Walkthrough of My First Data Science Challenge

As expected, 90% of my time went into data cleaning and feature engineering and the rest 10% on applying Machine Learning models onto them. I understood how important it is to **‘know your data’**. Blindly feeding it to the SOTA model without any data engineering will only take you so far. I had to read a lot of blogs and Kaggle notebooks but the one I religiously followed was [this](https://www.kaggle.com/agodwinp/stacking-house-prices-walkthrough-to-top-5/). In this one, I’ll give an outline of the process I followed in very simple words and with examples so that beginners like me could get a sense of where to start.

For reference, here are my notebooks with the complete code so that you could follow along with the explanations —

*Data cleaning and feature engineering*:

<https://github.com/Shikhargupta/kaggle_house_pricing/blob/master/Feat_Engg.ipynb>

*Feature importance and modelling*:

<https://github.com/Shikhargupta/kaggle_house_pricing/blob/master/Ensemble.ipynb>

**The Ames Housing Dataset**

Before getting into the process, let’s get an overview of the dataset in question. It has 80 columns (features) each describing a particular aspect of the house (e.g. neighborhood, utilities, roof type etc.) which may or may not play a role in deciding its price (target variable). It consists of a total of 2920 observations, 1460 of which will be used for fitting (*training data*) and the rest for determining how good is your model performing (*test data*). Our goal is to predict the SalePrice for a house as closely as possible by taking the features (might not be all of them) in consideration. I would suggest you go through the [data description](https://www.kaggle.com/c/house-prices-advanced-regression-techniques/data) to get accustomed to what each field means. The criteria for evaluation is [*Root-Mean-Squared-Error (RMSE)*](https://en.wikipedia.org/wiki/Root-mean-square_deviation) between the logarithm of the predicted value and the logarithm of the observed sales price. Lower the score, better is your model at prediction!

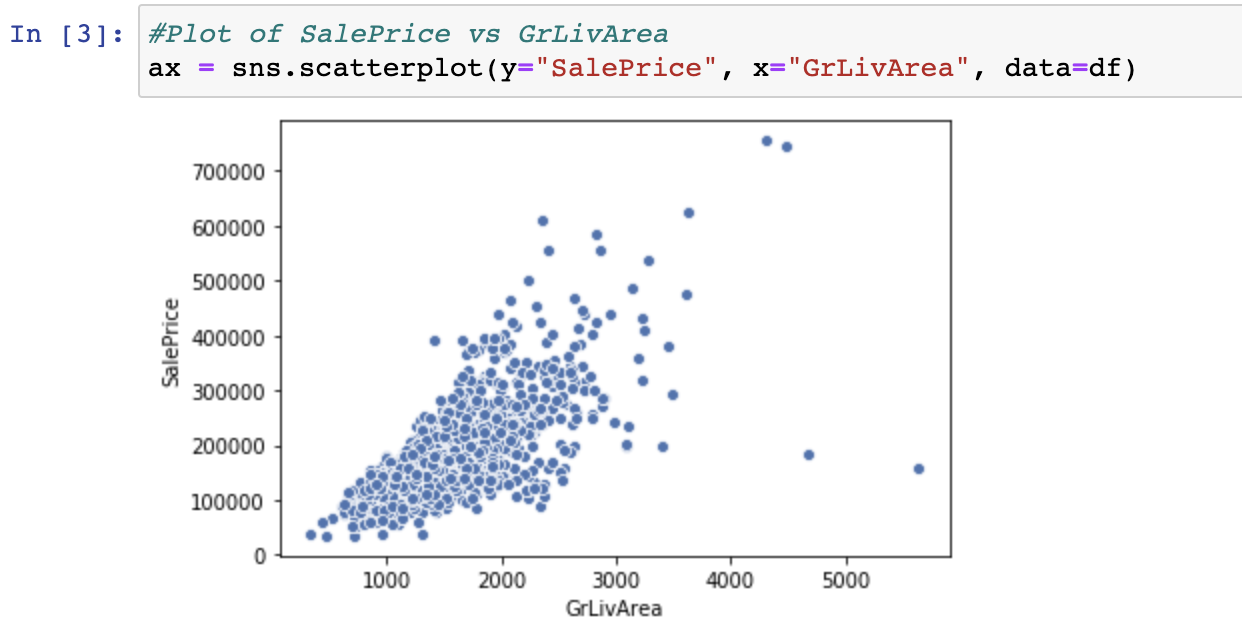
**Data Cleaning (Outliers and NAN’s)**

First and foremost, we’ll clean our dataset to avoid any errors or discrepancies when we feed it to our model.

Always keep in mind that the end goal is to train our machine learning model and the data should be prepared in the same way. Be aware of how any action we perform might at the end improve or degrade the efficiency of the regressor/classifier.

For example, the presence of a **NAN** field might send the training routine for a toss and throw an error. Another killer of our prediction could be an [**outlier**](https://towardsdatascience.com/ways-to-detect-and-remove-the-outliers-404d16608dba). The housing data we have was collected and stored manually and is prone to human error. An obvious large (20 bedrooms) or small (10 sq. ft. of LotArea) observation could sway our model in the wrong direction and affect our score. We either need to get rid of these data points or impute the values appropriately.

As far as Ames Housing dataset is concerned, we are good on the outliers front. The author has already pointed out the suspected outliers in the [paper](http://jse.amstat.org/v19n3/decock.pdf) which I have visualised here in this plot



Data points with *GrLivArea > 4000* are outliers due to abnormally high SalePrices and LivArea and shall be removed. Although there is nothing else mentioned, I found one discrepancy in the feature ‘GarageYr’ which depicts the year in which the garage was built in the house. The value was *2027* which is way into the future. Instead of getting rid of the observation, I imputed it with the mode. Before doing that I made sure that the house *does* have a garage by checking the ‘GarageType’ attribute. It is important to ‘***know your data’*** before making any changes so be aware of these small but significant aspects.

For NAN’s I plotted a graph for all the features having at least one of those. Then I analysed them one by one and decided how to handle the missing value. Most of them were replaced by the Mode of that feature but there were some special cases like this –

#For BsmtFinSF1 we observed the only datapoint for which the value was missing had BsmtFinType1 as NA i.e

#there is no basement. So we can fill the SF as 0

df\_cum['BsmtFinSF1'] = df\_cum['BsmtFinSF1'].fillna(0)

#Same goes for BsmtFinSF2

df\_cum['BsmtFinSF2'] = df\_cum['BsmtFinSF2'].fillna(0)

#Same data point is valid for BsmtUnfSF and TotalBsmtSF. We can fill them all with zeros.

df\_cum['BsmtUnfSF'] = df\_cum['BsmtUnfSF'].fillna(0)

df\_cum['TotalBsmtSF'] = df\_cum['TotalBsmtSF'].fillna(0)

In the absence of a basement, we can impute the missing value in basement square feet as 0

**Feature Engineering**

This is undoubtedly the most important and time-consuming part of the process. My approach involved taking them up one at a time (*80* *of them!*) and apply the transformations. This might sound cumbersome but believe you me, it will go a long way. My methodology was to [visualize](https://towardsdatascience.com/complete-guide-to-data-visualization-with-python-2dd74df12b5e) them using different graphs (seaborn gives you pretty graphs!) and base my decision (to either drop or keep them with any changes) on it. This will be a good time to use your analytical powers. There is no rocket science here and the dataset is very intuitive (more the number of bedrooms, higher the price) and logical. I’ll explain what I did with some examples in broad categories.

**Polynomials**

If you are planning to regress on a huge dataset, linear shouldn’t be the only way to do it. Some features might have a relationship with the target variable which cannot be expressed by a singular degree. Therefore, I handpicked the *10* most correlated variables (from the correlation matrix) and created their square, cube and square root versions and added them to the dataset.

#Square

for col in poly\_cols:

df\_cum[col + '\_square'] = df\_cum[col]\*\*2

#Cube

for col in poly\_cols:

df\_cum[col + '\_cube'] = df\_cum[col]\*\*3

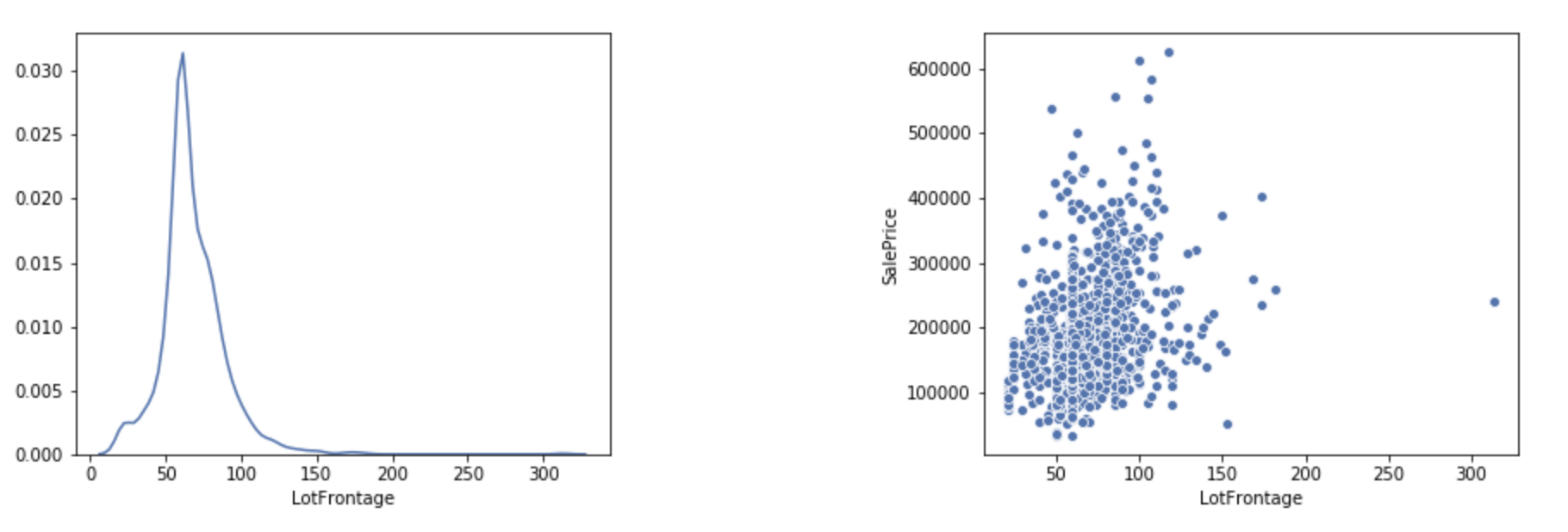
#Square root

for col in poly\_cols:

df\_cum[col + '\_sqrt'] = df\_cum[col]\*\*0.5

**Continuous Features**

These are the ones which have continuous values like LotArea, GarageArea, 1stFlrSF etc. My go-to tactic for this kind of features was to plot their distribution graphs and their scatter-plot with the SalePrice (our target variable). Here is a snippet of the ‘LotFrontage’



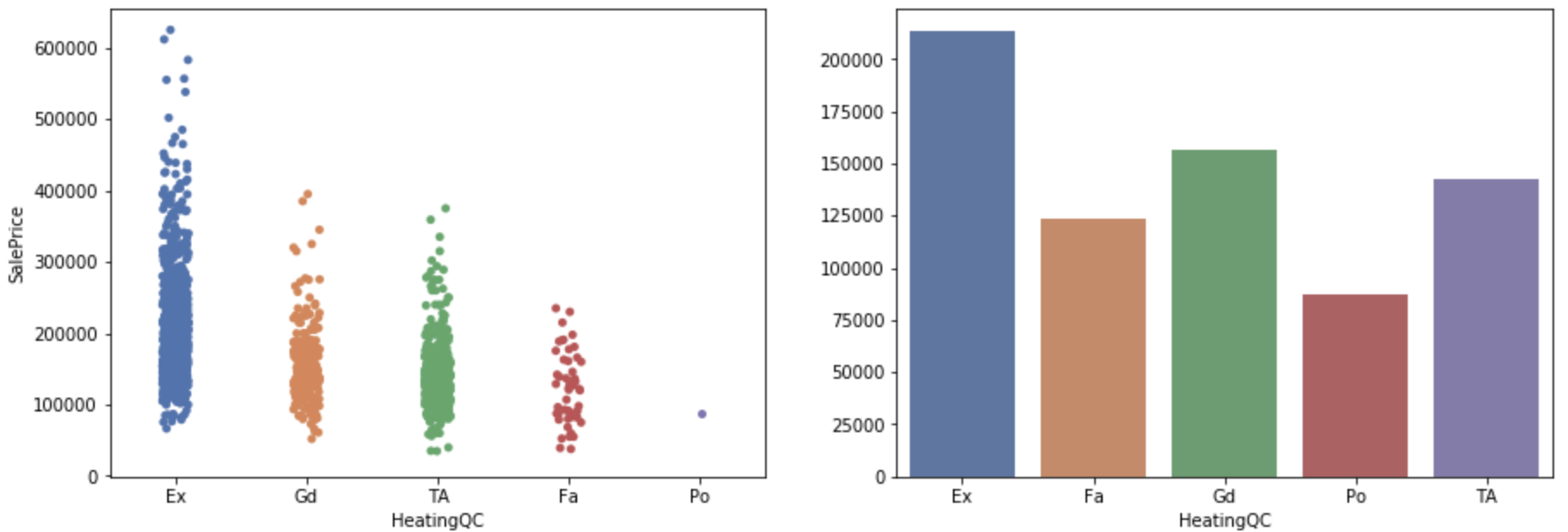
Left: Distribution graph; Right: Scatter plot with SalePrice

The scatter plot shows that the effect of LotFrontage might not be very significant on the SalePrice but it is something and we’ll keep it. The left graph shows how positively skewed this feature distribution is. *Always keep in mind, more the distribution is closer to the normal, better your prediction model will perform.*So, always try to [transform](https://becominghuman.ai/how-to-deal-with-skewed-dataset-in-machine-learning-afd2928011cc) your data to make it more normal-y. [Techniques](https://towardsdatascience.com/understanding-feature-engineering-part-1-continuous-numeric-data-da4e47099a7b) usually used to do so are — log transformation and binning. I’ll take a log here.

Some of the features may seem continuous but will better suit when converted to categorical. For example, features related to year (year when the house was built, year when it was remodelled etc.). Also, other integral features like — number of bedrooms, number of pools, cars that could fit in the garage. They all will make more sense if converted to categorical and I did the same.

**Categorical Features**

All non-continuous variables are categorical. The practice I follow here is to visualise the feature using strip-plot and the average SalePrice for each category. Let’s take *HeatingQC* (heating quality) as an example:

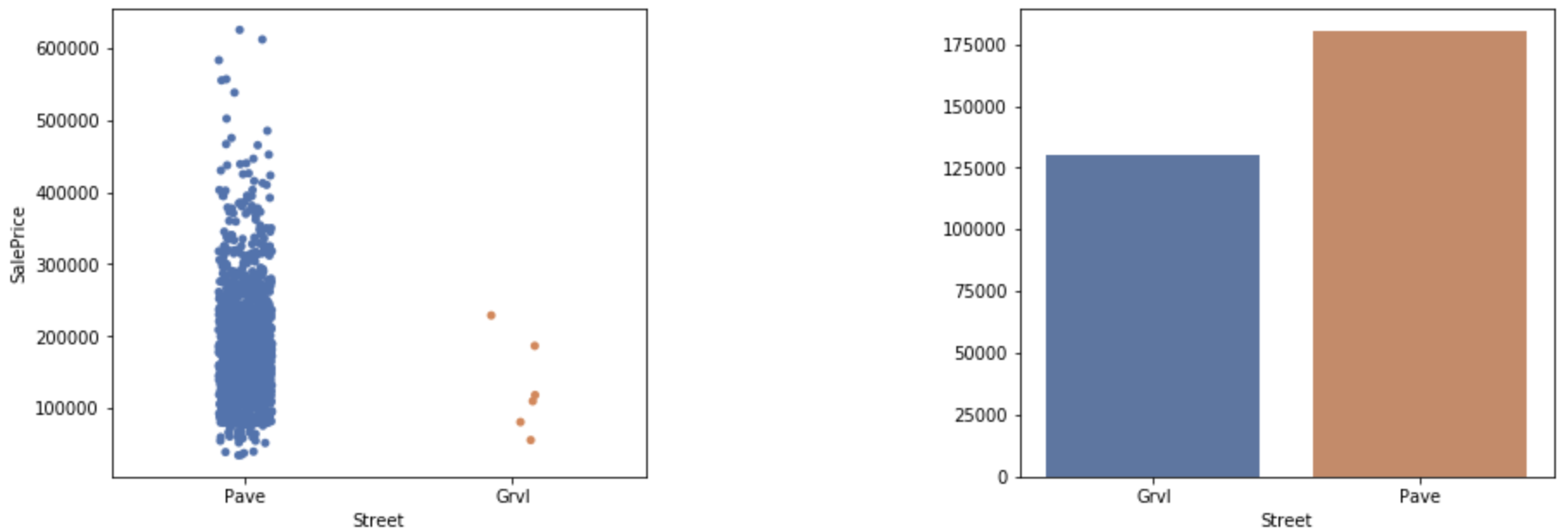


Left: Stripplot; Right: Average

The first one lets me visualise the number of houses in each category and points out any discrepancies (we’ll see later how). The second one helps me decide whether to treat it as an ordered feature (replace each category with an integer) or create [dummy variables](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.get_dummies.html). This example is clearly ordinal (Ex being the highest rating and having the highest average price and so on) so we’ll replace them with ordered integers. [**Boxplots**](https://towardsdatascience.com/understanding-boxplots-5e2df7bcbd51) could also be an efficient way to visualize the data.

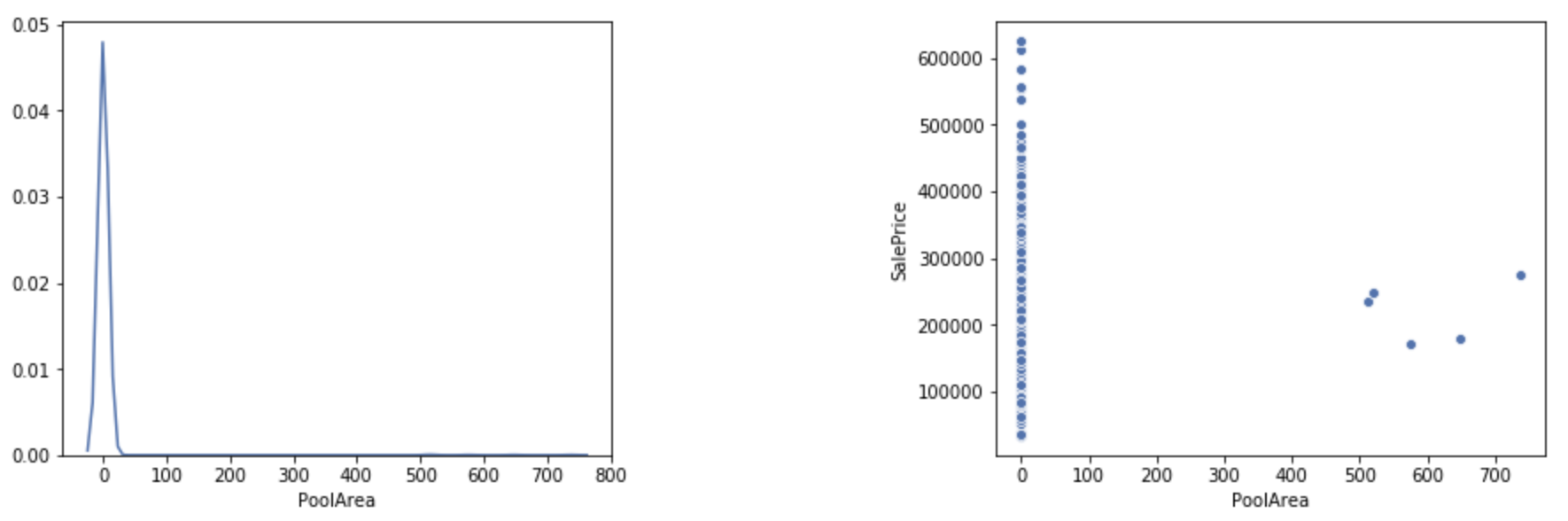
**Drop (useless) Features**

We want to keep our feature space as dense as possible to get the maximum efficiency from our model. So we try to get rid of the features we think might not contribute significantly enough to the target variable. I came across a few and here are some examples.



Street: Type of road access to the property

The strip plot reveals that most of the houses have ‘*Pave*’ as the street type and the count for ‘*Grvl*’ is negligible. This feature has almost constant behaviour for all house prices and hence do not contribute much to the target. We can drop this feature.



PoolArea: *Pool area in square feet*

Similarly for *PoolArea*, the value is 0 for most of the observations (almost all the houses do not have a pool) and hence wouldn’t serve us much in the prediction. We can drop it.

**Feature Importance**

Even after an extensive analysis of all the features, some redundant features might escape the naked eye. We will use XGBoost’s inbuilt feature importance [function](https://towardsdatascience.com/be-careful-when-interpreting-your-features-importance-in-xgboost-6e16132588e7) to further identify the insignificant columns and get rid of them before we fit our model. Let’s first get the index for all the 324 features we have after transformations.

xg\_reg = xgb.XGBRegressor(base\_score=0.5, colsample\_bylevel=1, colsample\_bytree=0.4,

gamma=0, learning\_rate=0.07, max\_delta\_step=0, max\_depth=3,

min\_child\_weight=1.5, missing=None, n\_estimators=400, nthread=-1,

objective='reg:linear', reg\_alpha=0.75, reg\_lambda=0.45,

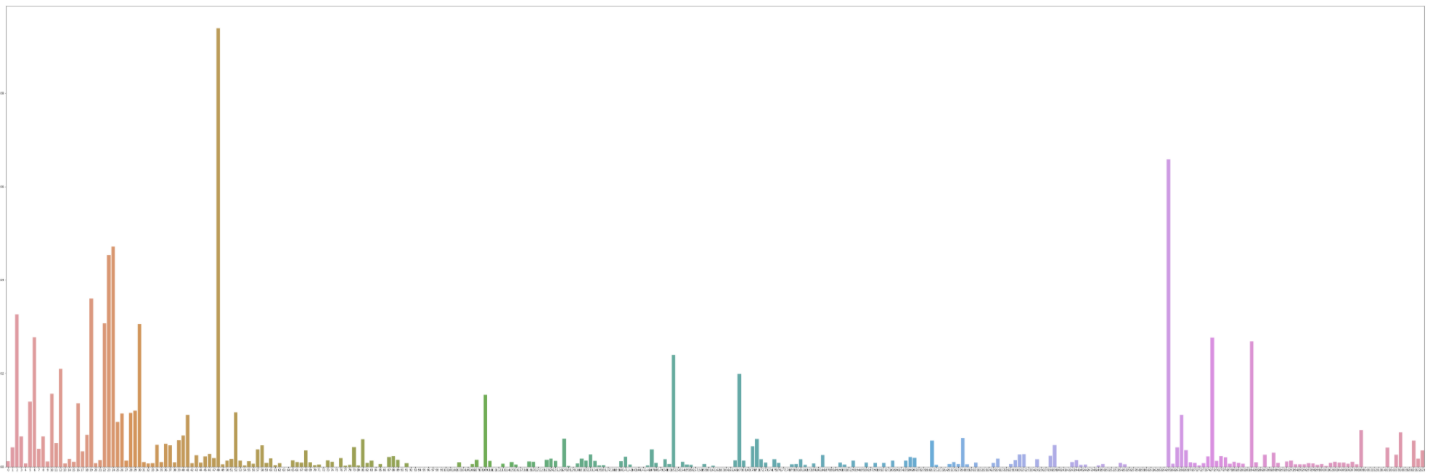
scale\_pos\_weight=1, seed=42, silent=True, subsample=0.6)

xg\_reg.fit(X\_train,np.log(y\_train))

x\_ax = np.arange(len(xg\_reg.feature\_importances\_))

plt.figure(figsize=(90, 30))

sns.barplot(x=x\_ax, y=xg\_reg.feature\_importances\_)



Importance of features with respect to the SalePrice. Quite a lot of them are zero.

We can see that numerous of our features do not contribute towards determining the SalePrice and hence we’ll get rid of them. After dropping the features with score 0, our dataset got reduced from *324* columns to *218!*

**Machine Learning Modelling**

inally, the part for which we started all this. Frankly, 90% of our job is already done. If you have a well structured and dense dataset, modelling will be a cakewalk. I prepared a list of the most famous regression models that I wanted to fit my data on (listed later in this section). Then I split my test data into test and validation sets. I trained the model on the test set and evaluated on validation. This helped me visualize which model is performing better. Here is an example of the XGBoost regressor.

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_train, y\_train, test\_size=0.2, random\_state=0)

xg\_reg = xgb.XGBRegressor(base\_score=0.5, colsample\_bylevel=1, colsample\_bytree=0.4,

gamma=0, learning\_rate=0.07, max\_delta\_step=0, max\_depth=3,

min\_child\_weight=1.5, missing=None, n\_estimators=400, nthread=-1,

objective='reg:linear', reg\_alpha=0.75, reg\_lambda=0.45,

scale\_pos\_weight=1, seed=42, silent=True, subsample=0.6)

xg\_reg.fit(X\_train,np.log(y\_train))

y\_pred = np.exp(xg\_reg.predict(X\_test))

print('Root Mean Squared Error:', np.sqrt(metrics.mean\_squared\_error(np.log(y\_test), np.log(y\_pred))))

If you notice, I have taken a log transform of the SalePrice as it gives a better distribution. While getting the results I have converted the prices back to the original scale by taking an exponential of the value.

**Parameter Tuning**

Each regression model has its own set of parameters which are highly sensitive to the dataset upon which it is trained. If you want highly optimized results (which you do!), then it is imperative to have the best performing set of parameters. The conventional way is to run a grid search using [GridSearchCV](https://scikit-learn.org/stable/modules/grid_search.html) but it requires a considerable amount of computation power. If you don’t have access to a GPU, better avoid it (my MacBookPro took more than 7 hours for XGBoost and still could complete the search). The alternate way is to do a ‘**parameter sweep**’ in which we evaluate our training by varying one parameter and keeping the rest as same. This requires one to have a fair idea of the range of the values that might be suitable for the parameter we are sweeping. Here is an example of finding alpha for RidgeRegressor.

###################### Parameter sweeping ##########################

val\_list = []

score\_list = []

for x in np.arange(10,300,10):

model = Ridge(alpha=x)

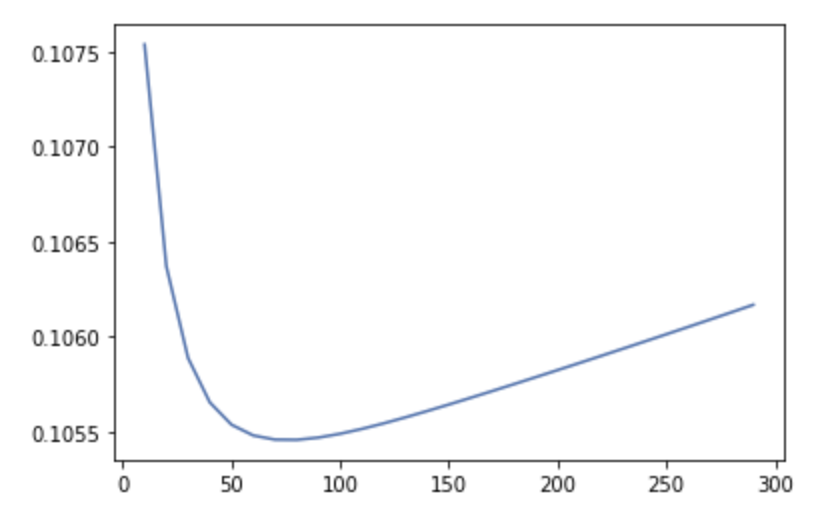
model.fit(X\_train,np.log(y\_train))

y\_pred = np.exp(model.predict(X\_test))

score\_list.append(np.sqrt(metrics.mean\_squared\_error(np.log(y\_test), np.log(y\_pred))))

val\_list.append(x)

sns.lineplot(x=val\_list, y=score\_list, markers=True)



alpha~75 gives the lowest RMSE and is the best possible value for this dataset

**Ensemble**

So till now, we have scores from the individual regression models. But wouldn’t it be better if we could weigh in the predictions of all the models and get a combined result? Every model has its own speciality. Some are better at picking up certain info from the data and some are more susceptible to others. It would be better if we ‘consult’ various models and then reach to a final prediction. This is called [ensembling](https://towardsdatascience.com/simple-guide-for-ensemble-learning-methods-d87cc68705a2" \t "_blank). In this, we train all our models over the training data, and while predicting we combine the results. This combination could be anything — an average of all the values, weighted mean etc. I used the weighted mean option and decided the weights according to how individual models performed over the validation set. Here is the code.

#### Gradient Boosting Regressor (0.1134)####

gbregressor = GradientBoostingRegressor(learning\_rate=0.1, n\_estimators=180)

gbregressor.fit(X\_train, np.log(y\_train))

y\_pred = np.exp(gbregressor.predict(X\_test))

all\_pred = np.concatenate((all\_pred, 0.05\*y\_pred))

#### Lasso LarsIC (0.119)####

lassolars = LassoLarsIC(criterion='aic')

lassolars.fit(X\_train, np.log(y\_train))

y\_pred = np.exp(lassolars.predict(X\_test))

all\_pred = np.vstack((all\_pred, 0.05\*y\_pred))

#### Random Forest (0.12401)####

rf\_regressor = RandomForestRegressor(n\_estimators = 325, random\_state = 0)

rf\_regressor.fit(X\_train, np.log(y\_train))

y\_pred = np.exp(rf\_regressor.predict(X\_test))

all\_pred = np.vstack((all\_pred, 0.05\*y\_pred))

#### Bayesian Ridge (0.1054)####

bayesianridge = BayesianRidge(alpha\_1=130, alpha\_2=0.0017, lambda\_1=0.00001, lambda\_2=0.000001)

bayesianridge.fit(X\_train, np.log(y\_train))

y\_pred = np.exp(bayesianridge.predict(X\_test))

all\_pred = np.vstack((all\_pred, 0.12\*y\_pred))

#### ElasticNET (0.10032)####

elasticnet = ElasticNet(alpha=0.011, l1\_ratio=0.5)

elasticnet.fit(X\_train, np.log(y\_train))

y\_pred = np.exp(elasticnet.predict(X\_test))

all\_pred = np.vstack((all\_pred, 0.21\*y\_pred))

#### Lasso (0.10026)####

lasso = Lasso(alpha=0.005)

lasso.fit(X\_train, np.log(y\_train))

y\_pred = np.exp(lasso.predict(X\_test))

all\_pred = np.vstack((all\_pred, 0.24\*y\_pred))

#### Ridge Regressor (0.1054)####

ridge = Ridge(alpha=75)

ridge.fit(X\_train, np.log(y\_train))

y\_pred = np.exp(ridge.predict(X\_test))

all\_pred = np.vstack((all\_pred, 0.14\*y\_pred))

#### XGBoost Regressor (0.10624)#####

xg\_reg = xgb.XGBRegressor(base\_score=0.5, colsample\_bylevel=1, colsample\_bytree=0.4,

gamma=0, learning\_rate=0.07, max\_delta\_step=0, max\_depth=3,

min\_child\_weight=1.5, missing=None, n\_estimators=400, nthread=-1,

objective='reg:linear', reg\_alpha=0.75, reg\_lambda=0.45,

scale\_pos\_weight=1, seed=42, silent=True, subsample=0.6)

xg\_reg.fit(X\_train,np.log(y\_train))

y\_pred = np.exp(xg\_reg.predict(X\_test))

all\_pred = np.vstack((all\_pred, 0.14\*y\_pred))

y\_pred = np.sum(all\_pred, axis=0)

**Conclusion**

Although experimenting with different models on the surface might give you good results, but it is important to understand the math behind it so that in future you could choose a regressor by reasoning. Also, my way is surely not the best to approach this house pricing problem and you can devise your own, so, keep trying y’all!