Predicting sales -- Bread

This notebook attempts to build a model to predict the sales of bread products at an Ecuador supermarket chain. In particular, we are trying to predict "Unit Sales" for each unique Store-ItemID-Date combination

In [74]:

```
import pandas as pd
import numpy as np
import datetime
pd.set_option('display.max_columns', None)
from sklearn import linear_model
from sklearn import metrics
from sklearn import tree, ensemble
```

In [2]:

```
def loss_function(y_pred, y_true):
    '''Loss function is root mean squared logarithmic error'''
    MSLE = metrics.mean_squared_log_error(y_pred, y_true)
    RMSLE = MSLE**0.5
    return RMSLE

def add_features_from_previous_weeks(df, df_string):
    '''
    This function builds features based on prior unit sales for a particular
    store-item combination.

    In particular, for each unique store-item-date combination in the DF, the fu
nction
    adds new columns corresponding to the t-2 week, t-3 week, and t-52 week obse
rvations for
    the same store-item combination.

    For example, to predict unit sales for "Cornbread" from Store #25 on 7/30/20
17,
    the function will add features for Store #25's Cornbread sales on 7/16/2017,
7/9/2017,
    and 7/30/2016.

    I started at t-2 (and not t-1), because the unknown test set spans two weeks
. As such, test
    set observations in the second week would have NaN values for the t-1 column
(unless I filled
    these empty cells with the predictions from the first week of test set resul
ts...)

    With more time, I would done additional research (or look up kernels) to emp
```

```
loy

    time series models.
    '''

    w2 = df.rename(columns={'woy':'pwoy', 'year':'pyear', 'unit_sales':'sales_w2
_'+df_string})
    w2['woy'] = np.where(w2['pwoy']>50, (w2['pwoy']+2)%52 , w2['pwoy']+2)
    w2['year'] = np.where(w2['pwoy']>50, w2['pyear']+1 , w2['pyear'])
    w2 = w2.drop(['pwoy', 'pyear'], 1)

    w3 = df.rename(columns={'woy':'pwoy', 'year':'pyear', 'unit_sales':'sales_w3
_'+df_string})
    w3['woy'] = np.where(w3['pwoy']>49, (w3['pwoy']+3)%52 , w3['pwoy']+3)
    w3['year'] = np.where(w3['pwoy']>49, w3['pyear']+1 , w3['pyear'])
    w3 = w3.drop(['pwoy', 'pyear'], 1)

    w52 = df.rename(columns={'woy':'pwoy', 'year':'pyear', 'unit_sales':'sales_w
52_'+df_string})
    w52['woy'] = w52['pwoy']
    w52['year'] = w52['pyear']+1
    w52 = w52.drop(['pwoy', 'pyear'], 1)

    merge_cols = ['woy', 'year', 'store_nbr', 'item_nbr']

    if 'weekday' in df_string:
        merge_cols.append('weekday')

    w = w2.merge(w3, on=merge_cols, how='outer').\
           merge(w52, on=merge_cols, how='outer')

    for t in [2, 3, 52]:
        merge_cols.append('sales_w'+str(t)+'_'+df_string)

    w = w[merge_cols].reset_index(drop=True)
    w['sales_w2_'+df_string] = w['sales_w2_'+df_string].fillna(0)
    return w

#-----------------------------------------------------

def add_features_from_previous_months(df, df_string):
    '''
    Similar to "add_features_from_previous_weeks" function, except this time we
are looking at
    month-level data.

    For example, to predict unit sales for "Cornbread" from Store #25 on Friday,
7/14/2017,
    the function will add features for Store #25's average cornbread sales on Fr
idays in
    June 2017 (t-1), May 2017 (t-2), and July 2016 (t-12).
    '''

    #t-1 month
```

```
    t1 = df.rename(columns={'month':'pmonth', 'year':'pyear', 'unit_sales':'sale
s_t1_'+df_string})
    t1['month'] = np.where(t1['pmonth']==12, 1, t1['pmonth']+1)
    t1['year'] = np.where(t1['pmonth']==12, t1['pyear']+1 , t1['pyear'])
    t1 = t1.drop(['pmonth', 'pyear'], 1)

    #t-2 month
    t2 = df.rename(columns={'month':'pmonth', 'year':'pyear', 'unit_sales':'sale
s_t2_'+df_string})
    t2['month'] = np.where(t2['pmonth']>10, (t2['pmonth']+2)%12, t2['pmonth']+2)
    t2['year'] = np.where(t2['pmonth']>10, t2['pyear']+1 , t2['pyear'])
    t2 = t2.drop(['pmonth', 'pyear'], 1)

    #t-12 month
    t12 = df.rename(columns={'month':'pmonth', 'year':'pyear', 'unit_sales':'sal
es_t12_'+df_string})
    t12['month'] = t12['pmonth']
    t12['year'] = t12['pyear']+1
    t12 = t12.drop(['pmonth', 'pyear'], 1)

    merge_cols = ['month','year','cluster','class']
    if 'I' in df_string:
        merge_cols.append('item_nbr')
        merge_cols.remove('class')
    if 'S' in df_string:
        merge_cols.append('store_nbr')
        merge_cols.remove('cluster')
    if 'weekday' in df_string:
        merge_cols.append('weekday')

    final = t1.merge(t2, on=merge_cols, how='outer').merge(t12, on=merge_cols, h
ow='outer')
    for t in [1, 2, 12]:
        merge_cols.append('sales_t'+str(t)+'_'+df_string)

    final = final[merge_cols].reset_index(drop=True)

    #Ensure that values for most recent observations are non-empty
    final['sales_t1_'+df_string] = final['sales_t1_'+df_string].fillna(0)
    return final


def fill_blanks_new(DF, df_string):
    '''This function takes a data frame and fills missing observations in the au
toregressive
    feature columns. In particular, if an observation is missing, use front fill
, where the
    most recent observation is at the top of the list.

    Note that in the function above, we made it so that any empty cells for the
most recent
    observation were filled in as 0.
    '''
```

```
    transposed = DF.T

    transposed = transposed.fillna(method='ffill')
    new_DF = transposed.T
    return new_DF
```

Basic Features

In [3]:

```
'''
The Features dataset includes a combination of store-level and date-level featur
es (e.g. type of store,
oil price, whether the date was a holiday for the particular store).
'''
Features = pd.read_csv('Features.csv').drop('Unnamed: 0', 1)

#Main dataset spans from 1/1/13 to 8/31/17
Features = Features[(Features['date']>'2012-12-31')&(Features['date']<'2017-09-0
1')]
Features['date'] = pd.to_datetime(Features['date'])
Features = Features.dropna().reset_index(drop=True)
Features['store_nbr'] = Features['store_nbr'].astype(int)
Features.head()
```

Out[3]:

	date	oil_price	store_nbr	cluster	typeA	typeB	typeC	typeD	typeE	Local Holiday	Reg Ho
0	2013-01-01	93.14	25	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
1	2013-01-02	93.14	25	1.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
2	2013-01-02	93.14	1	13.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
3	2013-01-02	93.14	2	13.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0
4	2013-01-02	93.14	3	8.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0

In [4]:

```
#Bread.csv is a file that includes sales data for each store-date-item (bread pr
oduct) combination
#TransSI == transactions per store item

#Each row corresponds to the number of sales for a particular store-item-date co
mbination.

TransSI = pd.read_csv('./Datasets/Bread.csv').reset_index(drop=True).drop('Unnam
ed: 0',1)
TransSI['date'] = pd.to_datetime(TransSI['date'])
TransSI.head()
```

Out[4]:

	date	id	item_nbr	onpromotion	store_nbr	unit_sales	family	class
0	2013-01-01	0	103665	0	25	7.0	BREAD/BAKERY	2712
1	2013-01-02	578	103665	0	1	2.0	BREAD/BAKERY	2712
2	2013-01-02	1596	103665	0	2	5.0	BREAD/BAKERY	2712
3	2013-01-02	2699	103665	0	3	6.0	BREAD/BAKERY	2712
4	2013-01-02	3900	103665	0	4	2.0	BREAD/BAKERY	2712

In [5]:

```
#Merge TransSI table with Features table
TransSI = pd.merge(TransSI, Features, on=['store_nbr','date'], how='outer')
TransSI.head()
```

Out[5]:

	date	id	item_nbr	onpromotion	store_nbr	unit_sales	family	class	oil_
0	2013-01-01	0	103665	0	25	7.0	BREAD/BAKERY	2712	93.
1	2013-01-01	26	153239	0	25	3.0	BREAD/BAKERY	2712	93.
2	2013-01-01	28	153395	0	25	7.0	BREAD/BAKERY	2704	93.
3	2013-01-01	45	165718	0	25	2.0	BREAD/BAKERY	2718	93.
4	2013-01-01	66	215370	0	25	2.0	BREAD/BAKERY	2718	93.

In [6]:

```
#Create new variables based on the date column
TransSI['weekday'] = TransSI['date'].dt.weekday
TransSI['weekend'] = np.where(TransSI['weekday']>3, 1, 0) #Wkend=1 if weekday is
Fri, Sat, or Sun
TransSI['month'] = TransSI['date'].dt.month
TransSI['year'] = TransSI['date'].dt.year

#woy = "week of the year" -- so 1/5/2017 would be tagged as 1, 1/12/2017 tagged
as 2, etc.
TransSI['WOY'] = TransSI['date'].dt.week

#Minor cleaning issues
TransSI['woy'] = np.where((TransSI['WOY']==1) & (TransSI['month']==12), 52, Tran
sSI['WOY'])
TransSI['woy'] = np.where((TransSI['WOY']>51) & (TransSI['month']==1), 1, TransS
I['woy'])
TransSI['woy'] = np.where((TransSI['woy']==53), 52, TransSI['woy'] )
TransSI['woy'] = np.where((TransSI['date']>'2017-08-27')&(TransSI['date']<'2017-
09-01'),
                                                                  34, TransSI
['woy'])
TransSI = TransSI.drop('WOY', 1)
TransSI.head()
```

Out[6]:

	date	id	item_nbr	onpromotion	store_nbr	unit_sales	family	class	oil_
0	2013-01-01	0	103665	0	25	7.0	BREAD/BAKERY	2712	93.
1	2013-01-01	26	153239	0	25	3.0	BREAD/BAKERY	2712	93.
2	2013-01-01	28	153395	0	25	7.0	BREAD/BAKERY	2704	93.
3	2013-01-01	45	165718	0	25	2.0	BREAD/BAKERY	2718	93.
4	2013-01-01	66	215370	0	25	2.0	BREAD/BAKERY	2718	93.

Building Features from Past Data ("Autoregressive" Features)

In [7]:

```
'''
Autoregressive features will be split according to two criteria (hence 4 buckets
).
- weekday vs. smooth: Consider only observations with the same weekday? Or consi
der all days?
- woy vs. month: Data based on the previous week? Or the previous month?
'''

#SI_weekday -- unique row for each store-item-weekday-month combination
#For example, one row might be mean sales of cornbread at store #12 for Fridays
in July 2017.
SI_month_weekday = TransSI.groupby(['weekday','month','year','store_nbr','item_n
br']).mean() \
                            [['unit_sales']].reset_index()

#SI_smooth -- unique row for each store-item-month combination
#For example, one row might be mean sales of cornbread at store #12 for all days
in July 2017.
SI_month_smooth = TransSI.groupby(['month','year','store_nbr','item_nbr']).mean(
) \
                            [['unit_sales']].reset_index()

#SI_woy_weekday -- unique row for each store-item-date combination
#For example, one row might be sales of cornbread at store #12 for the Friday in
33rd week of 2017.
#SI_woy_weekday is essentially the same as TransSI
SI_woy_weekday = TransSI.groupby(['weekday','woy','year','store_nbr','item_nbr']
).mean() \
                                [['unit_sales']].reset_index()

#SI_woy_smooth -- unique row for each store-item-week combination
#For example, one row might be mean sales of cornbread at store #12 for all days
in 33rd week of 2017.
SI_woy_smooth = TransSI.groupby(['woy','year','store_nbr','item_nbr']).mean() \
                                [['unit_sales']].reset_index()

SI_woy_weekday.head()
```

Out[7]:

	weekday	woy	year	store_nbr	item_nbr	unit_sales
0	0	1	2016	1	153395	1.0
1	0	1	2016	1	153398	8.0
2	0	1	2016	1	165718	3.0
3	0	1	2016	1	215370	3.0
4	0	1	2016	1	265279	5.0

In [8]:

```
SI_month_smooth2 = add_features_from_previous_months(SI_month_smooth, 'SI_month_
smooth')
SI_month_weekday2 = add_features_from_previous_months(SI_month_weekday, 'SI_mont
h_weekday')

SI_woy_smooth2 = add_features_from_previous_weeks(SI_woy_smooth, 'SI_woy_smooth'
)
SI_woy_weekday2 = add_features_from_previous_weeks(SI_woy_weekday, 'SI_woy_weekd
ay')


#-----------
#sample sideshow

sample = SI_woy_weekday2[(SI_woy_weekday2['item_nbr']==153398)&
                (SI_woy_weekday2['store_nbr']==1)&(SI_woy_weekday2['year']==2017
)].head()

sample2 = sample.merge(TransSI, on=['item_nbr', 'weekday', 'store_nbr', 'year',
'woy'])
sample2 = sample2[['item_nbr', 'weekday', 'woy', 'year', 'store_nbr', 'unit_sale
s',
                   'sales_w2_SI_woy_weekday', 'sales_w3_SI_woy_weekday', 'sales_w
52_SI_woy_weekday'  ]]

#Note that sales_w2[i+2] = unit_sales[i] for all i, sales_w3[i+3] = unit_sales[i
]
sample2
```

Out[8]:

	item_nbr	weekday	woy	year	store_nbr	unit_sales	sales_w2_SI_woy_weekday	s
0	153398	0	3	2017	1	1.0	3.0	1
1	153398	0	4	2017	1	5.0	5.0	3
2	153398	0	5	2017	1	1.0	1.0	5
3	153398	0	6	2017	1	1.0	5.0	1
4	153398	0	7	2017	1	2.0	1.0	5

In [9]:

```
#Fill blank cells
SI_month_smooth_clean = fill_blanks_new(SI_month_smooth2, 'SI_month_smooth')
SI_month_weekday_clean = fill_blanks_new(SI_month_weekday2, 'SI_month_weekday')
SI_woy_smooth_clean = fill_blanks_new(SI_woy_smooth2, 'SI_woy_smooth')
SI_woy_weekday_clean = fill_blanks_new(SI_woy_weekday2, 'SI_woy_weekday')
```

Merging Step

Merge these four sets of autoregressive features with the primary dataset

In [10]:

```
DF = TransSI.merge(SI_month_smooth_clean, on=['month','year','store_nbr','item_n
br'], how='left')
DF = DF.merge(SI_month_weekday_clean, on=['month','year','store_nbr','item_nbr',
'weekday'], how='left')
DF = DF.merge(SI_woy_smooth_clean, on=['woy', 'year','store_nbr','item_nbr'], ho
w='left')
DF = DF.merge(SI_woy_weekday_clean, on=['woy','year','store_nbr','item_nbr','wee
kday'], how='left')

#Ignore first month of data -- no records for t-1 month, since dataset started o
n 1/1/2013
DF = DF[DF['date']>'2013-01-31']
```

In [11]:

```
DF.head()
```

Out[11]:

	date	id	item_nbr	onpromotion	store_nbr	unit_sales	family
42352	2013-02-01	1211964	103665	0	1	6.000	BREAD/BAKERY
42353	2013-02-01	1212076	215370	0	1	1.000	BREAD/BAKERY
42354	2013-02-01	1212157	310644	0	1	10.000	BREAD/BAKERY
42355	2013-02-01	1212160	311994	0	1	42.274	BREAD/BAKERY
42356	2013-02-01	1212182	315474	0	1	2.000	BREAD/BAKERY

Dealing with unknown test set entries

A large percentage of the test set observations (post 8/15/2017) do not have ANY historical records in the training set. One of the challenges of this competition is predicting sales when stores introduce new products.

The "autoregressive" features for these unknown observations are empty, which is a big problem. To fill these blank cells, I tried taking the mean of each column (month t-1, month t-2, month t-12, etc.) and then applied store-level weights and item-level weights. For example, let's say we are trying to populate cells for observations of Store #12's sales of baguettes, even though the store has never sold baguettes.

Let's say Store #12 is small, with total transaction volume (across all products) about 75% of the transaction volume of the average store. However, let's say on average that stores sell twice as many baguettes as the average item on a given day. We would then have Item Weight = 2 and Store Weight = 0.75. If the mean of the column "t-2" is 4.0 unit sales, we would fill in the blank cells of this observation as 4 times 2 times 0.75 = 6.0 unit sales.

In [12]:

```
def fill_missing_and_clean_months(df):
    '''This function handles missing test set values for the month-level autoreg
ressive
    features. It's useful for store-item observations in test set that NEVER app
ear in the
    training set.

    For example, for the column (t-1) representing sales for an item at a store
in month t-1,
    the function will first find the mean of the column. Then, it will fill in a
ny missing
    values with by multiplying the mean by a "store weight" (e.g. how busy the s
tore is)
    and an "item weight" (e.g., how much the item has sold at other stores). If
no "item weight"
    exists, use a "class weight" instead -- there are often multiple items per c
lass.
    Finally, if nothing else works, multiply the mean only by store weight.
    '''

    for s in ['_SI_month_smooth','_SI_month_weekday']:
        for n in ['t1', 't2', 't12']:
            mu = df['sales_'+n+s].mean()
            df['sales_'+n+s] = df['sales_'+n+s].fillna(mu*df['store_weight']*df[
'item_weight'])
            df['sales_'+n+s] = df['sales_'+n+s].fillna(mu*df['store_weight']*df[
'class_weight'])
            df['sales_'+n+s] = df['sales_'+n+s].fillna(mu*df['store_weight'])
    return df

def fill_missing_and_clean_weeks(df):
    '''Very similar to function above'''
    for s in ['_SI_woy_smooth','_SI_woy_weekday']:
        for n in ['w2', 'w3', 'w52']:
            mu = df['sales_'+n+s].mean()
            df['sales_'+n+s] = df['sales_'+n+s].fillna(mu*df['store_weight']*df[
'item_weight'])
            df['sales_'+n+s] = df['sales_'+n+s].fillna(mu*df['store_weight']*df[
'class_weight'])
            df['sales_'+n+s] = df['sales_'+n+s].fillna(mu*df['store_weight'])
    return df
```

Weights

In [13]:

```
#Transactions table gives total transactions per store per date
transactions = pd.read_csv('transactions.csv')
stores = pd.read_csv('stores.csv')

transactions['date'] = pd.to_datetime(transactions['date'])
transactions['weekday'] = transactions['date'].dt.weekday
transactions['month'] = transactions['date'].dt.month
transactions['year'] = transactions['date'].dt.year
TS = transactions.merge(stores, on='store_nbr')

#Store_weights_2017
TS17 = TS[TS['year']==2017]
s_weights17 = TS17.groupby(['store_nbr']).sum()[['transactions']].reset_index()
avg = s_weights17['transactions'].mean() #mean number of transactions across sto
res
s_weights17['store_weight'] = s_weights17['transactions']/avg
store_weights17 = s_weights17.drop('transactions', 1)

#Item weights
I = TransSI.groupby('item_nbr').sum()[['unit_sales']].reset_index()
Iavg = I['unit_sales'].mean() #mean number of sales among different items
I['item_weight'] = I['unit_sales']/Iavg
I = I.drop('unit_sales',1)
```

```
#Class weights
#These are for items that have NEVER appeared in ANY store in training data
C = TransSI.groupby('class').sum()[['unit_sales']].reset_index()
Cavg = C['unit_sales'].mean()
C['class_weight'] = C['unit_sales']/Cavg
C = C.drop(['unit_sales'], 1)
```

In [14]:

```
#Incorporate weights to main DF
DF = DF.merge(store_weights17, on=['store_nbr']). \
        merge(C, on='class'). \
        merge(I, on='item_nbr').sort_values(by='date')
```

In [15]:

```
#Split DF into Train, Validation, Test Sets
X_train_raw = DF[DF['date']<'2017-07-01']
X_valid_raw = DF[(DF['date']>'2017-06-30')&(DF['date']<'2017-08-16')]
X_test_raw = DF[(DF['date']>'2017-08-15')&(DF['date']<'2017-09-01')]

#ids relevant for Kaggle submission
ids_test = X_test_raw['id']
```

In [47]:

```
# Deal with unknowns in test set
X_train = fill_missing_and_clean_months(X_train_raw)
X_valid = fill_missing_and_clean_months(X_valid_raw)
X_test = fill_missing_and_clean_months(X_test_raw)

X_train = fill_missing_and_clean_weeks(X_train)
X_valid = fill_missing_and_clean_weeks(X_valid)
X_test = fill_missing_and_clean_weeks(X_test)
```

```
/Users/Brenton/anaconda/lib/python3.5/site-packages/ipykernel/__main
__.py:17: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/panda
s-docs/stable/indexing.html#indexing-view-versus-copy
/Users/Brenton/anaconda/lib/python3.5/site-packages/ipykernel/__main
__.py:18: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/panda
s-docs/stable/indexing.html#indexing-view-versus-copy
/Users/Brenton/anaconda/lib/python3.5/site-packages/ipykernel/__main
__.py:19: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/panda
s-docs/stable/indexing.html#indexing-view-versus-copy
/Users/Brenton/anaconda/lib/python3.5/site-packages/ipykernel/__main
__.py:27: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/panda
s-docs/stable/indexing.html#indexing-view-versus-copy
/Users/Brenton/anaconda/lib/python3.5/site-packages/ipykernel/__main
__.py:28: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/panda
s-docs/stable/indexing.html#indexing-view-versus-copy
/Users/Brenton/anaconda/lib/python3.5/site-packages/ipykernel/__main
__.py:29: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: http://pandas.pydata.org/panda
s-docs/stable/indexing.html#indexing-view-versus-copy
```

In [48]:

```
X_train = X_train.dropna().reset_index(drop=True)

y_train = X_train['unit_sales']
y_valid = X_valid['unit_sales']

#Correct errors (negative values) in y_train, y_valid -- just set to 0
y_train = np.where(y_train<0, 0, y_train)
y_valid = np.where(y_valid<0, 0, y_valid)

#Remove unnecessary columns
X_train = X_train.drop(['unit_sales','date','id', 'woy', 'store_weight', 'item_w
eight', 'class_weight',
                        'store_nbr', 'cluster','month','class', 'family', 'item_
nbr', 'weekday'],1)
X_valid = X_valid.drop(['unit_sales','date','id', 'woy', 'store_weight', 'item_w
eight', 'class_weight',
                        'store_nbr', 'cluster','month','class','family', 'item_n
br', 'weekday'],1)
X_test = X_test.drop(['unit_sales','date','id', 'woy', 'store_weight', 'item_wei
ght', 'class_weight',
                        'store_nbr', 'cluster','month','class','family', 'item_n
br', 'weekday'],1)
```

Modeling - Linear Regression

In [49]:

```
LM = linear_model.LinearRegression()
LM.fit(X_train, y_train)
```

Out[49]:

```
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=1, normaliz
e=False)
```

In [52]:

```
prediction = LM.predict(X_valid)
pred = np.where(prediction<0, 0, prediction)
print('RMSLE: ', round(loss_function(pred, y_valid), 3))
print('R-squared: ', round(LM.score(X_valid, y_valid),3))
```

```
RMSLE:  0.525
R-squared:  0.681
```

Modeling - Decision Trees

In [70]:

```
depths = []
min_samples = []
losses = []
R2s = []

for d in [3, 5, 10, 20]:
    for m in [1, 10, 50, 100]:
        DT = tree.DecisionTreeRegressor(max_depth=d, min_samples_leaf=m)
        DT.fit(X_train, y_train)
        prediction = DT.predict(X_valid)
        pred = np.where(prediction<0, 0, prediction)
        loss = loss_function(pred, y_valid)
        R2 = DT.score(X_valid, y_valid)

        depths.append(d)
        min_samples.append(m)
        losses.append(round(loss, 3))
        R2s.append(round(R2, 3))

DT_results = pd.DataFrame(np.vstack((depths, min_samples, losses, R2s))).T
DT_results = DT_results.rename(columns={0: 'Max Depth', 1: 'Min Leaf Samples',
                                        2: 'RMSLE (Loss)', 3: 'R-Squared' })
```

In [71]:

```
#Unfortunately, not a major improvement over Linear Regression model
DT_results.sort_values(by='RMSLE (Loss)')
```

Out[71]:

	Max Depth	Min Leaf Samples	RMSLE (Loss)	R-Squared
15	20.0	100.0	0.523	0.678
14	20.0	50.0	0.525	0.671
9	10.0	10.0	0.527	0.675
10	10.0	50.0	0.527	0.678
11	10.0	100.0	0.527	0.680
8	10.0	1.0	0.528	0.667
13	20.0	10.0	0.533	0.642
12	20.0	1.0	0.537	0.559
4	5.0	1.0	0.547	0.659
5	5.0	10.0	0.547	0.659
6	5.0	50.0	0.547	0.659
7	5.0	100.0	0.547	0.659
0	3.0	1.0	0.581	0.628
1	3.0	10.0	0.581	0.628
2	3.0	50.0	0.581	0.628
3	3.0	100.0	0.581	0.628

In [73]:

```
#no pruning - terrible performance
DT = tree.DecisionTreeRegressor()
DT.fit(X_train, y_train)
prediction = DT.predict(X_valid)
pred = np.where(prediction<0, 0, prediction)
loss = loss_function(pred, y_valid)
print(round(loss,3))
```

```
0.715
```

Modeling - Random Forests

In [78]:

```
estimators = []
losses = []
R2s = []

for n in [10, 20, 40]:
    RF = ensemble.RandomForestRegressor(n_estimators=n)
    RF.fit(X_train, y_train)
    prediction = RF.predict(X_valid)
    pred = np.where(prediction<0, 0, prediction)
    loss = loss_function(pred, y_valid)
    R2 = RF.score(X_valid, y_valid)

    estimators.append(n)
    losses.append(round(loss, 3))
    R2s.append(round(R2, 3))

RF_results = pd.DataFrame(np.vstack((estimators, losses, R2s))).T
RF_results = RF_results.rename(columns={0: 'Trees in Forest', 1: 'RMSLE (Loss)',
2: 'R-Squared' })
RF_results
```

Out[78]:

	Trees in Forest	RMSLE (Loss)	R-Squared
0	10.0	0.547	0.660
1	20.0	0.536	0.674
2	40.0	0.530	0.679

Surprisingly, no significant improvement with Random Forests

With more time, would have tried other models (e.g. gradient boosting, adaboost, etc.)

Ultimately, based on these results, we might just want to use Linear Regression for all items.

Feature Importances (DT)

Strongest feature is the store's "average sales of item in relevant weekday" in previous month. So if we want to predict sales of baguettes on a Friday in November 2016, an important feature would be average sales of baguettes on Fridays in October 2016.

With more time, I would have added new features for weather conditions (precipitation, temperature). I would have also considered adding features for the sales of similar products/substitutes. For example, to predict the sales of hot dog buns, I could have used historical data on unit sales of frankfurters, ketchup and mustard.

In [81]:

```
features = X_train.columns.values
DT_best = tree.DecisionTreeRegressor(max_depth=20, min_samples_leaf=100)
DT_best.fit(X_train, y_train)
importances = DT_best.feature_importances_

featuresDF = pd.DataFrame(np.vstack((features, importances))).T.rename(columns={
0:'Feature',
                                                                        1
: 'Importance'})
featuresDF = featuresDF.sort_values(by='Importance', ascending=False)
featuresDF
```

Out[81]:

	Feature	Importance
15	sales_t1_SI_month_weekday	0.758498
12	sales_t1_SI_month_smooth	0.132719
16	sales_t2_SI_month_weekday	0.0529334
18	sales_w2_SI_woy_smooth	0.0181032
17	sales_t12_SI_month_weekday	0.01025
21	sales_w2_SI_woy_weekday	0.00633122
23	sales_w52_SI_woy_weekday	0.0037135
20	sales_w52_SI_woy_smooth	0.00370136
14	sales_t12_SI_month_smooth	0.00299326
19	sales_w3_SI_woy_smooth	0.00186678
1	oil_price	0.00184837
13	sales_t2_SI_month_smooth	0.00178973
0	onpromotion	0.00161816
10	weekend	0.00113377
9	National Holiday	0.000767761
22	sales_w3_SI_woy_weekday	0.000739184
2	typeA	0.000455406
11	year	0.000273313
5	typeD	0.000143756
3	typeB	5.42768e-05
4	typeC	5.17265e-05
6	typeE	1.417e-05
8	Regional Holiday	0
7	Local Holiday	0

Save Test Predictions

In [53]:

```
#Linear Model
LM_test_prediction = LM.predict(X_test)
LM_test_pred = np.where(LM_test_prediction<0, 0, LM_test_prediction)
LM_final = pd.DataFrame(np.vstack((ids_test, LM_test_pred))).T.rename(columns={0
:'id',1:'unit_sales'})
LM_final.head()

print('Each id corresponds to a unique store-item-date combination')
```

Out[53]:

	id	unit_sales
0	125579031.0	0.589816
1	125572169.0	5.995403
2	125526297.0	2.028140
3	125610821.0	8.421807
4	125557760.0	0.164480

In [82]:

```
#Decision Trees

#Choose best model based on validation set performance
DT_best = tree.DecisionTreeRegressor(max_depth=20, min_samples_leaf=100)
DT_best.fit(X_train, y_train)

DT_test_prediction = DT_best.predict(X_test)
DT_test_pred = np.where(DT_test_prediction<0, 0, DT_test_prediction)
DT_final = pd.DataFrame(np.vstack((ids_test, DT_test_pred))).T.rename(columns={0
:'id',1:'unit_sales'})
DT_final.head()
```

Out[82]:

	id	unit_sales
0	125579031.0	1.657063
1	125572169.0	4.184971
2	125526297.0	2.383112
3	125610821.0	9.600000
4	125557760.0	4.032680

In [83]:

```
#Tempted to average the predictions from Linear Regression/DT, but for now keepi
ng them separate.
category = 'Bread'
final.to_csv('Predictions/'+category+'_LM_preds.csv')
final.to_csv('Predictions/'+category+'_DT_preds.csv')
```