notebook

May 23, 2018

1 Content

1. Exploratory Visualization
2. Data Cleaning
3. Feature Engineering
4. Modeling & Evaluation
5. Ensemble Methods

```
In [ ]: import numpy as np
        import pandas as pd
        import matplotlib.pyplot as plt
        import seaborn as sns
        import warnings
        warnings.filterwarnings('ignore')
        %matplotlib inline
        plt.style.use('ggplot')
```

```
In [ ]: from sklearn.base import BaseEstimator, TransformerMixin, RegressorMixin, clone
        from sklearn.preprocessing import LabelEncoder
        from sklearn.preprocessing import RobustScaler, StandardScaler
        from sklearn.metrics import mean_squared_error
        from sklearn.pipeline import Pipeline, make_pipeline
        from scipy.stats import skew
        from sklearn.decomposition import PCA, KernelPCA
        from sklearn.preprocessing import Imputer
```

```
In [ ]: from sklearn.model_selection import cross_val_score, GridSearchCV, KFold
        from sklearn.linear_model import LinearRegression
        from sklearn.linear_model import Ridge
        from sklearn.linear_model import Lasso
        from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, ExtraTree
        from sklearn.svm import SVR, LinearSVR
        from sklearn.linear_model import ElasticNet, SGDRegressor, BayesianRidge
        from sklearn.kernel_ridge import KernelRidge
        from xgboost import XGBRegressor
```

```
In [ ]: train = pd.read_csv('../input/train.csv')
        test = pd.read_csv('../input/test.csv')
```

2 Exploratory Visualization

- **It seems that the price of recent-built houses are higher. So later I 'll use labelencoder for three "Year" feature.**

```
In [ ]: plt.figure(figsize=(15,8))
        sns.boxplot(train.YearBuilt, train.SalePrice)
```

- **As is discussed in other kernels, the bottom right two two points with extremely large GrLivArea are likely to be outliers. So we delete them.**

```
In [ ]: plt.figure(figsize=(12,6))
        plt.scatter(x=train.GrLivArea, y=train.SalePrice)
        plt.xlabel("GrLivArea", fontsize=13)
        plt.ylabel("SalePrice", fontsize=13)
        plt.ylim(0,800000)
```

```
In [ ]: train.drop(train[(train["GrLivArea"]>4000)&(train["SalePrice"]<300000)].index,inplace=Tr
```

```
In [ ]: full=pd.concat([train,test], ignore_index=True)
```

```
In [ ]: full.drop(['Id'],axis=1, inplace=True)
        full.shape
```

3 Data Cleaning

3.0.1 Missing Data

```
In [ ]: aa = full.isnull().sum()
        aa[aa>0].sort_values(ascending=False)
```

- **Let's first imput the missing values of LotFrontage based on the median of LotArea and Neighborhood. Since LotArea is a continuous feature, We use qcut to divide it into 10 parts.**

```
In [ ]: full.groupby(['Neighborhood'])[['LotFrontage']].agg(['mean','median','count'])
```

```
In [ ]: full["LotAreaCut"] = pd.qcut(full.LotArea,10)
```

```
In [ ]: full.groupby(['LotAreaCut'])[['LotFrontage']].agg(['mean','median','count'])
```

```
In [ ]: full['LotFrontage']=full.groupby(['LotAreaCut','Neighborhood'])['LotFrontage'].transform
```

```
In [ ]: # Since some combinations of LotArea and Neighborhood are not available, so we just LotA
        full['LotFrontage']=full.groupby(['LotAreaCut'])['LotFrontage'].transform(lambda x: x.fi
```

- **Then we filling in other missing values according to data_description.**

```
In [ ]: cols=["MasVnrArea", "BsmtUnfSF", "TotalBsmtSF", "GarageCars", "BsmtFinSF2", "BsmtFinSF1"
        for col in cols:
            full[col].fillna(0, inplace=True)
```

```
In [ ]: cols1 = ["PoolQC" , "MiscFeature", "Alley", "Fence", "FireplaceQu", "GarageQual", "Garag
        for col in cols1:
            full[col].fillna("None", inplace=True)
```

```
In [ ]: # fill in with mode
        cols2 = ["MSZoning", "BsmtFullBath", "BsmtHalfBath", "Utilities", "Functional", "Electri
        for col in cols2:
            full[col].fillna(full[col].mode()[0], inplace=True)
```

- **And there is no missing data except for the value we want to predict !**

```
In [ ]: full.isnull().sum()[full.isnull().sum()>0]
```

4 Feature Engineering

- **Convert some numerical features into categorical features. It's better to use LabelEncoder and get_dummies for these features.**

```
In [ ]: NumStr = ["MSSubClass","BsmtFullBath","BsmtHalfBath","HalfBath","BedroomAbvGr","KitchenA
        for col in NumStr:
            full[col]=full[col].astype(str)
```

- **Now I want to do a long list of value-mapping.**
- **I was influenced by the insight that we should build as many features as possible and trust the model to choose the right features. So I decided to groupby SalePrice according to one feature and sort it based on mean and median. Here is an example:**

```
In [ ]: full.groupby(['MSSubClass'])[['SalePrice']].agg(['mean','median','count'])
```

- **So basically I'll do**
  '180' : 1 '30' : 2 '45' : 2 '190' : 3, '50' : 3, '90' : 3, '85' : 4, '40' : 4, '160' : 4 '70' : 5, '20' : 5, '75' : 5, '80' : 5, '150' : 5 '120': 6, '60' : 6

- __Different people may have different views on how to map these values, so just follow your instinct =^_^=__
  Below I also add a small "o" in front of the features so as to keep the original features to use get_dummies in a moment.

```
In [ ]: def map_values():
            full["oMSSubClass"] = full.MSSubClass.map({'180':1,
                                                '30':2, '45':2,
                                                '190':3, '50':3, '90':3,
                                                '85':4, '40':4, '160':4,
                                                '70':5, '20':5, '75':5, '80':5, '150':5,
                                                '120': 6, '60':6})


            full["oMSZoning"] = full.MSZoning.map({'C (all)':1, 'RH':2, 'RM':2, 'RL':3, 'FV':4})

            full["oNeighborhood"] = full.Neighborhood.map({'MeadowV':1,
```

```
                                              'IDOTRR':2, 'BrDale':2,
                                              'OldTown':3, 'Edwards':3, 'BrkSide':3,
                                              'Sawyer':4, 'Blueste':4, 'SWISU':4, 'NAme
                                              'NPkVill':5, 'Mitchel':5,
                                              'SawyerW':6, 'Gilbert':6, 'NWAmes':6,
                                              'Blmngtn':7, 'CollgCr':7, 'ClearCr':7, 'C
                                              'Veenker':8, 'Somerst':8, 'Timber':8,
                                              'StoneBr':9,
                                              'NoRidge':10, 'NridgHt':10})

full["oCondition1"] = full.Condition1.map({'Artery':1,
                                          'Feedr':2, 'RRAe':2,
                                          'Norm':3, 'RRAn':3,
                                          'PosN':4, 'RRNe':4,
                                          'PosA':5 ,'RRNn':5})

full["oBldgType"] = full.BldgType.map({'2fmCon':1, 'Duplex':1, 'Twnhs':1, '1Fam':2,

full["oHouseStyle"] = full.HouseStyle.map({'1.5Unf':1,
                                          '1.5Fin':2, '2.5Unf':2, 'SFoyer':2,
                                          '1Story':3, 'SLvl':3,
                                          '2Story':4, '2.5Fin':4})

full["oExterior1st"] = full.Exterior1st.map({'BrkComm':1,
                                            'AsphShn':2, 'CBlock':2, 'AsbShng':2,
                                            'WdShing':3, 'Wd Sdng':3, 'MetalSd':3, 'Stu
                                            'BrkFace':4, 'Plywood':4,
                                            'VinylSd':5,
                                            'CemntBd':6,
                                            'Stone':7, 'ImStucc':7})

full["oMasVnrType"] = full.MasVnrType.map({'BrkCmn':1, 'None':1, 'BrkFace':2, 'Stone

full["oExterQual"] = full.ExterQual.map({'Fa':1, 'TA':2, 'Gd':3, 'Ex':4})

full["oFoundation"] = full.Foundation.map({'Slab':1,
                                          'BrkTil':2, 'CBlock':2, 'Stone':2,
                                          'Wood':3, 'PConc':4})

full["oBsmtQual"] = full.BsmtQual.map({'Fa':2, 'None':1, 'TA':3, 'Gd':4, 'Ex':5})

full["oBsmtExposure"] = full.BsmtExposure.map({'None':1, 'No':2, 'Av':3, 'Mn':3, 'Gd

full["oHeating"] = full.Heating.map({'Floor':1, 'Grav':1, 'Wall':2, 'OthW':3, 'GasW'

full["oHeatingQC"] = full.HeatingQC.map({'Po':1, 'Fa':2, 'TA':3, 'Gd':4, 'Ex':5})

full["oKitchenQual"] = full.KitchenQual.map({'Fa':1, 'TA':2, 'Gd':3, 'Ex':4})
```

```
        full["oFunctional"] = full.Functional.map({'Maj2':1, 'Maj1':2, 'Min1':2, 'Min2':2, '

        full["oFireplaceQu"] = full.FireplaceQu.map({'None':1, 'Po':1, 'Fa':2, 'TA':3, 'Gd':

        full["oGarageType"] = full.GarageType.map({'CarPort':1, 'None':1,
                                                   'Detchd':2,
                                                   '2Types':3, 'Basment':3,
                                                   'Attchd':4, 'BuiltIn':5})

        full["oGarageFinish"] = full.GarageFinish.map({'None':1, 'Unf':2, 'RFn':3, 'Fin':4})

        full["oPavedDrive"] = full.PavedDrive.map({'N':1, 'P':2, 'Y':3})

        full["oSaleType"] = full.SaleType.map({'COD':1, 'ConLD':1, 'ConLI':1, 'ConLw':1, 'Ot
                                               'CWD':2, 'Con':3, 'New':3})

        full["oSaleCondition"] = full.SaleCondition.map({'AdjLand':1, 'Abnorml':2, 'Alloca':



        return "Done!"
```

In [ ]:
```
map_values()
```

In [ ]:
```
# drop two unwanted columns
full.drop("LotAreaCut",axis=1,inplace=True)
full.drop(['SalePrice'],axis=1,inplace=True)
```

4.1 Pipeline

- **Next we can build a pipeline. It's convenient to experiment different feature combinations once you've got a pipeline.**
- **Label Encoding three "Year" features.**

In [ ]:
```
class labelenc(BaseEstimator, TransformerMixin):
    def __init__(self):
        pass

    def fit(self,X,y=None):
        return self

    def transform(self,X):
        lab=LabelEncoder()
        X["YearBuilt"] = lab.fit_transform(X["YearBuilt"])
        X["YearRemodAdd"] = lab.fit_transform(X["YearRemodAdd"])
        X["GarageYrBlt"] = lab.fit_transform(X["GarageYrBlt"])
        return X
```

- **Apply log1p to the skewed features, then get_dummies.**

In [ ]:
```
class skew_dummies(BaseEstimator, TransformerMixin):
    def __init__(self,skew=0.5):
        self.skew = skew

    def fit(self,X,y=None):
        return self

    def transform(self,X):
        X_numeric=X.select_dtypes(exclude=["object"])
        skewness = X_numeric.apply(lambda x: skew(x))
        skewness_features = skewness[abs(skewness) >= self.skew].index
        X[skewness_features] = np.log1p(X[skewness_features])
        X = pd.get_dummies(X)
        return X
```

In [ ]:
```
# build pipeline
pipe = Pipeline([
    ('labenc', labelenc()),
    ('skew_dummies', skew_dummies(skew=1)),
    ])
```

In [ ]:
```
# save the original data for later use
full2 = full.copy()
```

In [ ]:
```
data_pipe = pipe.fit_transform(full2)
```

In [ ]:
```
data_pipe.shape
```

In [ ]:
```
data_pipe.head()
```

- **use robustscaler since maybe there are other outliers.**

In [ ]:
```
scaler = RobustScaler()
```

In [ ]:
```
n_train=train.shape[0]

X = data_pipe[:n_train]
test_X = data_pipe[n_train:]
y= train.SalePrice

X_scaled = scaler.fit(X).transform(X)
y_log = np.log(train.SalePrice)
test_X_scaled = scaler.transform(test_X)
```

4.2 Feature Selection

- **I have to confess, the feature engineering above is not enough, so we need more.**

- **Combining different features is usually a good way, but we have no idea what features should we choose. Luckily there are some models that can provide feature selection, here I use Lasso, but you are free to choose Ridge, RandomForest or GradientBoostingTree.**

```
In [ ]: lasso=Lasso(alpha=0.001)
        lasso.fit(X_scaled,y_log)
```

```
In [ ]: FI_lasso = pd.DataFrame({"Feature Importance":lasso.coef_}, index=data_pipe.columns)
```

```
In [ ]: FI_lasso.sort_values("Feature Importance",ascending=False)
```

```
In [ ]: FI_lasso[FI_lasso["Feature Importance"]!=0].sort_values("Feature Importance").plot(kind=
        plt.xticks(rotation=90)
        plt.show()
```

- **Based on the "Feature Importance" plot and other try-and-error, I decided to add some features to the pipeline.**

```
In [ ]: class add_feature(BaseEstimator, TransformerMixin):
            def __init__(self,additional=1):
                self.additional = additional

            def fit(self,X,y=None):
                return self

            def transform(self,X):
                if self.additional==1:
                    X["TotalHouse"] = X["TotalBsmtSF"] + X["1stFlrSF"] + X["2ndFlrSF"]
                    X["TotalArea"] = X["TotalBsmtSF"] + X["1stFlrSF"] + X["2ndFlrSF"] + X["Garag

                else:
                    X["TotalHouse"] = X["TotalBsmtSF"] + X["1stFlrSF"] + X["2ndFlrSF"]
                    X["TotalArea"] = X["TotalBsmtSF"] + X["1stFlrSF"] + X["2ndFlrSF"] + X["Garag

                    X["+_TotalHouse_OverallQual"] = X["TotalHouse"] * X["OverallQual"]
                    X["+_GrLivArea_OverallQual"] = X["GrLivArea"] * X["OverallQual"]
                    X["+_oMSZoning_TotalHouse"] = X["oMSZoning"] * X["TotalHouse"]
                    X["+_oMSZoning_OverallQual"] = X["oMSZoning"] + X["OverallQual"]
                    X["+_oMSZoning_YearBuilt"] = X["oMSZoning"] + X["YearBuilt"]
                    X["+_oNeighborhood_TotalHouse"] = X["oNeighborhood"] * X["TotalHouse"]
                    X["+_oNeighborhood_OverallQual"] = X["oNeighborhood"] + X["OverallQual"]
                    X["+_oNeighborhood_YearBuilt"] = X["oNeighborhood"] + X["YearBuilt"]
                    X["+_BsmtFinSF1_OverallQual"] = X["BsmtFinSF1"] * X["OverallQual"]

                    X["-_oFunctional_TotalHouse"] = X["oFunctional"] * X["TotalHouse"]
                    X["-_oFunctional_OverallQual"] = X["oFunctional"] + X["OverallQual"]
                    X["-_LotArea_OverallQual"] = X["LotArea"] * X["OverallQual"]
                    X["-_TotalHouse_LotArea"] = X["TotalHouse"] + X["LotArea"]
                    X["-_oCondition1_TotalHouse"] = X["oCondition1"] * X["TotalHouse"]
```

```
        X["-_oCondition1_OverallQual"] = X["oCondition1"] + X["OverallQual"]


        X["Bsmt"] = X["BsmtFinSF1"] + X["BsmtFinSF2"] + X["BsmtUnfSF"]
        X["Rooms"] = X["FullBath"]+X["TotRmsAbvGrd"]
        X["PorchArea"] = X["OpenPorchSF"]+X["EnclosedPorch"]+X["3SsnPorch"]+X["Scree
        X["TotalPlace"] = X["TotalBsmtSF"] + X["1stFlrSF"] + X["2ndFlrSF"] + X["Gara


        return X
```

- **By using a pipeline, you can quickily experiment different feature combinations.**

```
In [ ]: pipe = Pipeline([
            ('labenc', labelenc()),
            ('add_feature', add_feature(additional=2)),
            ('skew_dummies', skew_dummies(skew=1)),
            ])
```

4.3 PCA

- **Im my case, doing PCA is very important. It lets me gain a relatively big boost on leaderboard. At first I don't believe PCA can help me, but in retrospect, maybe the reason is that the features I built are highly correlated, and it leads to multicollinearity. PCA can decorrelate these features.**

- **So I'll use approximately the same dimension in PCA as in the original data. Since the aim here is not deminsion reduction.**

```
In [ ]: full_pipe = pipe.fit_transform(full)

In [ ]: full_pipe.shape

In [ ]: n_train=train.shape[0]
        X = full_pipe[:n_train]
        test_X = full_pipe[n_train:]
        y= train.SalePrice

        X_scaled = scaler.fit(X).transform(X)
        y_log = np.log(train.SalePrice)
        test_X_scaled = scaler.transform(test_X)

In [ ]: pca = PCA(n_components=410)

In [ ]: X_scaled=pca.fit_transform(X_scaled)
        test_X_scaled = pca.transform(test_X_scaled)

In [ ]: X_scaled.shape, test_X_scaled.shape
```

5 Modeling & Evaluation

```
In [ ]: # define cross validation strategy
        def rmse_cv(model,X,y):
            rmse = np.sqrt(-cross_val_score(model, X, y, scoring="neg_mean_squared_error", cv=5)
            return rmse
```

- **We choose 13 models and use 5-folds cross-calidation to evaluate these models.**

  Models include:

- LinearRegression
- Ridge
- Lasso
- Random Forrest
- Gradient Boosting Tree
- Support Vector Regression
- Linear Support Vector Regression
- ElasticNet
- Stochastic Gradient Descent
- BayesianRidge
- KernelRidge
- ExtraTreesRegressor
- XgBoost

```
In [ ]: models = [LinearRegression(),Ridge(),Lasso(alpha=0.01,max_iter=10000),RandomForestRegres
                  ElasticNet(alpha=0.001,max_iter=10000),SGDRegressor(max_iter=1000,tol=1e-3),Ba
                  ExtraTreesRegressor(),XGBRegressor()]
```

```
In [ ]: names = ["LR", "Ridge", "Lasso", "RF", "GBR", "SVR", "LinSVR", "Ela","SGD","Bay","Ker","
        for name, model in zip(names, models):
            score = rmse_cv(model, X_scaled, y_log)
            print("{}: {:.6f}, {:.4f}".format(name,score.mean(),score.std()))
```

- **Next we do some hyperparameters tuning. First define a gridsearch method.**

```
In [ ]: class grid():
            def __init__(self,model):
                self.model = model

            def grid_get(self,X,y,param_grid):
                grid_search = GridSearchCV(self.model,param_grid,cv=5, scoring="neg_mean_squared
                grid_search.fit(X,y)
                print(grid_search.best_params_, np.sqrt(-grid_search.best_score_))
                grid_search.cv_results_['mean_test_score'] = np.sqrt(-grid_search.cv_results_['m
                print(pd.DataFrame(grid_search.cv_results_)[['params','mean_test_score','std_tes
```

5.0.1 Lasso

```
In [ ]: grid(Lasso()).grid_get(X_scaled,y_log,{'alpha': [0.0004,0.0005,0.0007,0.0009],'max_iter'
```

5.0.2 Ridge

```
In [ ]: grid(Ridge()).grid_get(X_scaled,y_log,{'alpha':[35,40,45,50,55,60,65,70,80,90]})
```

5.0.3 SVR

```
In [ ]: grid(SVR()).grid_get(X_scaled,y_log,{'C':[11,13,15],'kernel':["rbf"],"gamma":[0.0003,0.0
```

5.0.4 Kernel Ridge

```
In [ ]: param_grid={'alpha':[0.2,0.3,0.4], 'kernel':["polynomial"], 'degree':[3],'coef0':[0.8,1]
        grid(KernelRidge()).grid_get(X_scaled,y_log,param_grid)
```

5.0.5 ElasticNet

```
In [ ]: grid(ElasticNet()).grid_get(X_scaled,y_log,{'alpha':[0.0008,0.004,0.005],'l1_ratio':[0.0
```

6 Ensemble Methods

6.0.1 Weight Average

- **Average base models according to their weights.**

```
In [ ]: class AverageWeight(BaseEstimator, RegressorMixin):
            def __init__(self,mod,weight):
                self.mod = mod
                self.weight = weight

            def fit(self,X,y):
                self.models_ = [clone(x) for x in self.mod]
                for model in self.models_:
                    model.fit(X,y)
                return self

            def predict(self,X):
                w = list()
                pred = np.array([model.predict(X) for model in self.models_])
                # for every data point, single model prediction times weight, then add them toge
                for data in range(pred.shape[1]):
                    single = [pred[model,data]*weight for model,weight in zip(range(pred.shape[0
                    w.append(np.sum(single))
                return w
```

```
In [ ]: lasso = Lasso(alpha=0.0005,max_iter=10000)
        ridge = Ridge(alpha=60)
        svr = SVR(gamma= 0.0004,kernel='rbf',C=13,epsilon=0.009)
        ker = KernelRidge(alpha=0.2 ,kernel='polynomial',degree=3 , coef0=0.8)
        ela = ElasticNet(alpha=0.005,l1_ratio=0.08,max_iter=10000)
        bay = BayesianRidge()
```

```
In [ ]: # assign weights based on their gridsearch score
        w1 = 0.02
        w2 = 0.2
        w3 = 0.25
        w4 = 0.3
        w5 = 0.03
        w6 = 0.2
```

```
In [ ]: weight_avg = AverageWeight(mod = [lasso,ridge,svr,ker,ela,bay],weight=[w1,w2,w3,w4,w5,w6
```

```
In [ ]: score = rmse_cv(weight_avg,X_scaled,y_log)
        print(score.mean())
```

- **But if we average only two best models, we gain better cross-validation score.**

```
In [ ]: weight_avg = AverageWeight(mod = [svr,ker],weight=[0.5,0.5])
```

```
In [ ]: score = rmse_cv(weight_avg,X_scaled,y_log)
        print(score.mean())
```

6.1 Stacking

- **Aside from normal stacking, I also add the "get_oof" method, because later I'll combine features generated from stacking and original features.**

```
In [ ]: class stacking(BaseEstimator, RegressorMixin, TransformerMixin):
            def __init__(self,mod,meta_model):
                self.mod = mod
                self.meta_model = meta_model
                self.kf = KFold(n_splits=5, random_state=42, shuffle=True)

            def fit(self,X,y):
                self.saved_model = [list() for i in self.mod]
                oof_train = np.zeros((X.shape[0], len(self.mod)))

                for i,model in enumerate(self.mod):
                    for train_index, val_index in self.kf.split(X,y):
                        renew_model = clone(model)
                        renew_model.fit(X[train_index], y[train_index])
                        self.saved_model[i].append(renew_model)
                        oof_train[val_index,i] = renew_model.predict(X[val_index])

                self.meta_model.fit(oof_train,y)
                return self

            def predict(self,X):
                whole_test = np.column_stack([np.column_stack(model.predict(X) for model in sing
                                              for single_model in self.saved_model])
                return self.meta_model.predict(whole_test)
```

```
    def get_oof(self,X,y,test_X):
        oof = np.zeros((X.shape[0],len(self.mod)))
        test_single = np.zeros((test_X.shape[0],5))
        test_mean = np.zeros((test_X.shape[0],len(self.mod)))
        for i,model in enumerate(self.mod):
            for j, (train_index,val_index) in enumerate(self.kf.split(X,y)):
                clone_model = clone(model)
                clone_model.fit(X[train_index],y[train_index])
                oof[val_index,i] = clone_model.predict(X[val_index])
                test_single[:,j] = clone_model.predict(test_X)
            test_mean[:,i] = test_single.mean(axis=1)
        return oof, test_mean
```

- **Let's first try it out ! It's a bit slow to run this method, since the process is quite compliated.**

```
In [ ]: # must do imputer first, otherwise stacking won't work, and i don't know why.
        a = Imputer().fit_transform(X_scaled)
        b = Imputer().fit_transform(y_log.values.reshape(-1,1)).ravel()
```

```
In [ ]: stack_model = stacking(mod=[lasso,ridge,svr,ker,ela,bay],meta_model=ker)
```

```
In [ ]: score = rmse_cv(stack_model,a,b)
        print(score.mean())
```

- **Next we extract the features generated from stacking, then combine them with original features.**

```
In [ ]: X_train_stack, X_test_stack = stack_model.get_oof(a,b,test_X_scaled)
```

```
In [ ]: X_train_stack.shape, a.shape
```

```
In [ ]: X_train_add = np.hstack((a,X_train_stack))
```

```
In [ ]: X_test_add = np.hstack((test_X_scaled,X_test_stack))
```

```
In [ ]: X_train_add.shape, X_test_add.shape
```

```
In [ ]: score = rmse_cv(stack_model,X_train_add,b)
        print(score.mean())
```

- **You can even do parameter tuning for your meta model after you get "X_train_stack", or do it after combining with the original features. but that's a lot of work too !**

6.1.1 Submission

```
In [ ]: # This is the final model I use
        stack_model = stacking(mod=[lasso,ridge,svr,ker,ela,bay],meta_model=ker)
```

```
In [ ]: stack_model.fit(a,b)
```

```
In [ ]: pred = np.exp(stack_model.predict(test_X_scaled))
```

```
In [ ]: result=pd.DataFrame({'Id':test.Id, 'SalePrice':pred})
        result.to_csv("submission.csv",index=False)
```