databricks hw06_solution (1)

# The Diamonds Data

For this assignment we will use the diamonds dataset which can be found on the distributed filesystem here:

```
dbfs:/databricks-datasets/Rdatasets/data-001/csv/ggplot2/diamonds.csv
```

To learn more about the data read the ggplot description (http://ggplot2.tidyverse.org/reference/diamonds.html).

## Question 2.1

In the following cell use the (Databricks) magic `%fs ls <path>` to compute the size of this file. Copy the size into the Jupyter notebook for this assignment.

```
%fs ls dbfs:/databricks-datasets/Rdatasets/data-001/csv/ggplot2/diamonds.csv
```

| path |
| --- |
| dbfs:/databricks-datasets/Rdatasets/data-001/csv/ggplot2/diamonds.csv |

We can examine the top of the file using the `%fs head <path>` command.

```
%fs head dbfs:/databricks-datasets/Rdatasets/data-001/csv/ggplot2/diamonds.csv
```

```
[Truncated to first 65536 bytes]
"","carat","cut","color","clarity","depth","table","price","x","y","z"
"1",0.23,"Ideal","E","SI2",61.5,55,326,3.95,3.98,2.43
"2",0.21,"Premium","E","SI1",59.8,61,326,3.89,3.84,2.31
"3",0.23,"Good","E","VS1",56.9,65,327,4.05,4.07,2.31
"4",0.29,"Premium","I","VS2",62.4,58,334,4.2,4.23,2.63
"5",0.31,"Good","J","SI2",63.3,58,335,4.34,4.35,2.75
"6",0.24,"Very Good","J","VVS2",62.8,57,336,3.94,3.96,2.48
"7",0.24,"Very Good","I","VVS1",62.3,57,336,3.95,3.98,2.47
"8",0.26,"Very Good","H","SI1",61.9,55,337,4.07,4.11,2.53
"9",0.22,"Fair","E","VS2",65.1,61,337,3.87,3.78,2.49
"10",0.23,"Very Good","H","VS1",59.4,61,338,4,4.05,2.39
"11",0.3,"Good","J","SI1",64,55,339,4.25,4.28,2.73
"12",0.23,"Ideal","J","VS1",62.8,56,340,3.93,3.9,2.46
"13",0.22,"Premium","F","SI1",60.4,61,342,3.88,3.84,2.33
"14",0.31,"Ideal","J","SI2",62.2,54,344,4.35,4.37,2.71
"15",0.2,"Premium","E","SI2",60.2,62,345,3.79,3.75,2.27
"16",0.32,"Premium","E","I1",60.9,58,345,4.38,4.42,2.68
"17",0.3,"Ideal","I","SI2",62,54,348,4.31,4.34,2.68
"18",0.3,"Good","J","SI1",63.4,54,351,4.23,4.29,2.7
"19",0.3,"Good","J","SI1",63.8,56,351,4.23,4.26,2.71
```

# Question 2.2: Loading the Data

Load the data into a `PySpark.DataFrame` called `data` by using the `spark.read.csv` command (https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrameRe
We will want infer the schema and incorporate the header information so look at the options:

1. `inferSchema`
2. `header`

```
# Use this Cell:
data = spark.read.csv("dbfs:/databricks-datasets/Rdatasets/data-
001/csv/ggplot2/diamonds.csv",
                      inferSchema=True, header=True)
```

Once you have loaded the data compute the number of rows using `count` method of dataframes (https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.co
Save your answer in the jupyter notebook for grading.

```
data.count()
```

```
Out[2]: 53940
```

Use the `df.show()` function to examine a few rows in the dataframe.

```
data.show()
```

```
+---+-----+---------+-----+-------+-----+-----+-----+----+----+----+
|_c0|carat|      cut|color|clarity|depth|table|price|   x|   y|   z|
+---+-----+---------+-----+-------+-----+-----+-----+----+----+----+
|  1| 0.23|    Ideal|    E|    SI2| 61.5| 55.0|  326|3.95|3.98|2.43|
|  2| 0.21|  Premium|    E|    SI1| 59.8| 61.0|  326|3.89|3.84|2.31|
|  3| 0.23|     Good|    E|    VS1| 56.9| 65.0|  327|4.05|4.07|2.31|
|  4| 0.29|  Premium|    I|    VS2| 62.4| 58.0|  334| 4.2|4.23|2.63|
|  5| 0.31|     Good|    J|    SI2| 63.3| 58.0|  335|4.34|4.35|2.75|
|  6| 0.24|Very Good|    J|   VVS2| 62.8| 57.0|  336|3.94|3.96|2.48|
|  7| 0.24|Very Good|    I|   VVS1| 62.3| 57.0|  336|3.95|3.98|2.47|
|  8| 0.26|Very Good|    H|    SI1| 61.9| 55.0|  337|4.07|4.11|2.53|
|  9| 0.22|     Fair|    E|    VS2| 65.1| 61.0|  337|3.87|3.78|2.49|
| 10| 0.23|Very Good|    H|    VS1| 59.4| 61.0|  338| 4.0|4.05|2.39|
| 11|  0.3|     Good|    J|    SI1| 64.0| 55.0|  339|4.25|4.28|2.73|
| 12| 0.23|    Ideal|    J|    VS1| 62.8| 56.0|  340|3.93| 3.9|2.46|
| 13| 0.22|  Premium|    F|    SI1| 60.4| 61.0|  342|3.88|3.84|2.33|
| 14| 0.31|    Ideal|    J|    SI2| 62.2| 54.0|  344|4.35|4.37|2.71|
| 15|  0.2|  Premium|    E|    SI2| 60.2| 62.0|  345|3.79|3.75|2.27|
| 16| 0.32|  Premium|    E|     I1| 60.9| 58.0|  345|4.38|4.42|2.68|
| 17|  0.3|    Ideal|    I|    SI2| 62.0| 54.0|  348|4.31|4.34|2.68|
| 18|  0.3|     Good|    J|    SI1| 63.4| 54.0|  351|4.23|4.29| 2.7|
| 19|  0.3|     Good|    J|    SI1| 63.8| 56.0|  351|4.23|4.26|2.71|
| 20|  0.3|Very Good|    J|    SI1| 62.7| 59.0|  351|4.21|4.27|2.66|
+---+-----+---------+-----+-------+-----+-----+-----+----+----+----+
only showing top 20 rows
```

# Question 2.3: Train Test Split.

Use the `df.randomSplit` command (https://spark.apache.org/docs/latest/api/python/pyspark.sql.html#pyspark.sql.DataFrame.ran to split the data into a 90% and 10% train-test split with Professor Gonalez's favorite seed (42). Name the training data `data_tr` and the test data `data_te` . Then report the size of the training data in the Jupyter notebook for this homework.

```
(data_tr, data_te) = data.randomSplit([0.9, 0.1], seed=42)
```

```
data_tr.count()
```

```
Out[5]: 48646
```

To speed up computation use the following cell to cache `data_tr` and `data_te` by invoking their `persist()` method.

```
data_tr.persist()
data_te.persist()
```

```
Out[6]: DataFrame[_c0: int, carat: double, cut: string, color: string, clarity: string
, depth: double, table: double, price: int, x: double, y: double, z: double]
```

# Question 2.4: DF to RDD

The Spark DataFrame is actually a DataFrame API that runs on top of RDDs (Resilient Distributed Datasets). However for this assingment we will want to work with the RDD directly. This will allow us to apply arbitrary Python code to the data without worrying about the schema. Use the `df.rdd` member to extract the RDD and then use the `rdd.take` command to get the first two records from the training data ( `data_tr` ):

```
# Solution
data_tr.rdd.take(2)
```

```
Out[7]:
```

```
[Row(_c0=1, carat=0.23, cut=u'Ideal', color=u'E', clarity=u'SI2', depth=61.5, table=55
.0, price=326, x=3.95, y=3.98, z=2.43),
 Row(_c0=2, carat=0.21, cut=u'Premium', color=u'E', clarity=u'SI1', depth=59.8, table=
61.0, price=326, x=3.89, y=3.84, z=2.31)]
```

The RDD is composed of SparkSQL Row objects that behave much like python dictionaries. Using `rdd.map(...).mean(...)` compute the average price of the diamonds in the training data. Copy your answer into the Jupyter notebook.

```
# Solution
data_tr.rdd.map(lambda x: x['price']).mean()
```

```
Out[8]: 3921.282674834481
```

# Question 3

From class, when we solve the normal equations we obtain the following estimate for our linear model:

$$\hat{\theta} = (X^T X)^{-1}(X^T y)$$

In lab10, we already saw how to compute the formula using basic linear algebra. Now we're going to implement this in Spark. The major difference is that, in lab you have a matrix representation for X, while now we have a RDD.

The rows in the RDD are equivalent to the rows in the matrix X. However, we can't directly compute outer product on the RDD. An alternative solution is we first comput the outer product for each row of the matrix (RDD), then we compute the sum of those results. For example, if

$$X = [[1, 2], [3, 4], [5, 6]]$$

then:

$$X^T X = [1, 2]^T [1, 2] + [3, 4]^T [3, 4] + [5, 6]^T [5, 6]$$

Therefore we can use the RDD `map()` function to compute the outer product for each row of the RDD, then use the `sum()` to collect the final result.

On the other hand, each row of the RDD is a dictionary-like data structure, and it contains both the features and the target that we want to predict. (For example, we may want to use the carat and depth to predict the price). Hence we should select a subset of the features. This will be done by providing a list of the feature names.

Now your task is to finish the following `compute_XtX` function. This function will be applied to each row of the RDD and it should return the outer product of the selected features in that row.

```
import numpy as np

def compute_XtX(row, features, add_intercept=True):
  """
  Compute XtX (the outer product) for a single row of data.
    row: is a SparkSQL Row (behaves like a dictionary)
    features: the list of feature names (column names to use as features)
    add_intercept: default is true, adds an addition 1.0 feature

    return: the outer product of the features.
  """

  return None # Finish
```

```
#solution
import numpy as np

def compute_XtX(row, features, add_intercept=True):
  """
  Compute XtX (the outer product) for a single row of data.
    row: is a SparkSQL Row (behaves like a dictionary)
    features: the list of feature names (column names to use as features)
    add_intercept: defaut is true, adds an addition 1.0 feature

    return: the outer product of the features.
  """
  features = [row[k] for k in features]
  if add_intercept:
    features += [1.0]
  x = np.array(features)
  return np.outer(x, x)



assert np.allclose(data_tr.rdd.map(lambda r:compute_XtX(r,["carat"])).take(2),[[[
0.0529,  0.23  ],[ 0.23  ,  1.    ]],[[ 0.0441,  0.21  ],[ 0.21  ,  1.    ]]])
```

Then we need to finish the `compute_XtY` function, where the y is the target. Here the target will be given as one column name of the RDD row.

Same with `compute_XtX` , the function `compute_XtY` you write will be applied on each row of the RDD and it should return a numpy array.

```
def compute_XtY(row, features, target, add_intercept=True):
  """
  Compute XtY (the outer product) for a single row of data.
    row: is a SparkSQL Row (behaves like a dictionary)
    features: the list of feature names (column names to use as features)
    target: the column name containing the target we are trying to predict
    add_intercept: defaut is true, adds an addition 1.0 feature

    return: the the features times the target
  """
  return None # Finish Me
```

```
#solution
def compute_XtY(row, features, target, add_intercept=True):
  """
  Compute XtY (the outer product) for a single row of data.
    row: is a SparkSQL Row (behaves like a dictionary)
    features: the list of feature names (column names to use as features)
    target: the column name containing the target we are trying to predict
    add_intercept: defaut is true, adds an addition 1.0 feature

    return: the the features times the target
  """
  features = [row[k] for k in features]
  if add_intercept:
    features += [1.0]
  x = np.array(features)
  y = np.array(row[target])
  return x * y



assert np.allclose(data_tr.rdd.map(lambda r:compute_XtY(r,["carat"],"price")).take(2),
[[  74.98,  326.  ], [  68.46,  326.  ]])
```

So now let's run the following code to create the `XtX` and `XtY` matrix on the whole dataset.

```
features = ["carat", "depth", "table"]
target = "price"

XtX = data_tr.rdd.map(lambda row: compute_XtX(row, features)).sum()
XtY = data_tr.rdd.map(lambda row: compute_XtY(row, features, target)).sum()


XtX

Out[16]:
array([[  4.17405461e+04,   2.39356770e+06,   2.23565449e+06,
          3.87499900e+04],
       [  2.39356770e+06,   1.85569078e+08,   1.72523681e+08,
          3.00372130e+06],
       [  2.23565449e+06,   1.72523681e+08,   1.60808709e+08,
          2.79481180e+06],
       [  3.87499900e+04,   3.00372130e+06,   2.79481180e+06,
```

```
          4.86460000e+04]])
```

```
XtY
```

```
Out[17]:
array([  2.36388638e+08,   1.17754140e+10,   1.10154318e+10,
         1.90754717e+08])
```

# Question 3.1

Now let's use the `np.linalg.solve` to compute the \theta. Copy your answer into the ipython notebook

```
theta = None # Finishe Me ...
```

```
#solution
theta = np.linalg.solve(XtX, XtY)
theta
```

```
Out[19]: array([  7865.41870378,   -147.65515738,   -101.57023038,  12608.52774277])
```

```
assert np.allclose(theta,[  7865.41870378,   -147.65515738,   -101.57023038,
12608.52774277])
```

# Question 3.2:

It seems like the weight of `carat` is way bigger than the other two, could we say it is the dominating feature?

solution:

No, since the usual value of feature `carat` is small than 1, while the other two are around 60.

# Question 3.3

Now let's get some measurement about how well our estimation is. We can compute the RMSE (root mean squared error) for the given \theta.

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\theta^T X_i - Y_i)^2}$$

Again, we first need to compute the squared error for each row, and then use the `.mean()` and `np.sqrt` to get the RMSE. Copy your answer into the ipython notebook

```
def se(row, theta, features, target, add_intercept=True):
    """
    Compute the squared error for the parameters theta on a single row of data

        row: is a SparkSQL Row (behaves like a dictionary)
        theta: a numpy array
        features: the list of feature names (column names to use as features)
        target: the column name containing the target we are trying to predict
        add_intercept: defaut is true, adds an addition 1.0 feature

    return  (X.dot(theta) - Y)^2

    """
    return None # Finish Me
```

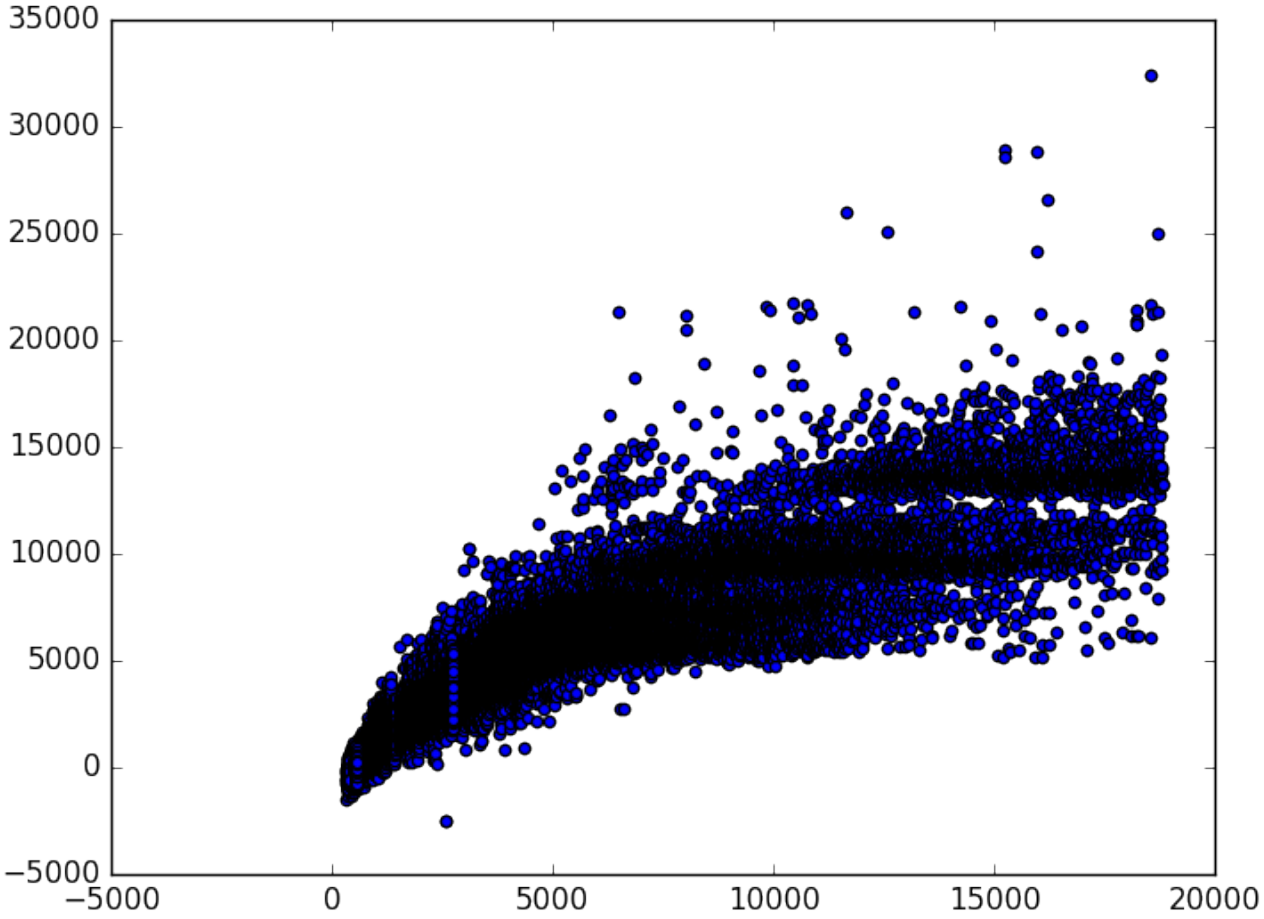
```
#solution
def se(row, theta, features, target, add_intercept=True):
    """
    Compute the squared error for the parameters theta on a single row of data

        row: is a SparkSQL Row (behaves like a dictionary)
        theta: a numpy array
        features: the list of feature names (column names to use as features)
        target: the column name containing the target we are trying to predict
        add_intercept: defaut is true, adds an addition 1.0 feature

    return  (X.dot(theta) - Y)^2

    """
    features = [row[k] for k in features]
    if add_intercept:
      features += [1.0]
    x = np.array(features)
    y = np.array(row[target])
    return (x.dot(theta) - y)**2


rmse = None # Finishe Me


#solution
rmse = np.sqrt(data_tr.rdd.map(lambda r:se(r,theta,features,target)).mean())
rmse

Out[24]: 1522.3582303379176

assert np.allclose(rmse, 1522.3582303379176)
```

It seems the error is quite big. Let's try to look at the prediction directly. Finish the following function to compute the actual predictions. It should be very similar to the `se` function.

```
def predict(row, theta, features, add_intercept=True):
    """
    Compute the actual prediction for the parameters theta on a single row of data

        row: is a SparkSQL Row (behaves like a dictionary)
        theta: a numpy array
        features: the list of feature names (column names to use as features)
        add_intercept: defaut is true, adds an addition 1.0 feature

    return  X.dot(theta)

    """
    return None # Finish Me


#solution
def predict(row, theta, features, add_intercept=True):
    """
    Compute the actual prediction for the parameters theta on a single row of data

        row: is a SparkSQL Row (behaves like a dictionary)
        theta: a numpy array
        features: the list of feature names (column names to use as features)
        add_intercept: defaut is true, adds an addition 1.0 feature

    return  X.dot(theta)

    """
    features = [row[k] for k in features]
    if add_intercept:
      features += [1.0]
    x = np.array(features)
    return x.dot(theta)
```

Then let's plot the predicted value verse the real value

```
import matplotlib.pyplot as plt
price_real =data_tr.rdd.map(lambda x:x["price"]).collect()
price_predict = data_tr.rdd.map(lambda r:predict(r,theta,features)).collect()
plt.scatter(price_real,price_predict)
display()
```




# Question 3.4

Now, let's try to add some more features to see if they can improve our prediction. You can add more column names into the `features` list. Save your rmse into the ipython notebook

```
features = ["carat", "depth"] # Add some more here
target = "price"

XtX = data_tr.rdd.map(lambda row: compute_XtX(row, features)).sum()
XtY = data_tr.rdd.map(lambda row: compute_XtY(row, features, target)).sum()
theta = None # Finish Me
rmse = None # Finish Me
```

```
#solution
features = ["carat", "depth", "table","x"]
target = "price"

XtX = data_tr.rdd.map(lambda row: compute_XtX(row, features)).sum()
XtY = data_tr.rdd.map(lambda row: compute_XtY(row, features, target)).sum()
theta = np.linalg.solve(XtX, XtY)
rmse = np.sqrt(data_tr.rdd.map(lambda r:se(r,theta,features,target)).mean())
rmse

Out[32]: 1490.4978047105708

assert rmse < 1500
```

# One Hot Encoding Categorical Features

In the following we extend the training and test data with a few dummy columns.

```
print data_tr.select('cut').distinct().collect()
print data_tr.select('clarity').distinct().collect()

[Row(cut=u'Premium'), Row(cut=u'Ideal'), Row(cut=u'Good'), Row(cut=u'Fair'), Row(cut=u
'Very Good')]
[Row(clarity=u'VVS2'), Row(clarity=u'SI1'), Row(clarity=u'IF'), Row(clarity=u'I1'), Ro
w(clarity=u'VVS1'), Row(clarity=u'VS2'), Row(clarity=u'SI2'), Row(clarity=u'VS1')]
```

```
def add_columns(df):
    return (df
            .withColumn("cut_premium", (data_tr['cut'] == "Premium").astype("double"))
            .withColumn("cut_ideal", (data_tr['cut'] == "Ideal").astype("double"))
            .withColumn("cut_verygood", (data_tr['cut'] == "Very
Good").astype("double"))
            .withColumn("cut_good", (data_tr['cut'] == "Good").astype("double"))
            .withColumn("cut_fair", (data_tr['cut'] == "Fair").astype("double"))
            .withColumn("clarity_VVS2", (data_tr['clarity'] ==
"VVS2").astype("double"))
            .withColumn("clarity_SI1", (data_tr['clarity'] == "SI1").astype("double"))
            .withColumn("clarity_IF", (data_tr['clarity'] == "IF").astype("double"))
            .withColumn("clarity_I1", (data_tr['clarity'] == "I1").astype("double"))
            .withColumn("clarity_VVS1", (data_tr['clarity'] ==
"VVS1").astype("double"))
            .withColumn("clarity_VS2", (data_tr['clarity'] == "VS2").astype("double"))
            .withColumn("clarity_SI2", (data_tr['clarity'] == "SI2").astype("double"))
            .withColumn("clarity_VS1", (data_tr['clarity'] == "VS1").astype("double"))
           )
data_tr = add_columns(data_tr)
data_te = add_columns(data_te)


display(data_tr)
```

| _c0 | carat | cut | color | clarity | depth | table | price | x | y | z | cut_premium | cut_ideal | cut_v |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 0.23 | Ideal | E | SI2 | 61.5 | 55 | 326 | 3.95 | 3.98 | 2.43 | 0 | 1 | 0 |
| 2 | 0.21 | Premium | E | SI1 | 59.8 | 61 | 326 | 3.89 | 3.84 | 2.31 | 1 | 0 | 0 |
| 4 | 0.29 | Premium | I | VS2 | 62.4 | 58 | 334 | 4.2 | 4.23 | 2.63 | 1 | 0 | 0 |
| 5 | 0.31 | Good | J | SI2 | 63.3 | 58 | 335 | 4.34 | 4.35 | 2.75 | 0 | 0 | 0 |
| 6 | 0.24 | Very Good | J | VVS2 | 62.8 | 57 | 336 | 3.94 | 3.96 | 2.48 | 0 | 0 | 1 |
| 7 | 0.24 | Very Good | I | VVS1 | 62.3 | 57 | 336 | 3.95 | 3.98 | 2.47 | 0 | 0 | 1 |
| 8 | 0.26 | Very | H | SI1 | 61.9 | 55 | 337 | 4.07 | 4.11 | 2.53 | 0 | 0 | 1 |

Showing the first 1000 rows.

# Go Crazy :)

## Question 3.5

Try adding more features to see how low of a training error you can achieve. Paste your answer into the ipython notebook

```
# Thinks to consider:
#             "cut_premium", "cut_ideal", "cut_verygood", "cut_good", "cut_fair",
#             "clarity_VVS1", "clarity_VVS2", "clarity_SI1", "clarity_IF",
"clarity_I1",
#             "clarity_VS2"
# you can add more above as well.
# We got to:
#     train_rmse =  1336.79948051
#     test_rmse =  1391.14006912

# Have fun
features = ["carat", "depth", "table","x"]
target = "price"

XtX = data_tr.rdd.map(lambda row: compute_XtX(row, features)).sum()
XtY = data_tr.rdd.map(lambda row: compute_XtY(row, features, target)).sum()
theta = np.linalg.solve(XtX, XtY)
train_rmse = np.sqrt(data_tr.rdd.map(lambda r:se(r,theta,features,target)).mean())
test_rmse = np.sqrt(data_te.rdd.map(lambda r:se(r,theta,features,target)).mean())
print "train_rmse = ", train_rmse
print "test_rmse = ", test_rmse
```

```
train_rmse =  1490.49780471
test_rmse =  1556.08841125

assert test_rmse < 1400
```

```
train_rmse =  1336.79948051
test_rmse =  1391.14006912
```