=["SalePrice"])
exportFrame.to_csv("./zillow_predictions.csv", index=None)
```