Stat157_homework4_solution

1 Homework 4

In this homework, we will build a model based real house sale data from a Kaggle competition. This notebook contains codes to download the dataset, build and train a baseline model, and save the results in the submission format. Your jobs are

1. Developing a better model to reduce the prediction error. You can find some hints on the last section.
2. Submitting your results into Kaggle and take a sceenshot of your score. Then replace the following image URL with your screenshot.

We have two suggestions for this homework:

1. Start as earlier as possible. Though we will cover this notebook on Thursday's lecture, tuning hyper-parameters takes time, and Kaggle limits #submissions per day.
2. Work with your project teammates. It's a good opportunity to get familiar with each other.

Your scores will depend your positions on Kaggle's Leaderboard. We will award the top-3 teams/individuals 500 AWS credits.

1.1 Accessing and Reading Data Sets

The competition data is separated into training and test sets. Each record includes the property values of the house and attributes such as street type, year of construction, roof type, basement condition. The data includes multiple datatypes, including integers (year of construction), discrete labels (roof type), floating point numbers, etc.; Some data is missing and is thus labeled 'na'. The price of each house, namely the label, is only included in the training data set (it's a competition after all). The 'Data' tab on the competition tab has links to download the data.

We will read and process the data using `pandas`, an efficient data analysis toolkit. Make sure you have `pandas` installed for the experiments in this section.

```
In [2]: import numpy as np
        import pandas as pd
        pd.set_option('display.float_format', lambda x: '{:.3f}'.format(x)) #Limiting floats o

        %matplotlib inline
        import matplotlib.pyplot as plt  # Matlab-style plotting
```

```python
import seaborn as sns
color = sns.color_palette()
sns.set_style('darkgrid')

import warnings
def ignore_warn(*args, **kwargs):
    pass
warnings.warn = ignore_warn #ignore annoying warning (from sklearn and seaborn)


from scipy import stats
from scipy.stats import norm, skew #for some statistics


import d2l
from mxnet import autograd, gluon, init, nd
from mxnet.gluon import data as gdata, loss as gloss, nn, utils
from mxnet import ndarray as nd
from mxnet import autograd
from mxnet import gluon
import mxnet as mx
ctx = mx.gpu()
print(ctx)

mx.Context.default_ctx = mx.gpu(0)
```

```
gpu(0)
```

1.2 Step I : Load train/test data

In [3]:
```python
# utils.download('https://github.com/d2l-ai/d2l-en/raw/master/data/kaggle_house_pred_t
# utils.download('https://github.com/d2l-ai/d2l-en/raw/master/data/kaggle_house_pred_t

train = pd.read_csv('kaggle_house_pred_train.csv')
test = pd.read_csv('kaggle_house_pred_test.csv')
test_ID = test['Id']

train.head(5)
```

Out[3]:
```
   Id  MSSubClass MSZoning  LotFrontage  LotArea Street Alley LotShape  \
0   1          60       RL       65.000     8450   Pave   NaN      Reg
1   2          20       RL       80.000     9600   Pave   NaN      Reg
2   3          60       RL       68.000    11250   Pave   NaN      IR1
3   4          70       RL       60.000     9550   Pave   NaN      IR1
4   5          60       RL       84.000    14260   Pave   NaN      IR1

  LandContour Utilities    ...     PoolArea PoolQC Fence MiscFeature MiscVal  \
```

```
0          Lvl     AllPub    ...           0     NaN    NaN          NaN        0
1          Lvl     AllPub    ...           0     NaN    NaN          NaN        0
2          Lvl     AllPub    ...           0     NaN    NaN          NaN        0
3          Lvl     AllPub    ...           0     NaN    NaN          NaN        0
4          Lvl     AllPub    ...           0     NaN    NaN          NaN        0

  MoSold YrSold  SaleType  SaleCondition  SalePrice
0      2   2008        WD         Normal     208500
1      5   2007        WD         Normal     181500
2      9   2008        WD         Normal     223500
3      2   2006        WD        Abnorml     140000
4     12   2008        WD         Normal     250000

[5 rows x 81 columns]
```

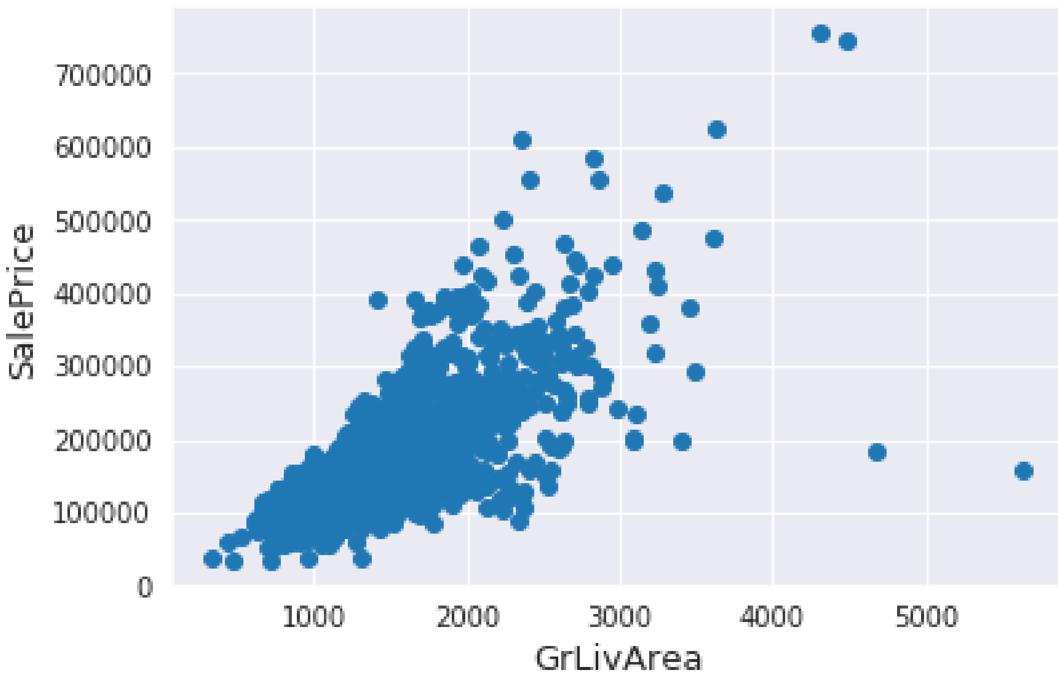
In [4]:
```
## Drop the  'Id' colum since it's unnecessary for  the prediction process.
train.drop("Id", axis = 1, inplace = True)
test.drop("Id", axis = 1, inplace = True)

## check again the data size after dropping the 'Id' variable
print("\nThe train data size after dropping Id feature is : {} ".format(train.shape))
print("The test data size after dropping Id feature is : {} ".format(test.shape))
```

```
The train data size after dropping Id feature is : (1460, 80)
The test data size after dropping Id feature is : (1459, 79)
```

2 Step II : Data Processing

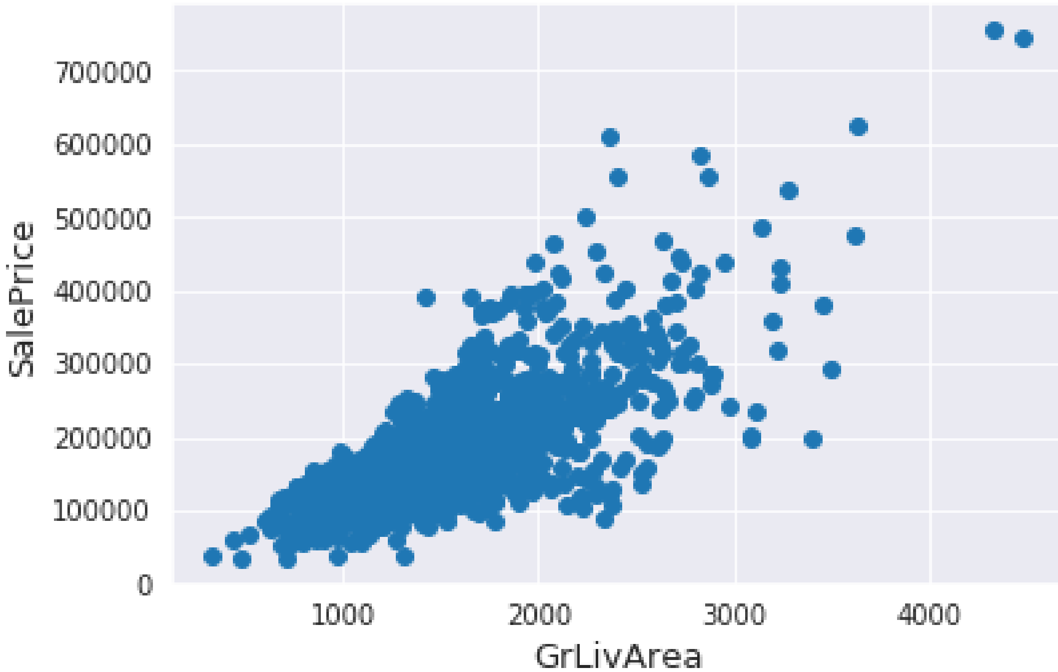
2.1 Outliers

In [5]:
```
fig, ax = plt.subplots()
ax.scatter(x = train['GrLivArea'], y = train['SalePrice'])
plt.ylabel('SalePrice', fontsize=13)
plt.xlabel('GrLivArea', fontsize=13)
plt.show()
```



We can see at the bottom right two with extremely large GrLivArea that are of a low price. These values are huge oultliers. Therefore, we can safely delete them.

```
In [6]: #Deleting outliers
        train = train.drop(train[(train['GrLivArea']>4000) & (train['SalePrice']<300000)].inde

        #Check the graphic again
        fig, ax = plt.subplots()
        ax.scatter(train['GrLivArea'], train['SalePrice'])
        plt.ylabel('SalePrice', fontsize=13)
        plt.xlabel('GrLivArea', fontsize=13)
        plt.show()
```



Note : Outliers removal is note always safe. We decided to delete these two as they are very huge and really bad (extremely large areas for very low prices).

There are probably others outliers in the training data. However, removing all them may affect badly our models if ever there were also outliers in the test data. That's why , instead of removing them all, we will just manage to make some of our models robust on them. You can refer to the modelling part of this notebook for that.

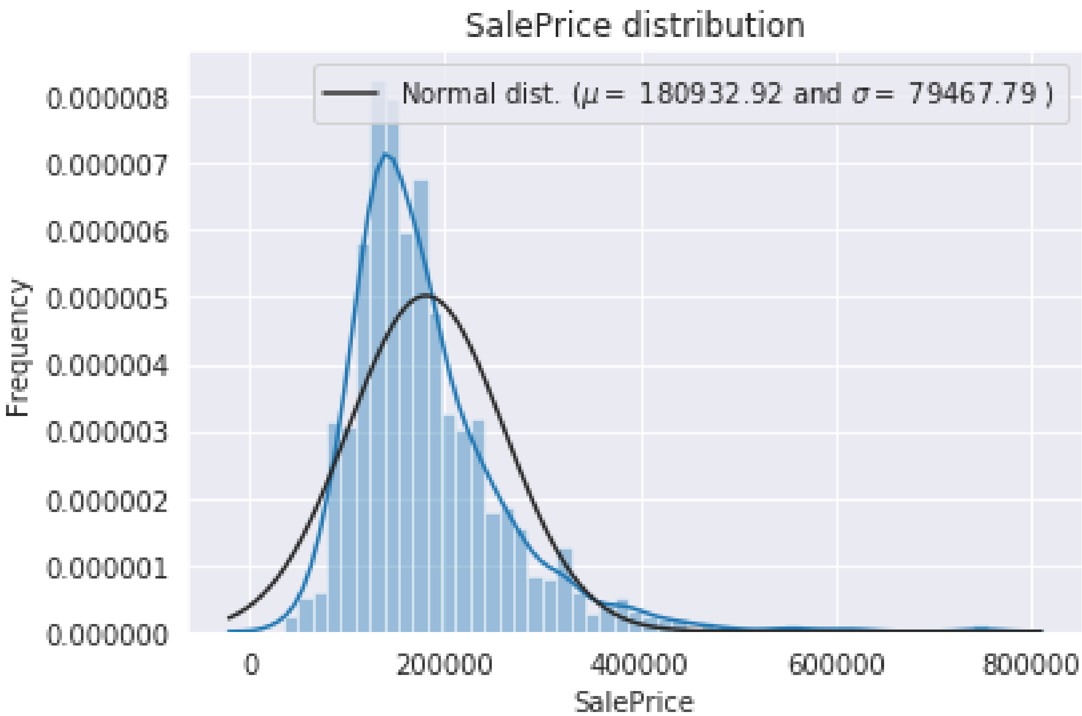
2.1.1 Target Variable

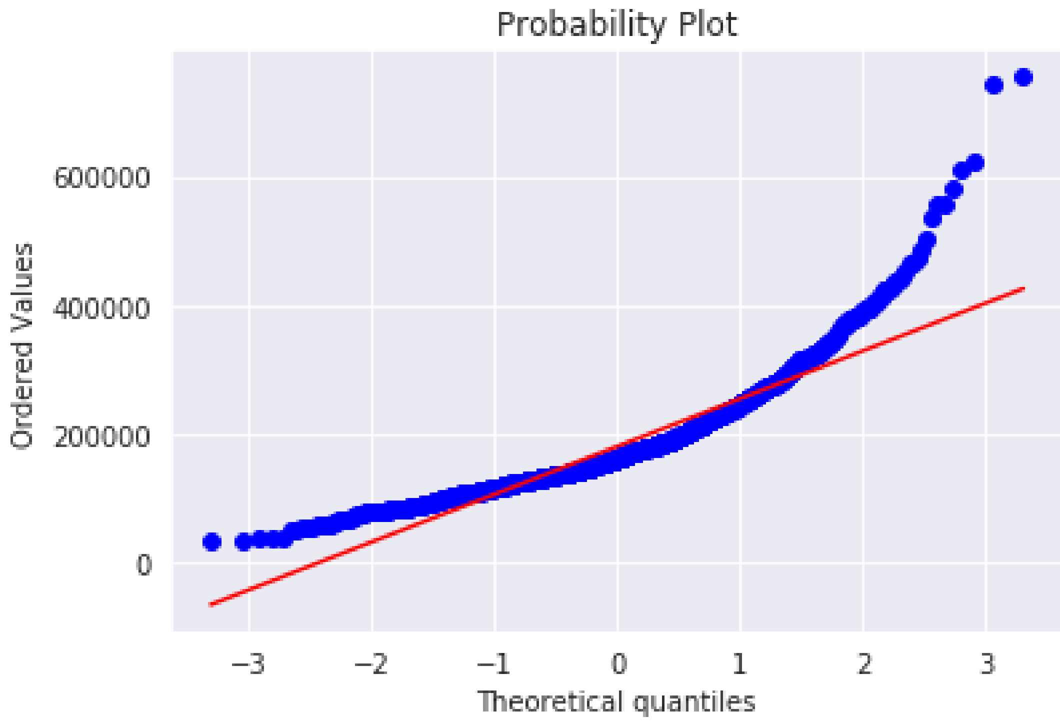
SalePrice is the variable we need to predict. So let's do some analysis on this variable first.

```
In [7]: sns.distplot(train['SalePrice'] , fit=norm);

        # Get the fitted parameters used by the function
        (mu, sigma) = norm.fit(train['SalePrice'])
        print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))

        #Now plot the distribution
        plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu, sigma)],
                    loc='best')
        plt.ylabel('Frequency')
        plt.title('SalePrice distribution')

        #Get also the QQ-plot
```

```
fig = plt.figure()
res = stats.probplot(train['SalePrice'], plot=plt)
plt.show()
```

```
mu = 180932.92 and sigma = 79467.79
```



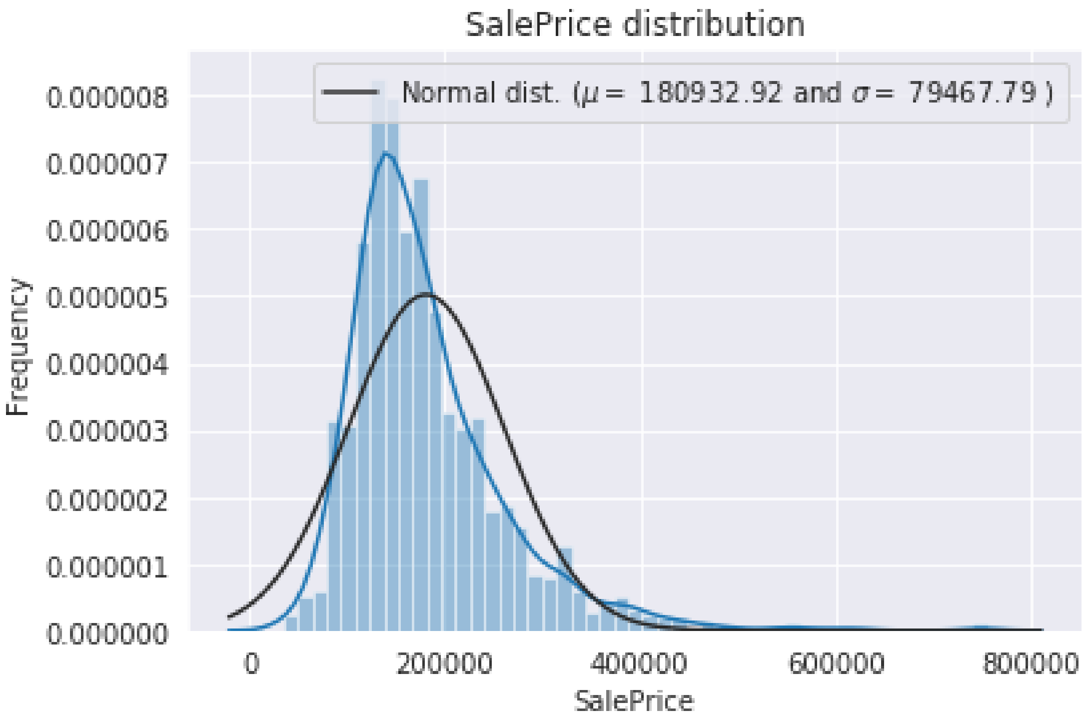


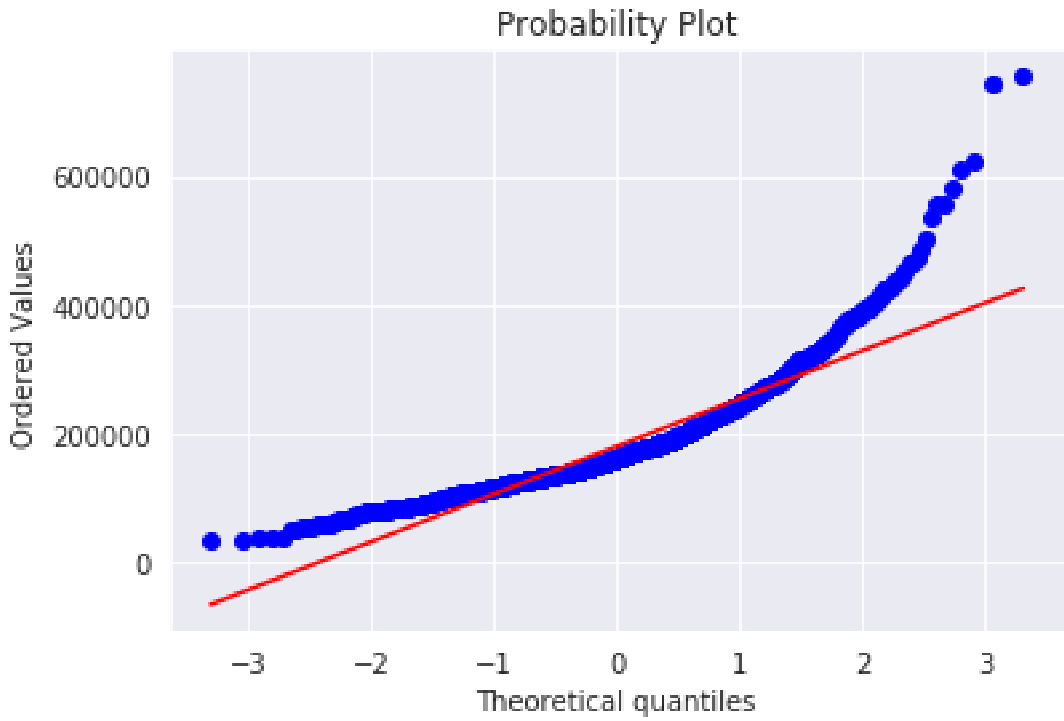
The target variable is right skewed. As (linear) models love normally distributed data , we need to transform this variable and make it more normally distributed.

Log-transformation of the target variable

```
In [8]: #We use the numpy fuction log1p which  applies log(1+x) to all elements of the column
        # train["SalePrice"] = np.log1p(train["SalePrice"])

        #Check the new distribution
        sns.distplot(train['SalePrice'] , fit=norm);

        # Get the fitted parameters used by the function
        (mu, sigma) = norm.fit(train['SalePrice'])
        print( '\n mu = {:.2f} and sigma = {:.2f}\n'.format(mu, sigma))

        #Now plot the distribution
        plt.legend(['Normal dist. ($\mu=$ {:.2f} and $\sigma=$ {:.2f} )'.format(mu, sigma)],
                    loc='best')
        plt.ylabel('Frequency')
        plt.title('SalePrice distribution')

        #Get also the QQ-plot
        fig = plt.figure()
        res = stats.probplot(train['SalePrice'], plot=plt)
        plt.show()
```

```
mu = 180932.92 and sigma = 79467.79
```





The skew seems now corrected and the data appears more normally distributed.

2.2 Features engineering

let's first concatenate the train and test data in the same dataframe

In [9]:
```
ntrain = train.shape[0]
ntest = test.shape[0]
y_train = train.SalePrice.values
all_data = pd.concat((train, test)).reset_index(drop=True)
all_data.drop(['SalePrice'], axis=1, inplace=True)
print("all_data size is : {}".format(all_data.shape))
```

```
all_data size is : (2917, 79)
```

2.2.1 Missing Data

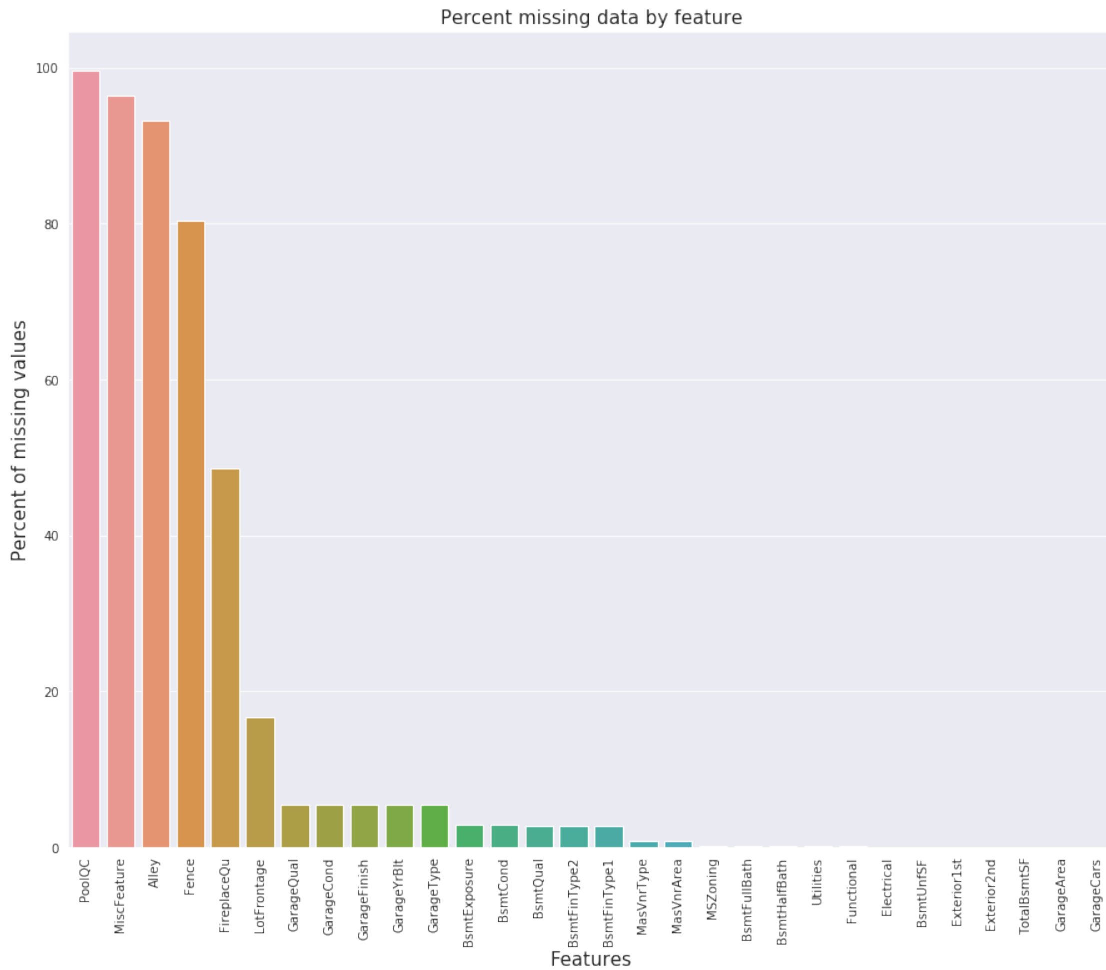
In [10]:
```
all_data_na = (all_data.isnull().sum() / len(all_data)) * 100
all_data_na = all_data_na.drop(all_data_na[all_data_na == 0].index).sort_values(ascend
missing_data = pd.DataFrame({'Missing Ratio' :all_data_na})
missing_data.head(20)
```

Out[10]:
```
                Missing Ratio
PoolQC                 99.691
```

```
MiscFeature         96.400
Alley               93.212
Fence               80.425
FireplaceQu         48.680
LotFrontage         16.661
GarageQual           5.451
GarageCond           5.451
GarageFinish         5.451
GarageYrBlt          5.451
GarageType           5.382
BsmtExposure         2.811
BsmtCond             2.811
BsmtQual             2.777
BsmtFinType2         2.743
BsmtFinType1         2.708
MasVnrType           0.823
MasVnrArea           0.788
MSZoning             0.137
BsmtFullBath         0.069
```

```
In [11]: f, ax = plt.subplots(figsize=(15, 12))
         plt.xticks(rotation='90')
         sns.barplot(x=all_data_na.index, y=all_data_na)
         plt.xlabel('Features', fontsize=15)
         plt.ylabel('Percent of missing values', fontsize=15)
         plt.title('Percent missing data by feature', fontsize=15)
```

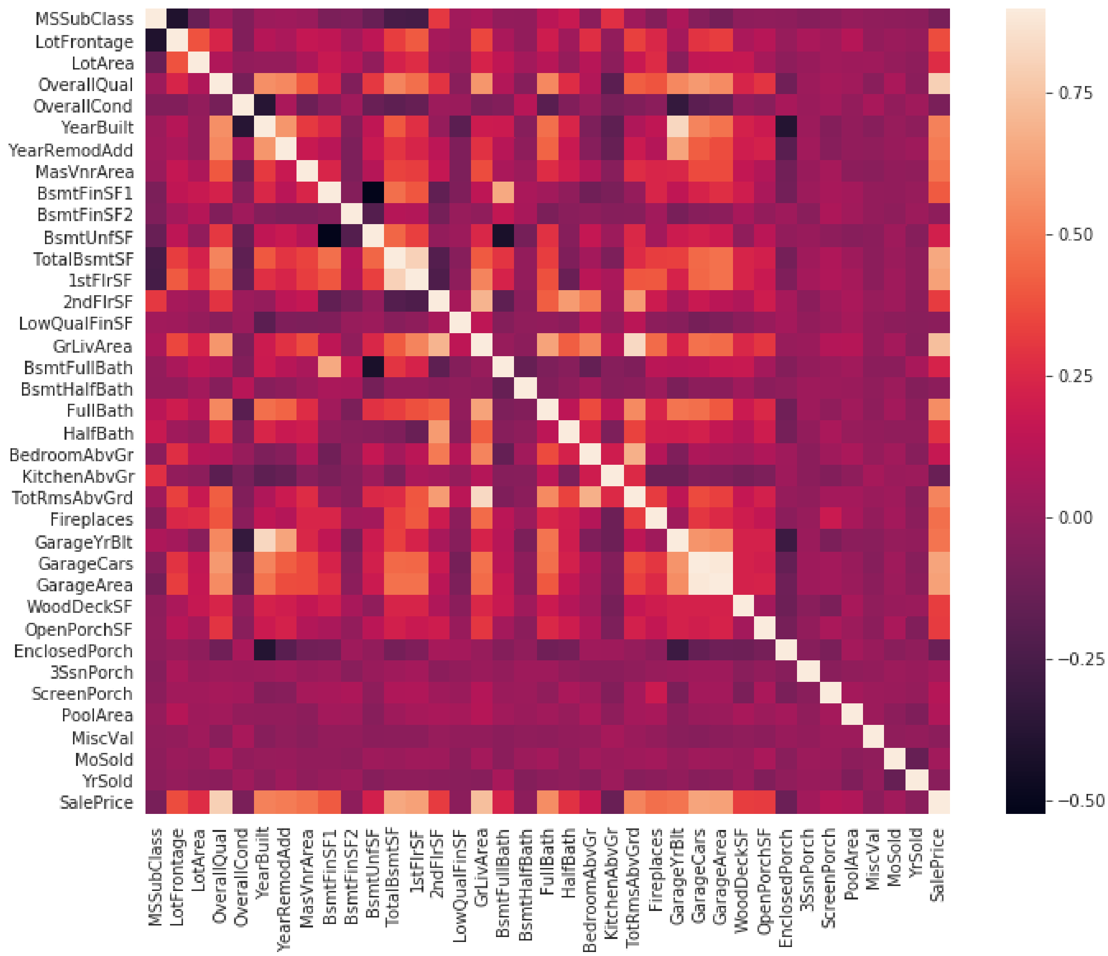
```
Out[11]: Text(0.5,1,'Percent missing data by feature')
```



Data Correlation

In [12]:
```
#Correlation map to see how features are correlated with each other
corrmat = train.corr()
plt.subplots(figsize=(12,9))
sns.heatmap(corrmat, vmax=0.9, square=True)
```

Out[12]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7fef4428ae80>
```



2.2.2 Imputing missing values

We impute them by proceeding sequentially through features with missing values

- **LotFrontage** : Since the area of each street connected to the house property most likely have a similar area to other houses in its neighborhood , we can **fill in missing values by the median LotFrontage of the neighborhood**.

```
In [13]: #Group by neighborhood and fill in missing value by the median LotFrontage of all the
         all_data["LotFrontage"] = all_data.groupby("Neighborhood")["LotFrontage"].transform(
                                                         lambda x: x.fillna(x.median()))
```

- **GarageYrBlt, GarageArea and GarageCars** : Replacing missing data with 0 (Since No garage = no cars in such garage.)

```
In [14]: for col in ('GarageYrBlt', 'GarageArea', 'GarageCars'):
             all_data[col] = all_data[col].fillna(0)
```

- **BsmtFinSF1, BsmtFinSF2, BsmtUnfSF, TotalBsmtSF, BsmtFullBath and BsmtHalfBath** : missing values are likely zero for having no basement

```
In [15]: for col in ('BsmtFinSF1', 'BsmtFinSF2', 'BsmtUnfSF','TotalBsmtSF', 'BsmtFullBath', 'Bs
             all_data[col] = all_data[col].fillna(0)
```

- **MasVnrArea** : NA most likely means no masonry veneer for these houses. We can fill 0 for the area.

```
In [16]: all_data["MasVnrArea"] = all_data["MasVnrArea"].fillna(0)
```

- **Functional** : data description says NA means typical

```
In [17]: all_data["Functional"] = all_data["Functional"].fillna("Typ")
```

- **Electrical, Exterior1st, Exterior2nd & KitchenQual** : only has one NA value, we can set that for the missing value.

```
In [18]: all_data['Electrical'] = all_data['Electrical'].fillna(all_data['Electrical'].mode()[0
         all_data['KitchenQual'] = all_data['KitchenQual'].fillna(all_data['KitchenQual'].mode
         all_data['Exterior1st'] = all_data['Exterior1st'].fillna(all_data['Exterior1st'].mode
         all_data['Exterior2nd'] = all_data['Exterior2nd'].fillna(all_data['Exterior2nd'].mode
```

Transforming some numerical variables that are really categorical

```
In [19]: #MSSubClass=The building class
         all_data['MSSubClass'] = all_data['MSSubClass'].apply(str)


         #Changing OverallCond into a categorical variable
         all_data['OverallCond'] = all_data['OverallCond'].astype(str)


         #Year and month sold are transformed into categorical features.
         all_data['YrSold'] = all_data['YrSold'].astype(str)
         all_data['MoSold'] = all_data['MoSold'].astype(str)
```

Label Encoding some categorical variables that may contain information in their ordering set

```
In [20]: from sklearn.preprocessing import LabelEncoder
         cols = ('FireplaceQu', 'BsmtQual', 'BsmtCond', 'GarageQual', 'GarageCond',
                 'ExterQual', 'ExterCond','HeatingQC', 'PoolQC', 'KitchenQual', 'BsmtFinType1'
                 'BsmtFinType2', 'Functional', 'Fence', 'BsmtExposure', 'GarageFinish', 'LandSl
                 'LotShape', 'PavedDrive', 'Street', 'Alley', 'CentralAir', 'MSSubClass', 'Over
                 'YrSold', 'MoSold')
         # process columns, apply LabelEncoder to categorical features
         for c in cols:
             lbl = LabelEncoder()
             lbl.fit(list(all_data[c].values))
```

```
                all_data[c] = lbl.transform(list(all_data[c].values))

        # shape
        print('Shape all_data: {}'.format(all_data.shape))
```

```
Shape all_data: (2917, 79)
```

2.3 Normalizing

```
In [21]: numeric_features = all_data.dtypes[all_data.dtypes != 'object'].index
         all_data[numeric_features] = all_data[numeric_features].apply(
             lambda x: (x - x.mean()) / (x.std()))
         # after standardizing the data all means vanish, hence we can set missing values to 0
         all_data = all_data.fillna(0)
```

Getting dummy categorical features

Next we deal with discrete values. This includes variables such as 'MSZoning'. We replace them by a one-hot encoding in the same manner as how we transformed multiclass classification data into a vector of 0 and 1. For instance, 'MSZoning' assumes the values 'RL' and 'RM'. They map into vectors $(1, 0)$ and $(0, 1)$ respectively. Pandas does this automatically for us.

```
In [22]: # Dummy_na=True refers to a missing value being a legal eigenvalue, and creates an in
         all_data = pd.get_dummies(all_data, dummy_na=True)
         print(all_data.shape)
```

```
(2917, 246)
```

3 Modelling

```
In [23]: # all_features = all_data.iloc[:,:-1]
         train_features = nd.array(all_data[:ntrain].values)
         test_features = nd.array(all_data[ntrain:].values)
         train_labels = nd.array(y_train) #[:,-1] #= nd.array(train)
         train_features.shape
```

```
Out[23]: (1458, 246)
```

log RMSE House prices, like shares, are relative. That is, we probably care more about the relative error $\frac{y-\hat{y}}{y}$ than about the absolute error. For instance, getting a house price wrong by USD 100,000 is terrible in Rural Ohio, where the value of the house is USD 125,000. On the other hand, if we err by this amount in Los Altos Hills, California, we can be proud of the accuracy of our model (the median house price there exceeds 4 million).

One way to address this problem is to measure the discrepancy in the logarithm of the price estimates. In fact, this is also the error that is being used to measure the quality in this competition. After all, a small value δ of $\log y - \log \hat{y}$ translates into $e^{-\delta} \leq \frac{\hat{y}}{y} \leq e^{\delta}$. This leads to the following loss function:

$$L = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\log y_i - \log \hat{y}_i\right)^2}$$

In [24]:
```
loss = gloss.L2Loss()

def log_rmse(net, features, labels):
    # To further stabilize the value when the logarithm is taken, set the value less
    out = net(features)
    out = out.reshape(out.shape[0])
    clipped_preds = nd.clip(out, 1, float('inf'))
    rmse = nd.sqrt(2 * loss(clipped_preds.log(), labels.log()).mean())
    return rmse.asscalar()

def get_batchnorm_net(dropout=0.0):  ## layer_unit=64,
    net = nn.Sequential()
    net.add(gluon.nn.BatchNorm(axis=1, center=True, scale=True))
    net.add(nn.Dense(256, activation="relu"))
    net.add(nn.Dropout(dropout))
    net.add(nn.Dense(64, activation="relu"))
#     net.add(nn.Dropout(dropout))
    net.add(nn.Dense(1))
    net.initialize(force_reinit=True, ctx=mx.gpu(0), init=init.Xavier(),) #  epoch.en

    return net

def get_net(dropout=0.0):
    net = nn.Sequential()
    net.add(nn.Dense(256, activation="relu"))
    net.add(nn.Dropout(dropout))
    net.add(nn.Dense(64, activation="relu"))
#     net.add(nn.Dropout(dropout))
    net.add(nn.Dense(1))
    net.initialize(force_reinit=True, ctx=mx.gpu(0), init=init.Xavier(),) #  epoch.en

    return net

def train(net, train_features, train_labels, test_features, test_labels,
          num_epochs, learning_rate, weight_decay, batch_size):
    best_val = float("Inf")
    train_ls, test_ls = [], []
    train_iter = gdata.DataLoader(gdata.ArrayDataset(
        train_features, train_labels), batch_size, shuffle=True)
    # The Adam optimization algorithm is used here.
    trainer = gluon.Trainer(net.collect_params(),  # reset_ctx(ctx)
                            'adam', {'learning_rate': learning_rate, 'wd': weight_deca
    for epoch in range(num_epochs):
        for X, y in train_iter:
```

```
            X, y = X.as_in_context(ctx), y.as_in_context(ctx)
            with autograd.record():
                with autograd.train_mode():
                    l = loss(net(X), y)
            l.backward()
            trainer.step(batch_size)
        temp_train_ls = log_rmse(net, train_features, train_labels)
        train_ls.append(temp_train_ls)
        if test_labels is not None:
            temp_test_ls = log_rmse(net, test_features, test_labels)
            test_ls.append(temp_test_ls)

    return train_ls, test_ls

def get_k_fold_data(k, i, X, y):
    assert k > 1
    fold_size = X.shape[0] // k
    X_train, y_train = None, None
    for j in range(k):
        idx = slice(j * fold_size, (j + 1) * fold_size)
        X_part, y_part = X[idx, :], y[idx]
        if j == i:
            X_valid, y_valid = X_part, y_part
        elif X_train is None:
            X_train, y_train = X_part, y_part
        else:
            X_train = nd.concat(X_train, X_part, dim=0)
            y_train = nd.concat(y_train, y_part, dim=0)
    return X_train, y_train, X_valid, y_valid

def k_fold(k, X_train, y_train, test_features,
           num_epochs, dropout,
           learning_rate, weight_decay, batch_size):

    folds = list(range(k))
    pred_df = pd.DataFrame(columns=['Id']+folds)
    pred_df['Id'] = np.asarray(test_ID)
#     print(X_train.shape, test_features.shape, pred_df.shape, len(test_ID))
    train_l_sum, valid_l_sum = 0, 0
    for i in range(k):
        X_train_temp, y_train_temp, X_valid, y_valid = get_k_fold_data(k, i, X_train,
        X_train_temp = X_train_temp.reshape(X_train_temp.shape[0], 1, X_train_temp.sha
        X_valid = X_valid.reshape(X_valid.shape[0], 1, X_valid.shape[-1])
        net = get_batchnorm_net(dropout)
        train_ls, valid_ls = train(net, X_train_temp, y_train_temp, X_valid, y_valid,
                                   num_epochs, learning_rate, weight_decay, batch_siz
        train_l_sum += train_ls[-1]
        valid_l_sum += valid_ls[-1]
```

```
        ## plotting train/validation loss
        if i == 0:
            d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'rmse',
                         range(1, num_epochs + 1), valid_ls,
                         ['train', 'valid'])
        print('fold %d, train rmse: %f, valid rmse: %f' % (
            i, train_ls[-1], valid_ls[-1]))

        ## predictions:
        test_features = test_features.reshape(test_features.shape[0], 1, test_feature
        preds = net(test_features)
        preds = preds.reshape(preds.shape[0]).asnumpy()
#         preds = net(test_features).asnumpy()
        pred_df.loc[:,i] = preds

#     args_save = args_save + "_valid{}".format(valid_l_sum / k)
    pred_df['SalePrice'] = pred_df[folds].mean(axis=1)
#     submission = pred_df[['Id','SalePrice']]
#     pred_df.to_csv('{}.csv'.format(args_save), index=False)

    return train_l_sum / k, valid_l_sum / k
```

In [ ]:
```
print('training on', ctx)
best_val_rmsle = 1.0
best_val_dict = {}
save_bar = 0.12  ## the hyperparameters to save if the loss is small than this number
k, num_epochs, batch_size = 4, 500, 64


for lr in [0.005, 0.01, 0.05, 0.1]:
    for weight_decay in [1, 3, 5, 7, 10, 25, 50, 100, 150, 300]:
        for dropout in [0.1, 0.2, 0.3, 0.4]:
            args_save = "lr{}_wd{}_dropout{}_k{}_ep{}_batch{}".format(lr, weight_decay
            print("lr:{}, weight_decay:{}, dropout:{}".format(lr, weight_decay, dropou
            train_l, valid_l = k_fold(k, train_features, train_labels, test_features,
                                      num_epochs, dropout,
                                      lr, weight_decay, batch_size)
            print('%d-fold validation: avg train rmse: %f, avg val2d rmse: %f'
                  % (k, train_l, valid_l))
            print("\n")
            if valid_l < save_bar:
                best_val_rmsle = valid_l
                best_val_dict[best_val_rmsle] = [lr, weight_decay, dropout]
```

3.1 Predict and Submit

Now that we know what a good choice of hyperparameters should be, we might as well use all the data to train on it (rather than just $1 - 1/k$ of the data that is used in the crossvalidation slices). The model that we obtain in this way can then be applied to the test set. Saving the estimates in a CSV file will simplify uploading the results to Kaggle.

```
In [ ]: def train_and_pred(train_features, test_features, train_labels, test_data, test_ID,
                           num_epochs, lr, weight_decay, dropout, batch_size):
            net = get_net(dropout=0)
            train_ls, _ = train(net, train_features, train_labels, None, None,
                                num_epochs, lr, weight_decay, batch_size)
            d2l.semilogy(range(1, num_epochs + 1), train_ls, 'epochs', 'rmse')
            print('train rmse %f' % train_ls[-1])

            # apply the network to the test set
            with autograd.predict_mode():
        #         print("train_mode? : ", autograd.is_training())
                preds = net(test_features).asnumpy()
            print(test_features.shape, preds.shape, len(test_ID))
            # reformat it for export to Kaggle
            test_data['SalePrice'] = preds  # pd.Series(preds.reshape(1, -1)[0])
            test_data['Id'] = np.asarray(test_ID)
            submission = test_data[['Id','SalePrice']]
            submission.to_csv('submission/submission_lr{}_wd{}_dropout{}.csv'.format(lr, weigh
            return(submission)
```

```
In [ ]: for k in sorted(best_val_dict.keys()):
            [lr, weight_decay, dropout] = best_val_dict[k]
            print("lr:{}, weight_decay:{}, dropout:{}".format(lr, weight_decay, dropout))
            submission = train_and_pred(train_features, test_features, train_labels, test_data
                           num_epochs, lr, weight_decay, dropout, batch_size)
            ensemble_df[k] = submission
```

```
In [ ]: ## ensemble all the prediction together

        total_concat = pd.DataFrame()
        for k in ensemble_df.keys():
            df = ensemble_df[k]
            df = df.set_index(["Id"])
            df.columns = [str(k)]
            total_concat = pd.concat([total_concat, df], axis=1)
        print(total_concat.shape)
        total_concat['SalePrice'] = total_concat.mean(axis=1)
        total_concat[['SalePrice']].to_csv("SalePrice.csv")
```