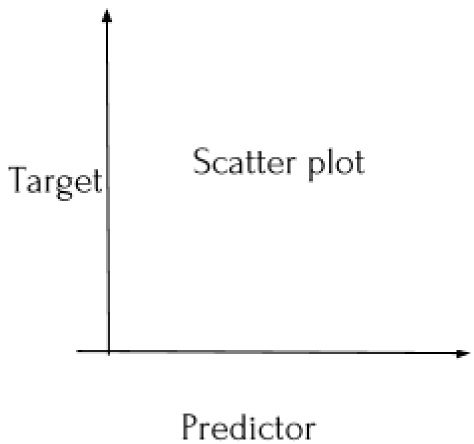
His data pre-processing strategy:

<u>Look for outliers in the training set by looking at scatter plots:</u>

Delete outliers if there aren't too many/you are not losing info.



<u>Target variable analysis:</u>

Is the distribution normal? Find out by plotting a histogram:

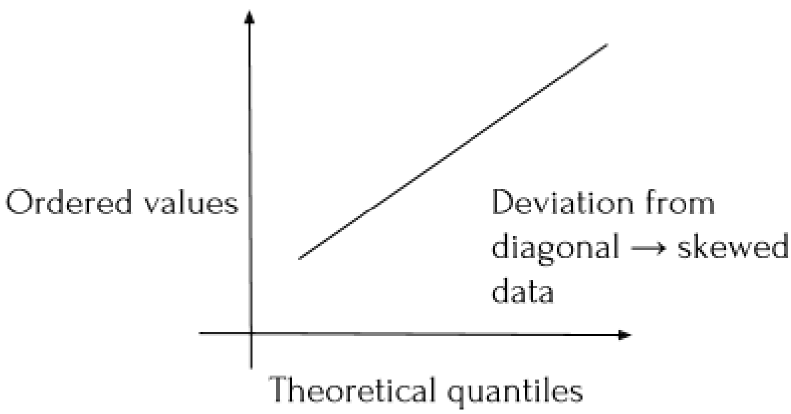
sns.distplot(train['SalePrice'], fit = norm)

Compare the fit parameters for the normal distribution, obtained by: (mu, sigma) = norm.fit(train['SalePrice']), with the actual mean & standard deviation

Analyze the QQ plot:

from scipy import stats

res = stats.probplot(train.SalePrice, plot=plt)



<u>If target variable is right skewed, apply log transformation:</u>

train.SalePrice = np.log1p(train.SalePrice)

<u>Before performing feature engineering, combine the train & test datasets:</u>

ntrain = train.shape[0]
ntest = test.shape[0]
all_data = pd.concat((train, test)).reset_index(drop=True)
#drop target variable
all_data.drop('SalePrice', inplace=True)

<u>Feature engineering includes:</u>

1. Imputing missing values
2. Transforming discrete numerical data (ex. month) into categorical data
3. Label encoding categorical data where ordering is important
4. Making linear combinations of features
5. Dealing with skewed features

Label Encoding (^^)

```
from sklearn.preprocessing import LabelEncoder
cols = (column names you've manually selected)
for c in cols:
	lbl = LabelEncoder()
	lbl.fit(list(all_data[c].values)
	all_data[c] = lbl.transform(list(all_data[c].values))
```

Dealing with skewed features (^^)

```
numeric_feats = all_data.dtypes[all_data.dtypes !=
"object"].index
skewed_feats = all_data[numeric_feats].apply(lambda x:
skew(x.dropna())).sort_values(ascending=False)
skewness = pd.DataFrame({'Skew': skewed_feats)
```

Skewed features can be dealt with either using log transforms (above) or BoxCox transformations

```
skewness = skewness[abs(skewness)>0.75]
from scipy.special import boxcox1p
skewed_feats = skewness.index
lam = 0.15
for feat in skewed_feats:
	all_data[feat] = boxcox1p(all_data[feat], lam)
```

Get dummies of categorical features

```
all_data = pd.get_dummies(all_data)
```

<u>Retrieve train & test</u>

train = all_data[:ntrain]
test = all_data[ntrain:]

His modeling strategy:

<u>Define a CV strategy</u>

```
from sklearn.model_selection import KFold, cross_val_score,
train_test_split

def rmsle_cv(model):
        kf = KFold(n_folds = 5, shuffle=True, random_state =
        42).get_n_splits(train.values)
        rmse = np.sqrt(-cross_val_score(model, train.values,
        y_train, scoring = “neg_mean_squared_error”, cv = kf))
        return(rmse)
```

<u>Define base models</u>

```
from sklearn.pipeline import make_pipeline
from sklearn.linear_model import Lasso, ElasticNet,
BayesianRidge
from sklearn.preprocessing import RobustScaler
```

lasso = make_pipeline(RobustScaler(), Lasso(alpha=0.0005, random_state=1))
//RobustScaler is to deal with outliers

Do this for each model (can also do for ensemble models or XGBoost)

Compute base model scores

score = rmsle_cv(lasso)
print(score.mean())
print(score.std())
Do for every base model

If we're not doing stacked regression, I think you just pick the model with the least error.

Example of a stacking approach

```
from sklearn.base import BaseEstimator, RegressorMixin,
TransformerMixin, clone

class AveragingModels(BaseEstimator, RegressorMixin,
TransformerMixin):
    def _init_(self, models):
        self.models = models

    #define clones of the original models to fit the data to
    def fit(self, X, y):
```

```
        self.models_ = [clone(x) for x in self.models]
        #train the cloned base models
        for model in self.models_:
            model.fit(X,y)
        return self

    #make predictions for clones models & average them
    def predict(self, X):
        predictions = np.column_stack([model.predict(X) for
        model in self.models_])
        return np.mean(predictions, axis=1)
```