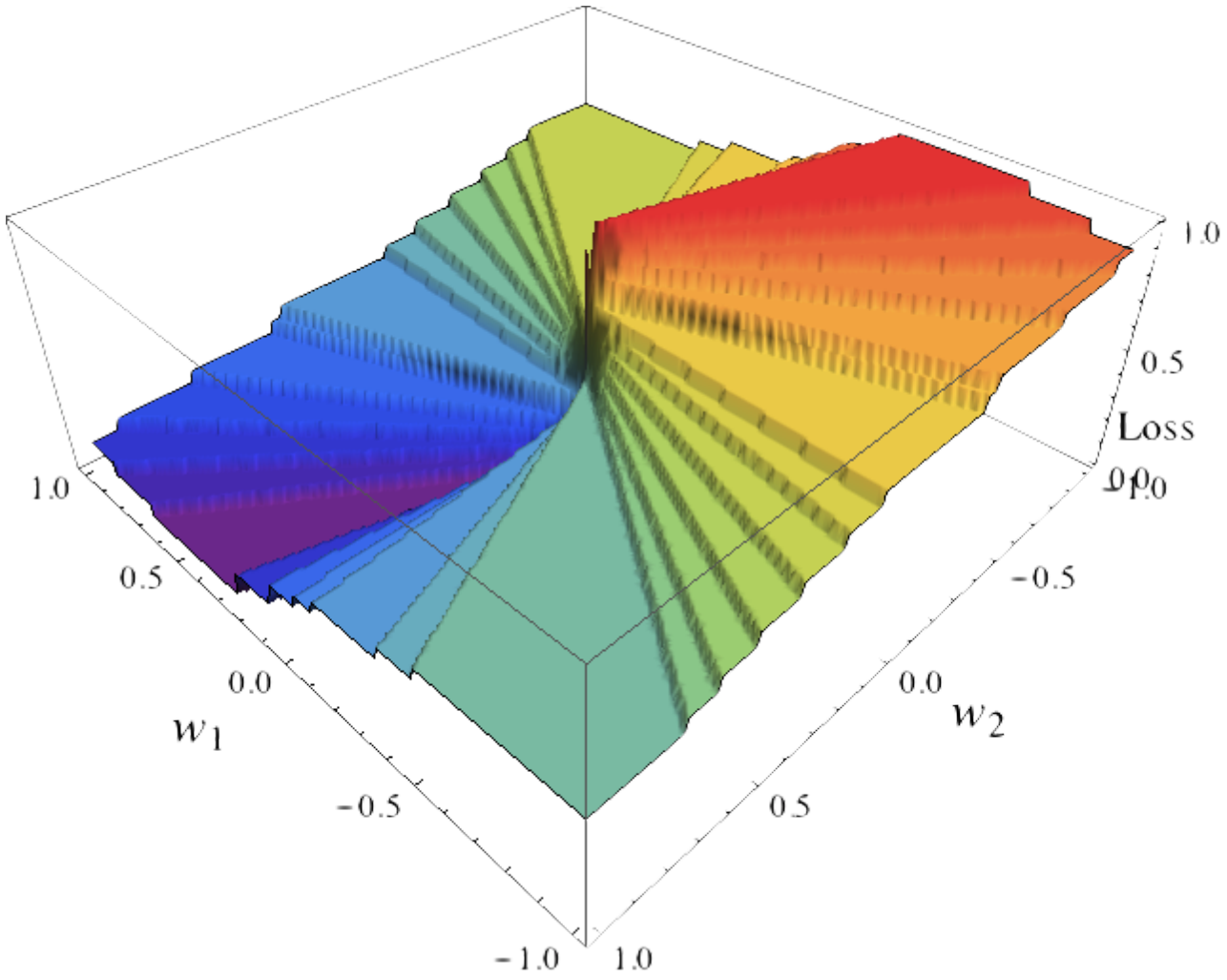
Machine Learning Engineering 1: House Sales Estimation.

Lee Tanenbaum

Aug 25, 2018 · 13 min read



Visualization of a Custom Loss Function

This blog post will be the first lesson of how to think as a Machine Learning (ML) Engineer. We will be working on a supervised learning problem with a focus on designing a custom loss function. Think with tricks like this and you will confidently get an entry-level ML Engineer role.

This blog post will assume you have a basic understanding of modern ML.

For a non-technical preface, read https://medium.com/@leetandata/machine-learning-preface-ba69bca4701d.

For a mathematical and numpy code detailed explanation, start with https://medium.com/@leetandata/neural-network-introduction-for-software-engineers-1611d382c6aa

For instructions installing Python for Machine Learning on Windows read: https://medium.com/@leetandata/basic-python-setup-for-ml-for-windows-users-aaadb2be534c

Problem Introduction

The problem at hand is to estimate the time it will take to sell a condo from a simulated dataset.

The data is the listing date, the sales date (if the condo has sold), the price, and numeric features about the condo.

Quick Review of Regression

Regression takes in a feature data as a vector. Every datapoint is defined as some number of input features, for instance features about the house. The regression model uses these features to predict the Target (it guesses Yhat which is meant to approximate the true Y, or the Target, from the dataset). The model trains itself by learning from the dataset to minimize a loss function.

Back To our Specific Problem

For simple regression problems, your data comes to you as a dataset of features X and target values Y. You split the dataset into a training set and a validation set, train the model on the training set and validate its performance on the validation set.

Condo sales estimation is similar to a simple regression problem, but poses four intricacies/complications that ML must account for:

- First, we need to divide up the data. We will use some of the data (called the training data) to train our model on and some of our data to validate the performance of the model we train. For general datasets, we can just divide up the

dataset randomly. For time series data we can chop up the data to train on data until a cutoff, and validate on data after the cutoff. This simulates how well the model would do if we trained it on the past, and released it to predict the future real-time. This is difficult for this dataset compared to most time-series problems because there are two time stamps for each sample, the listing date and the sales date.

- Next, we don't always know what the correct sales date is. For apartments that haven't sold yet, we know their sales date occurs after the last sale in the dataset, but we don't know when the final sale will be. How do we create our loss function if sometimes we only know a minimum value instead of the true value.

- Third, we notice that our input feature data has missing values. What do we do with that? Someone probably just forgot to list the number of bedrooms or bathrooms or the square feet of the condo at times. We will explore a few options for handling this intricacy.

- Lastly, what's the most meaningful evaluation metric to use to evaluate the many performance of our models? We want to use something standard like Mean Squared Error or an R2 coefficient, but how can we account for larger values not caring as much about small differences? For instance, 1 day vs 5 days seems like a big difference, but 81 days vs 85 days should be considered pretty accurate. And what do we do that without knowing the true values for houses that haven't sold yet?

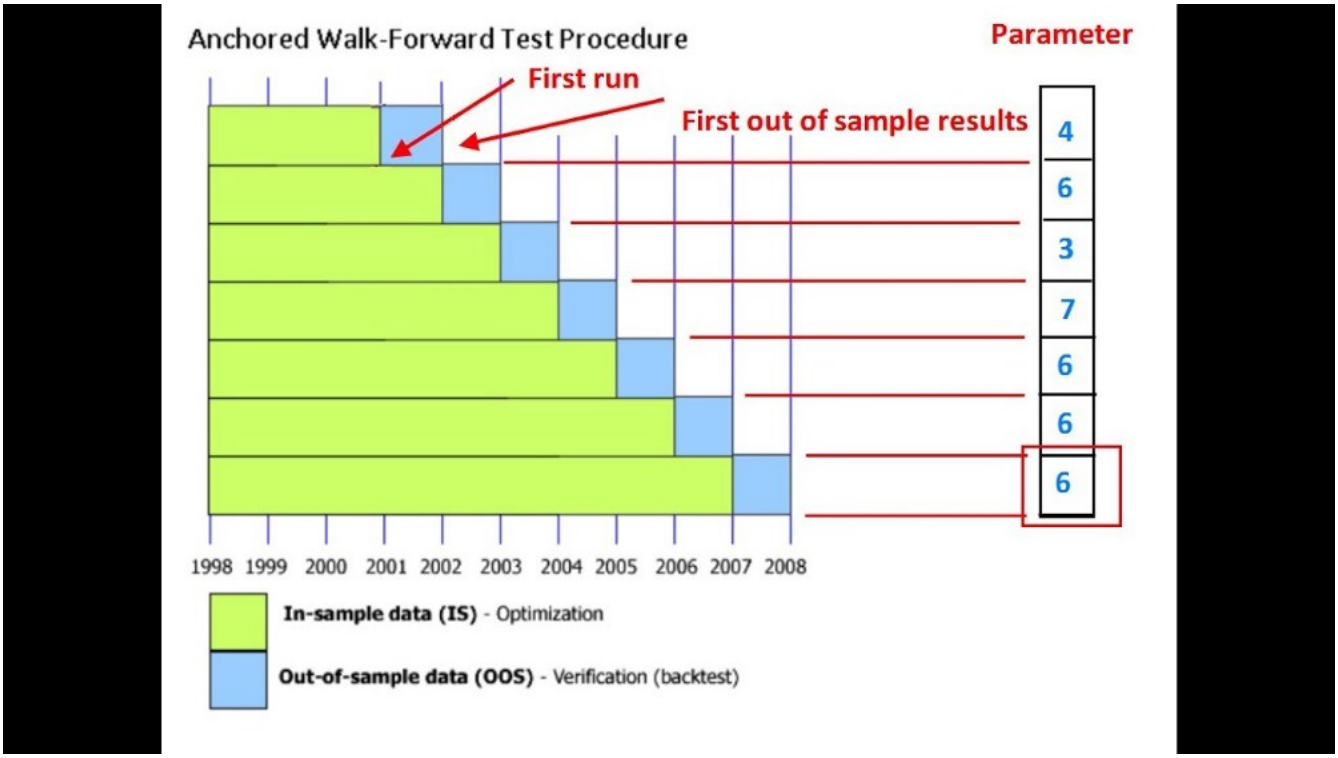
1: Train/Validate/Test Walk Forward Optimization

Normally, in Machine learning, we randomly split up (specific numbers can fluctuate depending on problem) 70% of our data to train a model on, 20% of our to validate the performance on, and reserve 10% of our data for the very end to test our release performance on.

We repeatedly change the model, train, and validate, until our validation accuracy seems to be maximized. We believe the model that generalizes to new unseen data performs best, so we utilize the model that does best on the validation set.

Finally, we evaluate once on our test set to approximate how well our model will generalize to the real world. In this way we can build complicated models to represent

our data, and then calculate metrics that don't over-estimate our performance estimates for how well it will perform.



Walk Forward Optimization

In time series modeling, we use Walk Forward Optimization, where we split up the time series into many temporal cutoffs T1, T2, T3, etc. We first train on data until T1, and validate our performance on the time period between T1 and T2. We then extend our training dataset until T2, and then validate on the period between T2 and T3. Our overall validation performance metric, therefore, is our average validation performance from the different data cuts, and our test set can be the very last time slice of the data, or some data points throughout the different time slices that were excluded from the validation data points.

We noted that every time step, however, has two data points: The listing date and the sales date. So if we are training until T1, what can we realistically have known until that point?

During training, we will know about all Listings before T1. Within these condos, we will hide all sales dates that occur after T1.

During validation, we need to validate on all condos that sold between T1 and T2, and also all listings that has been listed before T2 and not sold until after T1. Simplifying

our validation set, we validate our performance on listings that are sold after T1, and were listed before T2.

The above is crucial, because it allows us to use all the data we have available to train and validate, but not be overly optimistic and not allow ourselves to cheat and see into the future. Avoiding data leakage, or cheating, is very important in Machine Learning, because data leakage let's us believe our model is unrealistically accurate. We could then release our model and be very unfavorably surprised by how well it performs on new data.

2: Meaningful Loss Function

Let's propose a reasonable baseline loss function for the problem. We care about the difference between the time to sell a condo and the predicted time, and the more off we are the more we need to correct our prediction. Therefore we could propose the Mean Squared Error (MSE). But to some degree, if a condo sold in 1 day and we predicted 10, we are much more off than if the condo sold in 81 and we predicted 90. Therefore, maybe we should use Mean Squared Percent Error (MSPE), where we penalize the model based on the percent off our prediction was. Either of these seems reasonable, but since we care about both, let's just take the geometric mean of the two and use that as our loss function, taking both into consideration. This isn't that business interpretable, but it's our **loss function**, and doesn't have to be our **evaluation metric**. We'll get to that later.

So what happens when a condo hasn't sold? Let's say a condo was listed on T0, the training set has until T1, but the condo was sold at T2 (or never). At T1, the condo still hasn't sold, and we are pretending we haven't seen the future past T1, so we need to mark the condo as "Not yet sold". So how do we use this datapoint to improve our model?

Let's say our model predicts the condo will sell between T0 and T1. In that case, we know the model is wrong and the model should have predicted a bigger number, so let's pretend the true value is the end of the training dataset in that case.

But what if our model predicts a value larger than T1? We don't know if that is too big or too small, because it is correctly after the end of our training dataset, so let's not give a loss function to our model in that case.

In that way, we're using all of our available data and optimizing our model with a meaningful loss function towards a business-valuable approximation that minimizes percent error!

3: Data Imputation

Someone just forgot to list the number of bedrooms or bathrooms or the square feet of the condo for some of our datapoints. But we're those randomly forgotten or intentionally left out?

We create extra flags for each of our features that just represents was the data missing or not, and we feed those flags into the model, and we also fill in the missing values.

To fill in missing values, we could have a separate network that learns to predict values or other custom solutions (this is very popular for structured data such as language, image, or audio), but for our case lets just manually impute the date. Two popular approaches are to either replace the missing values with the average value from their field, or to find some very similar condos and replace the missing value with the average value from the similar condos.

We call finding similar data points to impute K-Nearest Neighbor (KNN) Imputation. The way this works is that we find K data points nearest to the data point with a missing feature. For the missing feature, we take the average of those K nearest points value of that feature.

Using the euclidean distance to measure "nearest neighbors" sounds reasonable, but what if one feature had values between 0–1000, and another value had features between 0–1? In that case, the feature with the large scale would influence euclidean distance way more than the small feature. For instance x1, x2 = [600, 0.1] may measure closer to [590, 0.9] than [580, 0.2], even though both have very similar x1 values, while their x2 values are on completely ends of the spectrum.

How do we force all the features to have similar impact on 'nearest neighbor' imputation? Let's normalize their impact by first scaling the features so that they all have mean 0 and standard deviation 1 so that every feature has the same amount of impact on the imputation!

4: Evaluation Metrics

Designing meaningful evaluation metrics is one of the most important tasks for any ML project. What are the numbers that best represents "Is this model learning something business meaningful"?

Some engineers have a bad habit of sticking with off-the-shelf metrics such as Mean squared error, or the correlation between their predictions and the true values. They then present these numbers to business members who will become frustrated with the meaningless (rightfully so) tech jargon. In our case, we could use many metrics like the Pearson correlation coefficient of the logarithm of the time, but that would be complicated to explain.

Instead let's use the business-simplest baseline, the mean absolute percent error (MAPE), which tells us the mean average absolute value of the ratio between your prediction and the true value. To interpret this metric, a value of .1 would represent on average being 10% off from the true value. A value of .2 would indicate an average of 20% from the true value such as 80% or 120% from the correct value. This metric cannot be negative, and 0 represents perfection.

But what about unknown values? Let's use MAPE for the values we know the true value for, and lets have another metric for the percent of unsold condos correctly classified as unsold (sales date predicted to be past the end of the dataset). We could have also replaced the unsold condos by pretending they were sold on the last day of the dataset, and calculated MAPE that way, which would been simpler to interpret but less accurate.

For Evaluation Metrics, Think like a Business Person

So as a mathematician, we wrote an optimization function. Next, we wrote an evaluation function to build explainable results. But is that the most meaningful way to understand our performance? In this case, no, because we really haven't tried to think about what's meaningful for our problem.

We calculated a metric that summarizes our model's performance. Great? But we shouldn't have summarized it completely! Our data came to us with significant feature intricacies that effected our performance. Some condos sold after a **week**, and some after a **year**. Also, some condos were priced in the tens of **thousands**, and some in the tens of **millions**. Especially since our evaluation metric is error divided by true time to sales, we propose to group our accuracy by region of true time to sales, and plot both the average and the Interquartile Range of these values.

In that way, we can ask the question “For quick house sales, we are around X1 percent off, but for houses that we will stay on the market longer, we are around X2 percent off”. And with that, both the business people and the mathematicians should be much happier understanding how their model is performing.

. . .

Transition Point: Ideas to Code

Wow! Lot of content above. We covered

- Meaningful train/val walk-forward optimization loop
- A custom loss function
- How to handle missing features
- A meaningful evaluation metric and graphical representation

We are now done with proposing concepts now, so now we just need to implement the ideas from above!

. . .

Implementation

Let’s get started with some imports and creating a directory for images for evaluation metrics, and read in our dataset.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import tensorflow as tf
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import r2_score
import datetime
import os
from Imputer.knn_imputer import KNNImputer
import pdb
imagedir = 'images'
if not os.path.isdir(imagedir):
    os.mkdir(imagedir)
```

```
14
15  # For numeric stability
16  EPSILON = 1e-10
17
18  df = pd.read_csv(
19      'houselistings_simulated.csv', parse_dates=['ListingDate', 'SalesDate'])
20
21  dataset_end = df['SalesDate'].max()
22
23  FIRST_CUTOFF = pd.to_datetime('2017-02-01')
24  LAST_CUTOFF = df['SalesDate'].max() - pd.DateOffset(months=1)
25
```

gistfile1.txt hosted with ♡ by GitHub view raw

Our data is in a Pandas DataFrame, which is much like a SQL Table.
The top of it is visualized below:

	ListingDate	SalesDate	bathrooms	bedrooms	broker	sqft	years_since_remodel
0	2017-10-08	NaT	4.0	2.0	0.0	2263.200297	3.189165
1	2017-12-20	NaT	2.0	NaN	NaN	1159.398712	13.919031
2	2017-10-04	NaT	3.0	0.0	0.0	771.294248	2.384003
3	2015-02-19	2015-08-06	1.0	1.0	1.0	1012.003459	2.515936
4	2017-09-22	NaT	3.0	2.0	1.0	1732.544302	4.220575
5	2015-05-22	2015-09-17	3.0	NaN	NaN	1978.633589	3.165217
6	2015-08-26	2016-01-26	3.0	1.0	0.0	1456.249476	3.764364
7	2017-06-10	2017-10-01	2.0	2.0	1.0	2087.071320	1.497180
8	2015-09-22	2016-01-30	2.0	2.0	0.0	2115.377784	7.116858
9	2017-07-12	2017-11-15	1.0	2.0	1.0	1574.613888	9.179006
10	2016-03-31	2016-06-11	2.0	5.0	1.0	4821.129464	12.194909
11	2016-01-15	2016-08-09	3.0	0.0	0.0	658.702465	1.694604
12	2016-05-15	2016-09-18	2.0	2.0	0.0	1563.651124	5.828813
13	2015-10-08	2016-02-28	3.0	1.0	1.0	1209.609109	10.114303
14	2015-07-11	2016-01-03	2.0	1.0	0.0	951.420517	28.459737
15	2017-06-17	2017-10-22	3.0	1.0	0.0	1306.011806	6.116814
16	2015-03-26	2015-10-14	1.0	1.0	0.0	891.324889	15.600577
17	2017-05-11	2017-10-28	3.0	0.0	1.0	701.406088	2.116797
18	2015-08-11	2015-12-06	3.0	2.0	0.0	1908.997352	2.606371
19	2016-06-09	2017-01-09	3.0	0.0	0.0	678.737605	17.345514
20	2016-09-18	2017-03-13	3.0	0.0	0.0	824.441365	1.044661
21	2017-12-23	NaT	3.0	0.0	1.0	764.168824	15.975150

Pandas DataFrame Data Table

Data Preparation

Now we're all set up. Now lets build a function that lets us cut data for our train/testing splits. We need to accept an optional start date for our data slice, and an optional end date.

```
def split_at(df, start_date=None, end_date=None):
    data_idx = np.ones(df.shape[0], dtype=bool)
    if start_date is not None:
        data_idx = data_idx & (df.SalesDate >= start_date)
    if end_date is not None:
        data_idx = data_idx & (df.ListingDate < end_date)
    return df.loc[data_idx, :].copy()
```

gistfile1.txt hosted with ♡ by GitHub view raw

This function cuts up the total data set to only keep data after a start date or before an end date. This will be helpful for build training and validation data sets.

Now we need a scaler to normalize our data to prepare it for imputation. A scaler learns from the training data how to normalize, and applies those normalization parameters (the mean and standard deviation) to the validation dataset.

We also need to build an imputer from our training data that learns to impute missing values, and we need to be able to apply it to both a training dataset and a validation dataset.

```
def scale(df, scaling_mean=None, scaling_std=None):
    numeric_features_train = df.select_dtypes(include=[np.number]).copy()
    if scaling_mean is None:
        scaling_mean, scaling_std = numeric_features_train.mean(
        ), numeric_features_train.std()
    numeric_features_train = (
        numeric_features_train - scaling_mean) / scaling_std
    return numeric_features_train, scaling_mean, scaling_std


def build_imputer(numeric_features_train):
    imputer = KNNImputer()
    imputer.fit(numeric_features_train)
    return imputer


def apply_imputer(imputer, features):
```

```
    features = features.copy()
    imputer.fill_in(features)
    return features
```

gistfile1.txt hosted with ♡ by GitHub view raw

The above code prepares our model's input data.

Now let's talk about our target variable! Our target variable is the number of days between the condo being listed and when it was sold. When the condo hasn't been sold yet, the sales date is set to the current day, and we have a sold flag to represent if the condo has sold yet (we'll use both later)

```
def build_Y(df, end):
    sold = df['SalesDate'] < end
    Y = df['SalesDate'].clip(upper=end).fillna(end) - df['ListingDate']
    return sold, Y.dt.days.astype(float)
```

gistfile1.txt hosted with ♡ by GitHub view raw

Modeling Using Tensorflow

Our model takes in some number of input features. It has placeholders to accept values for each data point, sold (whether that condo has sold), x (the input features), and y (days until condo sold).

The model uses a Neural Network (Easy Explanation or Detailed Explanation) to predict how long the condo will take to sell. The It then computes the loss function and evaluation metric, and builds an optimizer to minimize the loss function.

We also create lists to track our performance throughout training, and computes other loss functions just for demonstration's sake for the tutorial.

```
class Model():
    def __init__(self, input_size, layer_sizes):
        self.input_size = input_size
        self.layer_sizes = layer_sizes
        smoothing_factor = 1

        self.sold = tf.placeholder(tf.float32, shape=(None))
        self.x = tf.placeholder(tf.float32, shape=(None, input_size))
        self.y = tf.placeholder(tf.float32, shape=(None))

        self.layers = [self.x]
        for layer_size in layer_sizes:
```

```
        for layer_size in layer_sizes:
            next_layer = tf.nn.leaky_relu(
                tf.layers.dense(self.layers[-1], layer_size))
            self.layers.append(next_layer)

        self.output = tf.nn.softplus(tf.layers.dense(self.layers[-1], 1))[:, 0]

        self.loss_indicator = (tf.cast(self.output < self.y, tf.float32) *
                               (1 - self.sold) + self.sold)

        def build_loss(loss, loss_indicator):
            loss_numerator = tf.reduce_sum(loss * self.loss_indicator)
            loss_denominator = (tf.reduce_sum(self.loss_indicator)) + EPSILON
            return loss_numerator / loss_denominator

        error_by_sample = (self.output - self.y)
        percent_error_by_sample = error_by_sample / (self.y + smoothing_factor)

        MSPE_raw = tf.reduce_mean(tf.square(percent_error_by_sample))
        MSE_raw = tf.reduce_mean(tf.square(error_by_sample))

        # Unused, but a demonstration of reasonable loss functions:
        self.MSPE = build_loss(MSPE_raw, self.loss_indicator)
        self.MSE = build_loss(MSE_raw, self.loss_indicator)

        # loss function used that takes into account both raw and percent loss
        MSPE_MSE_geometric_mean = tf.sqrt(MSPE_raw * MSE_raw)
        self.loss = build_loss(MSE_raw, self.loss_indicator)

        MAPE_raw = tf.reduce_mean(tf.abs(percent_error_by_sample))
        self.MAPE = build_loss(MAPE_raw, self.loss_indicator)

        opt_fcn = tf.train.AdamOptimizer()

        def apply_clipped_optimizer(opt_fcn,
                                    loss,
                                    clip_norm=.1,
                                    clip_single=.03,
                                    clip_global_norm=False):
            gvs = opt_fcn.compute_gradients(loss)

            if clip_global_norm:
                gs, vs = zip(*[(g, v) for g, v in gvs if g is not None])
                capped_gs, grad_norm_total = tf.clip_by_global_norm(
                    [g for g in gs], clip_norm)
                capped_gvs = list(zip(capped_gs, vs))
            else:
                grad_norm_total = tf.sqrt(
```

```
                    tf.reduce_sum([
                        tf.reduce_sum(tf.square(grad)) for grad, var in gvs
                        if grad is not None
                    ]))
                capped_gvs = [(tf.clip_by_value(grad, -1 * clip_single,
                                                clip_single), var)
                              for grad, var in gvs if grad is not None]
                capped_gvs = [(tf.clip_by_norm(grad, clip_norm), var)
                              for grad, var in capped_gvs if grad is not None]

            optimizer = opt_fcn.apply_gradients(
                capped_gvs, global_step=tf.train.get_global_step())

            return optimizer, grad_norm_total

        self.optimizer, self.grad_norm_total = apply_clipped_optimizer(
            opt_fcn, self.loss)

        self.sess = tf.Session()
        self.sess.run(tf.global_variables_initializer())

        self.trn_losses = []
        self.val_losses = []
```

minimize the loss function

```
    def train_one_epoch(self, X_train, Y_train, sold_train, bs):
        # Train an epoch
        trn_loss = []
        # Randomly shuffle data and prepare for training
        trn_samples = X_train.shape[0]
        order = np.arange(trn_samples)
        np.random.shuffle(order)
        num_batches = (trn_samples // bs) + 1
        for itr in range(trn_samples // bs):
            rows = order[itr * bs:(itr + 1) * bs]
            if itr + 1 == num_batches:
                rows = order[itr * bs:]
            X_active, Y_active, Sold_active = [
                mat[rows] for mat in [X_train, Y_train, sold_train]
            ]
            feed_dict = {
                self.x: X_active,
                self.y: Y_active,
                self.sold: Sold_active
```

```
            }
            _, loss = self.sess.run([self.optimizer, self.loss], feed_dict)
            trn_loss.append(loss)
        self.trn_losses.append(np.mean(trn_loss))
```

gistfile1.txt hosted with ♡ by GitHub view raw

We validate our performance by checking our MAPE on our validation dataset.

```
    def validate(self, X_test, Y_test, sold_test, bs=512):
        val_samples = X_test.shape[0]
        num_batches = val_samples // bs
        order = np.arange(val_samples)
        val_loss_total = 0
        yhat_all = np.zeros(0)
        for itr in range(num_batches):
            rows = order[itr * bs:(itr + 1) * bs]
            if itr + 1 == num_batches:
                rows = order[itr * bs:]
            X_active, Y_active, Sold_active = [
                mat[rows] for mat in [X_test, Y_test, sold_test]
            ]
            feed_dict = {
                self.x: X_active,
                self.y: Y_active,
                self.sold: Sold_active
            }
            val_loss, yhat = self.sess.run([self.loss, self.output], feed_dict)
            yhat_all = np.concatenate((yhat_all, yhat), 0)
            val_loss_total += (val_loss * Sold_active.sum())

        val_loss_avg = val_loss_total / sold_test.sum()

        self.val_losses.append(val_loss_avg)
        self.r2_scores.append(r2_score(Y_test, yhat_all))
        return yhat_all
```

gistfile1.txt hosted with ♡ by GitHub view raw

Training is the process of training for one epoch and validating our performance. We call this process many times to learn from the dataset and track our performance.

```
    def train(self, Xtrn, Xval, Ytrn, Yval, Soldtrn, Soldval, epochs, bs=512):
        # Everything is set. Now train and validate
        for epoch in range(epochs):
            # run one epoch train and validation
            self.train_one_epoch(Xtrn, Ytrn, Soldtrn, bs)
```

```
            yhat = self.validate(Xval, Yval, Soldval)

            if (epoch % 3 == 0) or (epoch == epochs - 1):
                # Occasionally print to command line to inspect performance
                print('epoch:', epoch, 'train loss: ', self.trn_losses[-1],
                      'val loss: ', self.val_losses[-1], 'r2_score:',
                      self.r2_scores[-1])
        return yhat
```

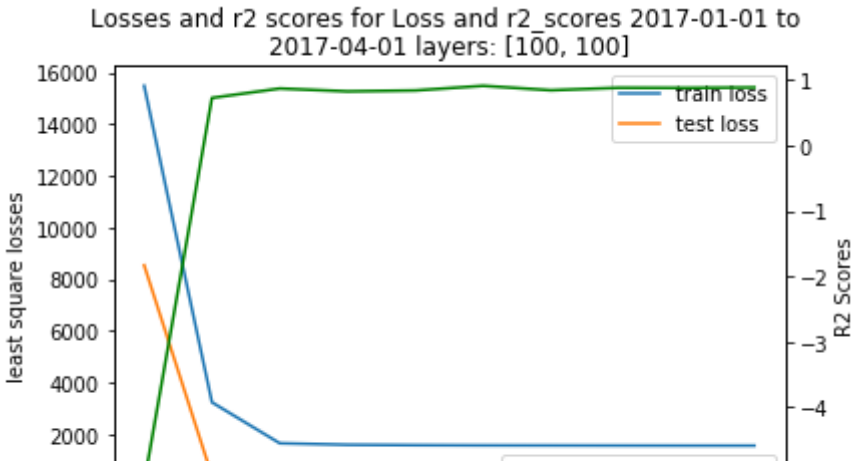
gistfile1.txt hosted with ♡ by GitHub view raw

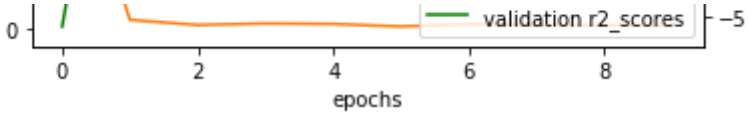
Finally, lets visualize the models performance by tracking training and validation loss, as well as r2 scores between predictions and ground truths.

```
    def visualize(self, name, fname=None):
        # Visualize training and validation losses and r2 scores on one plot
        _, ax1 = plt.subplots()
        ax2 = ax1.twinx()
        ax1.plot(self.trn_losses, label='train loss')
        ax1.plot(self.val_losses, label='test loss')
        ax2.plot(self.r2_scores, label='validation r2_scores', color='g')
        ax1.set_xlabel('epochs')
        ax1.set_ylabel('least square losses')
        ax2.set_ylabel('R2 Scores')
        ax2.legend()
        ax1.legend()
        plt.title('Losses and r2 scores for ' + name)
        if fname is not None:
            plt.savefig(imagedir + '/' + fname + '.jpg')
        plt.show()
```

gistfile1.txt hosted with ♡ by GitHub view raw

The above produces graphs like the following, with epochs of training on the x axis, and y-axes for losses and r2 scores:





Sample of what our validation function creates

Training the Model

Great! Now, finally, lets train on a training set defined up to a point, and validate on the following period. We'll use the code defined above to split the dataset and to train the model. At the end, let's return our final performance so we can track during walk forward optimization.

```
def trn_validate(df, trn_end, val_end, layer_sizes):
    df_train = split_at(df, end_date=trn_end)
    df_val = split_at(df, start_date=trn_end, end_date=val_end)

    numeric_features_train, scaling_mean, scaling_std = scale(df_train)
    numeric_features_val, _, _ = scale(df_val, scaling_mean, scaling_std)
    nan_trn = numeric_features_train.isnull()
    nan_val = numeric_features_val.isnull()
    imputer = build_imputer(numeric_features_train)

    trn_imputed = apply_imputer(imputer, numeric_features_train)
    val_imputed = apply_imputer(imputer, numeric_features_val)

    trn_imputed = pd.concat((trn_imputed, nan_trn.astype(float)), 1)
    val_imputed = pd.concat((val_imputed, nan_val.astype(float)), 1)

    trn_sold, trn_Y = build_Y(df_train, trn_end)

    val_sold, val_Y = build_Y(df_val, dataset_end)

    n_features = trn_imputed.shape[1]

    model = Model(n_features, layer_sizes=layer_sizes)
    yhat = model.train(
        trn_imputed.values,
        val_imputed.values,
        trn_Y.values,
        val_Y.values,
        trn_sold.values,
        val_sold.values,
        epochs=10)
    model.visualize(
        'Loss and r2_scores ' + str(trn_end)[:10] + ' to \n' +
        str(val_end)[:10] + ' layers: ' + str(layer_sizes)
```

```
        str(trn_end)[:7] + ':' + str(val_end)[:7])
    return model.val_losses[-1], yhat, val_Y
```

Summary: What Have we Done?

1. We split the data,

2. Learned to scale and impute the data based on the training set,

3. applied the scaling and imputing to the validation dataset,

4. defined our model, trained our model, and

5. visualized the training performance.

All by calling the functions we defined previously.

. . .

Walk Forward Optimization

Now that we've trained and validated, let's walk forward through the dataset, training and validating on subsequent periods.

```
def walkforward_optimization(df, FIRST_CUTOFF, LAST_CUTOFF, months_per_val=3):
    trn_end = FIRST_CUTOFF
    val_end = FIRST_CUTOFF + pd.DateOffset(months=months_per_val)
    validation_losses = pd.DataFrame()
    idx = 0
    yhat_all, val_Y_all = np.zeros(0), np.zeros(0)
    while val_end <= LAST_CUTOFF:
        idx += 1
        validation_loss, yhat, val_Y = trn_validate(df, trn_end, val_end,
                                                    [100, 100])
        yhat_all = np.concatenate((yhat_all, yhat.flatten()), 0)
        val_Y_all = np.concatenate((val_Y_all, val_Y), 0)
        validation_losses.loc[idx, 'cutoffs'] = val_end
        validation_losses.loc[idx, 'losses'] = validation_loss
        trn_end = trn_end + pd.DateOffset(months=months_per_val)
        val_end = val_end + pd.DateOffset(months=months_per_val)
    return validation_losses, yhat_all, val_Y_all
```

gistfile1.txt hosted with ♡ by GitHub view raw

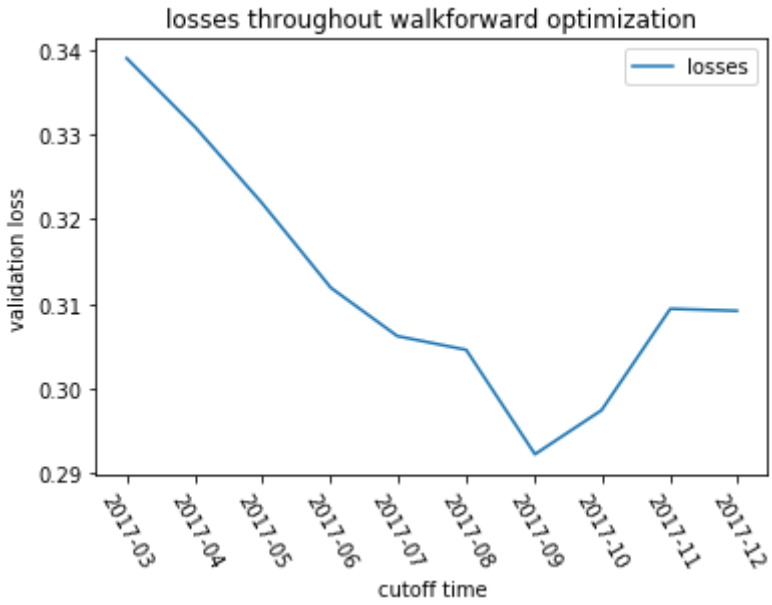
Lastly, let's visualize the walk forward optimization performance throughout time.

```
def visualize_walkforward_optimization_performance(validation_losses):
    plt.plot(validation_losses['cutoffs'], validation_losses['losses'])
    plt.xticks(rotation=-60)
    plt.legend()
    plt.xlabel('cutoff time')
    plt.ylabel('validation loss')
    plt.title('losses throughout walkforward optimization')

    plt.savefig(imagedir + '/walkforward.jpg')
    plt.show()
```

gistfile1.txt hosted with ♡ by GitHub view raw

This produces curves about the final validation performance over walk forward optimization cuts.



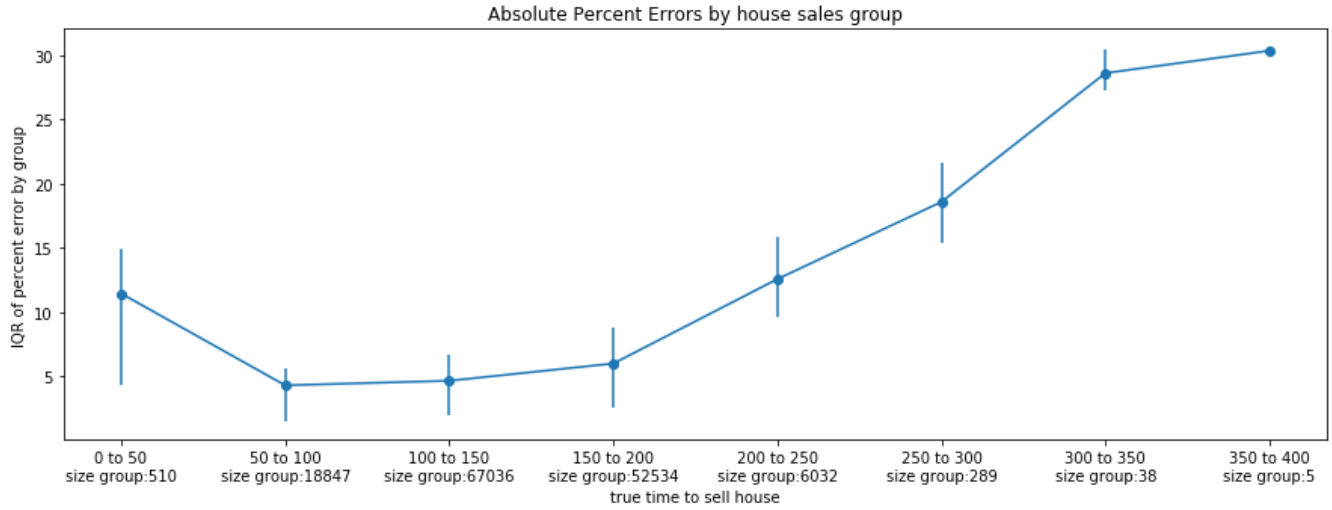
Finally, below is the code to run the full model from the first to the last cutoff, as defined at the beginning of the code section.

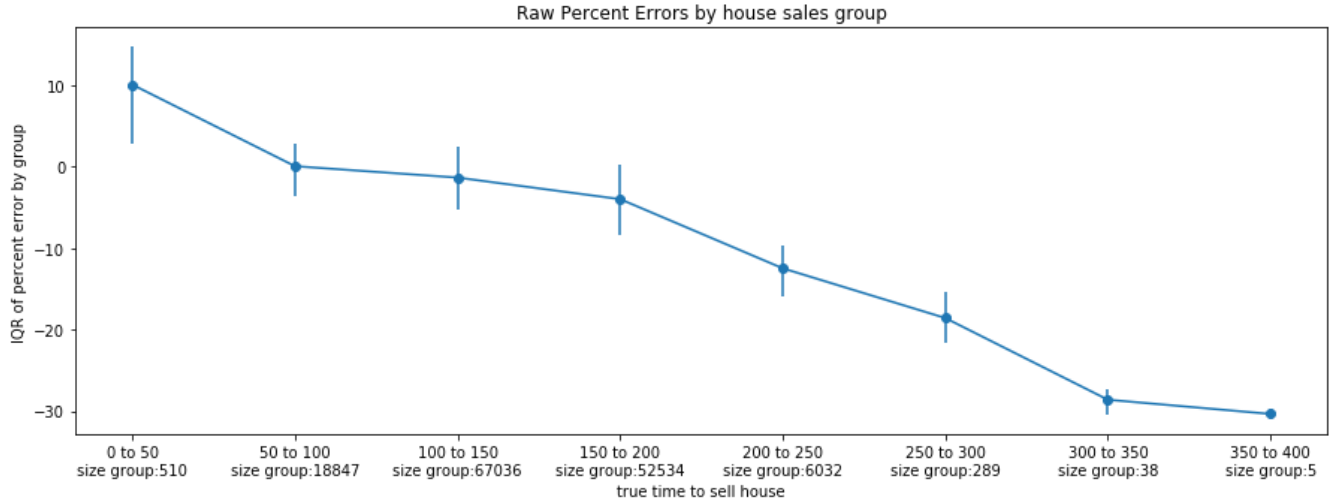
```
validation_losses = walkforward_optimization(df, FIRST_CUTOFF, LAST_CUTOFF)
visualize_walkforward_optimization_performance(validation_losses)
```

gistfile1.txt hosted with ♡ by GitHub view raw

Plotting Performance By Group

Now we choose to group by sales duration, and look at how our error percentages vary within group. To illustrate the final output we want, that we will derive below, we present the graphs below:





The images were drawn, one image per line, by the following function calls

```
plot_by_group(yhat_all, val_Y_all)
plot_by_group(yhat_all, val_Y_all, metric='raw')
```

gistfile1.txt hosted with ♡ by GitHub view raw

Finally, below, we present the code that generated the above images.

```
def plot_by_group(yhat_all, val_Y_all, increment=50, metric='abs'):

    # Either plot absolute error by group or directional error by group.
    if metric == 'abs':
        err = np.abs((yhat_all - val_Y_all) / val_Y_all)
```

```
        title = "Absolute Percent Errors by house sales group"
    else:
        err = (yhat_all - val_Y_all) / val_Y_all
        title = "Raw Percent Errors by house sales group"

    # group by distances of increment
    true_bin = ((val_Y_all // increment) + 1) * increment
    err_bins = {}

    # Place errors into appropriate bins
    for pe, tb in zip(err, true_bin):
        if not tb in err_bins.keys():
            err_bins[tb] = []
        err_bins[tb] += [pe * 100]

    # Sort keys (bins) and organize error lists
    err_list = [err_bins[key] for key in sorted(err_bins)]

    #Calculate means and Interquartile ranges
    medians = [np.mean(err_list[idx]) for idx in range(len(err_list))]
    q25s = [np.percentile(err_list[idx], 25) for idx in range(len(err_list))]
    q75s = [np.percentile(err_list[idx], 75) for idx in range(len(err_list))]

    keys = sorted(err_bins)

    # create blank graph and graph the axis variable
    _, ax = plt.subplots(figsize=(15, 5))

    # label axes
    ax.set(
        xlabel='true time to sell house',
        ylabel='IQR of percent error by group',
        title=title)

    # Add std deviation bars to the plot
    ax.errorbar(
        keys,
        medians,
        yerr=[[medians[i] - q25s[i] for i in range(len(medians))],
              [q75s[i] - medians[i] for i in range(len(medians))]],
        fmt='-o')

    # set where xticks should occur
    ax.set_xticks(keys)

    # Label the x-ticks
    xticks = [
        str(key - 50)[:-2] + ' to ' + str(key)[:-2] + '\n' + 'size group:' +
```

```
        str(len(err_list[idx])) for idx, key in enumerate(sorted(err_bins))
    ]
    ax.set_xticklabels(xticks)

    plt.show()
```

Wrap-Up

We did it! We prepared our dataset using a training/validation split, normalizing the input data, imputing the missing values, fitting a model to the training dataset, validating on the validation dataset, and doing this whole structure within a walk forward optimization loop! Woot!

So now we're ready and feasibly, we could train our model on our full dataset and launch it into the world, and we have an expectation of how well it would do on future classifications (according to the above curve it would have a Mean Absolute Percent Error of around 30%).

Preface if you got a little lost, skimmed through this article, and forgot why we're using Machine Learning and what we're talking about: https://medium.com/@leetandata/machine-learning-preface-ba69bca4701d

Math heavy follow up: https://medium.com/@leetandata/neural-network-introduction-for-software-engineers-1611d382c6aa

Comprehensive source for this post:
https://github.com/leedtan/LeeTanData/blob/master/SimpleML_HouseSales/HousePredictions.ipynb

Machine Learning