architecture : Combination of tabular, textual and image inputs to predict house prices.

Dave Cote, M.Sc. · Follow

10 min read · Nov 19, 2022

Modality refers to a certain type of information and/or the representation format in which information is stored.

According to wikipedia, « ***Multimodal learning attempts to model the combination of different modalities of data, often rising in real-world applications. An example of joint data is combining text (typically represented as discrete word count vectors) with imaging data consisting of pixel intensities and annotation tags. As these modalities have fundamentally different statistical properties, combining them is non-trivial, which is why specialized modelling strategies and algorithms are required.*** »

Can we simultaneously train structured (tabular) and unstructured (text, image) data in the same neural network model while optimizing the same output target ?

We'll perform this experiment using a very simple example by implementing a « concat » combination method that will concatenate text features, tabular features, and image features all at once before our final regressor layer(s).

Specifically, we will build a multimodal neural network that contains 3 input heads :

- **Image :** Inception V3 input layers (Transfer Learning from Imagenet)
- **Text :** Bidirectionnal LSTM with Attention layers pretrained on GlOve embeddings
- **Structured :** Multi-layers perceptron

And 1 output head :

- **House Price**

Of course, more advanced methods and strategies exist to solve multimodal problems. For example, the great HuggingFace offers a **multimodal-toolkit** that you can experiment with.

Table of contents

Dataset

First, let's take a quick look at the dataset we're going to use. I searched a very long time to find a dataset containing images, text and tabular data and

finally found one! The link to download the dataset is here: **https://www.kaggle.com/datasets/ericpierce/austinhousingprices** (You must have a Kaggle account to download it).

As it is written on the Kaggle page:

"Context — Austin Housing

This data was originally sourced in support of my capstone project at Northwestern University. The Austin Housing market is one of the hottest markets in 2021, and these listings show how that market has changed over the past couple of years."

The data contains a csv file with all the tabular features, a text description feature and a column that specify the image filename associated with the dataset row. The csv dataset comes with a huge zipped file containing the house images. Unfortunatly, there is only one image available for each house but we will work with that.

The goal here is to predict house **price** using all the data we have.

Tabular data engineering

Although it is said that the data has been cleaned, there are still many data quality issues left. For example, there are impossible feature values like houses with 0 bathrooms or 0 bedrooms or negative square foot of living area.

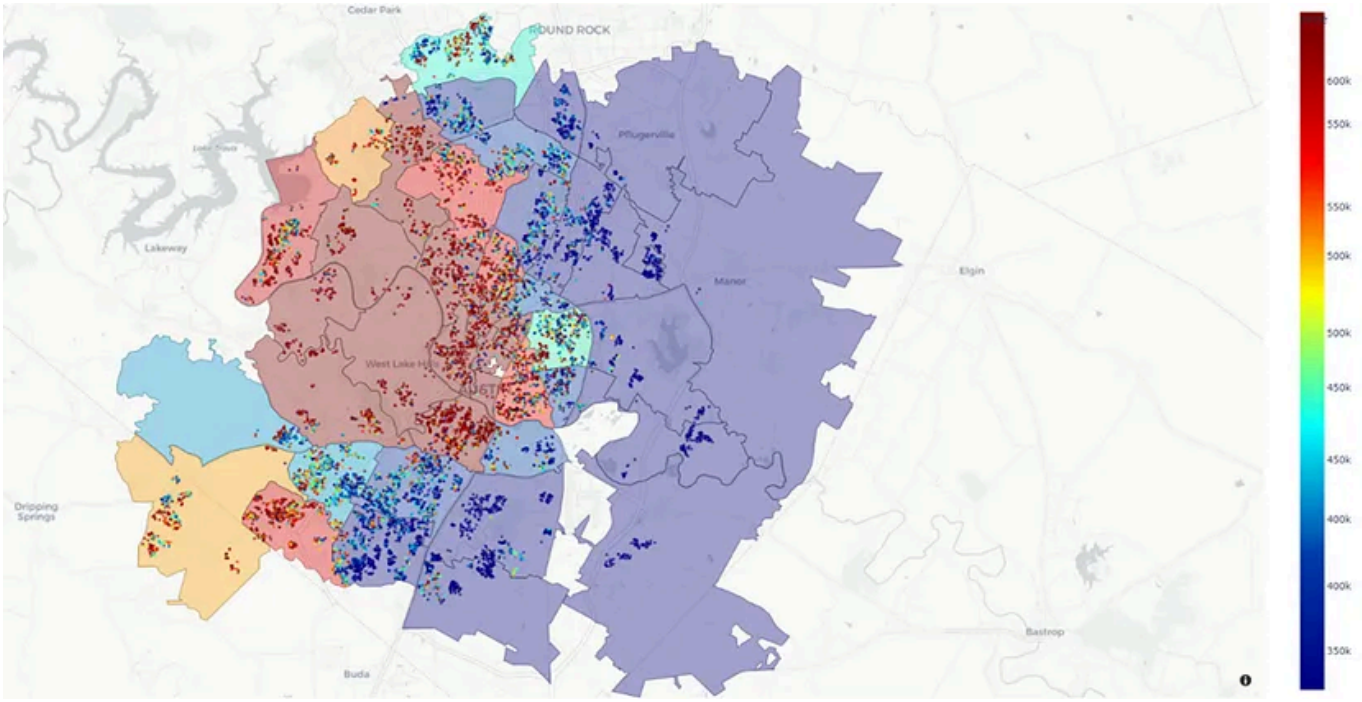
Here is a table that resume some of the major data transformations and cleaning steps performed:

Feature name	Quality problem / Opportunity	Transformation
numOfBathrooms	Impossible to have 0 bathrooms	> 1989= 2, <= 1989 = 1
numOfBedrooms	Impossible to have 0 bedroms	> 1989= 2.5, <= 1989 = 2
garageSpaces	More than 3 garageSpaces and price < 1M$ and homeType = Single Family	max = 3
parkingSpaces	More than 3 parkingSpaces and price < 1M$ and homeType = Single Family	max = 3
lotSizeSqFt	lotSizeSqFt <= 0, outliers > IQR*1.6, outliers < IQR*1.6	delete
livingAreaSqFt	livingAreaSqFt <= 0, outliers > IQR*1.6, outliers < IQR*1.6	delete
latestPrice (target)	Concept Drift (Price Inflation)	Adjust past price with inflation
latitude	Outliers < 30.12	delete
zipcode	Outiliers outside Austin	delete
numOfAccessibilityFeatures	Few when > 0	convert to binary, > 0
numOfPatioAndPorchFeatures	Few when > 0	convert to binary, > 0
numOfSecurityFeatures	Few when > 0	convert to binary, > 0
numOfWaterfrontFeatures	Few when > 0	convert to binary, > 0
numOfWindowFeatures	Few when > 0	convert to binary, > 0
numOfCommunityFeatures	Few when > 0	convert to binary, > 0
streetAddress	Unexploitable text format	Extract type of address (drive, lane, street, cove, road, avenue, trail, circle, court, way, other)
description	Dirty text description	Clean text (normalization, stop words, lemmatization...)
longitude, latitude	Opportunity: Geospatial clusters	Create clusters with Kmeans
zipcode	Opportunity: Census data	Map census data (crime_reports, median_household_income($), population_below_poverty_level(%), unemployment(%)...)

An important step is to clean up our target value, **house prices**, using inflation rates (concept drift problem). Past selling prices do not match actual selling prices due to inflation; the definition of 1$ unit is not the same over time. For this purpose, I used a table of inflation rates to correct the price amount.

Once cleaned, our dataset contains **56 features** and **10537 rows.**

Here is a geospatial plot (plotly) of all Austin Houses we have:



We can see that there are some zipcode clusters where the price is higher (red) than other regions (blue). I've created a new feature that represent the zipcode clusters called **regionCluster** using the kmeans algorithm:

```
from sklearn.cluster import KMeans

kmeans = KMeans(n_clusters=25, random_state=0).fit(data[['longitude', 'latitude']])
data['regionCluster'] = kmeans.predict(data[['longitude', 'latitude']])
```

creating_region_cluster.py hosted with ❤ by GitHub view raw

Text cleaning

To clean up house text descriptions, we apply normalization, remove stop words and find the root of each word using lemmatization.

Here is an example of the result:

```
data_text_raw_vs_clean[data_text_raw_vs_clean.index == 0]['description'].values[0]
```

```
"Absolutely GORGEOUS 4 Bedroom home with 2 full baths located in the tucked away Pflugerville community The Lakes at Northtown. This home was r
ecently updated with new carpets, new tile kitchen back splash, and fresh paint throughout the entire house. This charming abode is a MUST SEE
if you're looking for an affordable home in the Austin area. Something very hard to come by in this market!  Only 14 miles from downtown, this
home will accommodate downtown commuting as well as a work from home environment. This community is tucked away and yet only minutes from major
retailers and shopping. Do not miss this opportunity!"
```

```
data_text_raw_vs_clean[data_text_raw_vs_clean.index == 0]['description_clean'].values[0]
```

```
' absolutely gorgeous locate tuck away community lake northtown recently update new carpet new tile kitchen splash fresh paint entire house cha
rming abode yoube look affordable area hard come market mile downtown accommodate downtown commuting as as work environment community tuck away
minute major retailer shopping miss opportunity'
```

Image cleaning — Fine-tune VGG16 to predict good/bad images

This one is a bit tricky. After randomly exploring some images, there seem to be very different / heterogenous types of house images. Here are a few:

Nice picture!

Nice river but... which house are we talking about?

Not very useful...

Interior picture...

Would have a fresh beer on that patio but not very useful...

What is this!!? A cemetery? ...

As you can see, there are several heterogeneous types of house images. Another challenge is that we only have one image available per house, so the image must be useful.

Since there are about 10,000 images, we can't look at all of them to decide which data to keep and which data to delete. To help us, we will fine-tune a **VGG16** to predict good/bad images for us.

First, we need to determine what is a good image for us. I decided to keep only images where you can see the complete architecture of the house from the outside:

Bad images would be any interior image or any image where we cannot see the full architecture of the house from the outside.

As we are using **transfer learning**, fewer samples from each class should do the job.

We need to create two directories, one for the good images and one for the bad ones, then manually dispatch some of them in the right place. After an hour, I sorted around **800 images** for the train set (400 good and 400 bad) and around **250 images** for the test set.

We are now ready to fine-tune our VGG16 model!

Python Code:

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

import os
import glob
import datetime
import pickle
# keras imports
from keras.applications.vgg16 import VGG16, preprocess_input
from keras.preprocessing import image
from keras.models import Model
from keras.layers import Layer
from keras import backend as K
# other imports
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
import numpy as np
import h5py
import json
import time

# filter warnings
import warnings
warnings.simplefilter(action="ignore", category=FutureWarning)

def make_confusion_matrix(cf,
                          group_names=None,
                          categories='auto',
                          count=True,
                          percent=True,
                          cbar=True,
                          xyticks=True,
                          xyplotlabels=True,
                          sum_stats=True,
                          figsize=None,
                          cmap='Blues',
                          title=None):
    '''
    This function will make a pretty plot of an sklearn Confusion Matrix cm using a Seaborn heat
    Arguments
    ---------
```

```
    cf:            confusion matrix to be passed in
    group_names:   List of strings that represent the labels row by row to be shown in each squa
    categories:    List of strings containing the categories to be displayed on the x,y axis. De
    count:         If True, show the raw number in the confusion matrix. Default is True.
    normalize:     If True, show the proportions for each category. Default is True.
    cbar:          If True, show the color bar. The cbar values are based off the values in the
                   Default is True.
    xyticks:       If True, show x and y ticks. Default is True.
    xyplotlabels:  If True, show 'True Label' and 'Predicted Label' on the figure. Default is Tr
    sum_stats:     If True, display summary statistics below the figure. Default is True.
    figsize:       Tuple representing the figure size. Default will be the matplotlib rcParams v
    cmap:          Colormap of the values displayed from matplotlib.pyplot.cm. Default is 'Blues
                   See http://matplotlib.org/examples/color/colormaps_reference.html

    title:         Title for the heatmap. Default is None.
    '''


    # CODE TO GENERATE TEXT INSIDE EACH SQUARE
    blanks = ['' for i in range(cf.size)]

    if group_names and len(group_names)==cf.size:
        group_labels = ["{}\n".format(value) for value in group_names]
    else:
        group_labels = blanks

    if count:
        group_counts = ["{0:0.0f}\n".format(value) for value in cf.flatten()]
    else:
        group_counts = blanks

    if percent:
        group_percentages = ["{0:.2%}".format(value) for value in cf.flatten()/np.sum(cf)]
    else:
        group_percentages = blanks

    box_labels = [f"{v1}{v2}{v3}".strip() for v1, v2, v3 in zip(group_labels,group_counts,group_
    box_labels = np.asarray(box_labels).reshape(cf.shape[0],cf.shape[1])


    # CODE TO GENERATE SUMMARY STATISTICS & TEXT FOR SUMMARY STATS
    if sum_stats:
        #Accuracy is sum of diagonal divided by total observations
        accuracy  = np.trace(cf) / float(np.sum(cf))
```

```
90
91          #if it is a binary confusion matrix, show some more stats
92          if len(cf)==2:
93              #Metrics for Binary Confusion Matrices
94              precision = cf[1,1] / sum(cf[:,1])
95              recall    = cf[1,1] / sum(cf[1,:])
96              f1_score  = 2*precision*recall / (precision + recall)
97              stats_text = "\n\nAccuracy={:0.3f}\nPrecision={:0.3f}\nRecall={:0.3f}\nF1 Score={:0.
98                  accuracy,precision,recall,f1_score)
99          else:
100             stats_text = "\n\nAccuracy={:0.3f}".format(accuracy)
101     else:
102         stats_text = ""
103
104
105     # SET FIGURE PARAMETERS ACCORDING TO OTHER ARGUMENTS
106     if figsize==None:
107         #Get default figure size if not set
108         figsize = plt.rcParams.get('figure.figsize')
109
110     if xyticks==False:
111         #Do not show categories if xyticks is False
112         categories=False
113
114
115     # MAKE THE HEATMAP VISUALIZATION
116     plt.figure(figsize=figsize)
117     sns.heatmap(cf,annot=box_labels,fmt="",cmap=cmap,cbar=cbar,xticklabels=categories,yticklabel
118
119     if xyplotlabels:
120         plt.ylabel('True label')
121         plt.xlabel('Predicted label' + stats_text)
122     else:
123         plt.xlabel(stats_text)
124
125     if title:
126         plt.title(title)
127
128
129
130
131 config={
132   "model"           : "vgg16",
133   "weights"         : "imagenet",
134   "include_top"     : False,
```

```
135
136     "train_path"      : "/data/classifier_cleaner/train_house/",
137     "test_path"       : "/data/classifier_cleaner/test_house/",
138     "features_path"   : "/data/classifier_cleaner/features.h5",
139     "labels_path"     : "/data/classifier_cleaner/labels.h5",
140     "results"         : "/data/classifier_cleaner/results.txt",
141     "classifier_path" : "/data/classifier_cleaner/classifier.pickle",
142     "model_path"      : "/data/classifier_cleaner/model",
143     "label_encoder_path"      : "/data/classifier_cleaner/label_encoder.pickle",
144
145     "test_size"       : 0.20,
146     "seed"            : 9,
147     "num_classes"     : 2
148   }
149
150
151
152   # config variables
153   model_name    = config["model"]
154   weights     = config["weights"]
155   include_top   = config["include_top"]
156   train_path    = config["train_path"]
157   features_path   = config["features_path"]
158   labels_path   = config["labels_path"]
159   test_size     = config["test_size"]
160   results     = config["results"]
161   model_path    = config["model_path"]
162   label_encoder_path    = config["label_encoder_path"]
163
164   # Here we are loading the base VGG16 model with weights and then excluding the top dense layer
165   if model_name == "vgg16":
166       base_model = VGG16(weights=weights)#, include_top=True)
167       model = Model(base_model.input, base_model.get_layer('fc1').output)
168       image_size = (224, 224)
169   else:
170       base_model = None
171
172   print ("Successfully loaded base model and model...")
173
174   train_labels = os.listdir(train_path)
175   train_labels
176
177   # path to training dataset
178   #train_labels = os.listdir(train_path)
```

```
train_labels = ['good', 'bad']

# encode the labels
print ("Encoding labels...")
le = LabelEncoder()
le.fit([tl for tl in train_labels])

# variables to hold features and labels
features = []
labels   = []


#print(file_names)
# loop over all the labels in the folder
count = 1
for i, label in enumerate(train_labels):
    cur_path = train_path  + label +  '/'
    relevant_path = cur_path
    included_extensions = ['jpg']
    file_names = [fn for fn in os.listdir(relevant_path)
                  if any(fn.endswith(ext) for ext in included_extensions)]
    #print(cur_path)
    #print(cur_path)
    #print(len(labels))
    count = 1
    for image_path in file_names:

        img = image.load_img(train_path  + label +  '/' + image_path, target_size=image_size)
        x = image.img_to_array(img)
        #print(x)
        x = np.expand_dims(x, axis=0)
        x = preprocess_input(x)
        feature = model.predict(x)
        flat = feature.flatten()
        features.append(flat)
        labels.append(label)
        print ("Processed - " + str(count))
        count += 1
    print ("Completed label - " + label)

# encode the labels using LabelEncoder
le = LabelEncoder()
le_labels = le.fit_transform(labels)

# get the shape of training labels
```

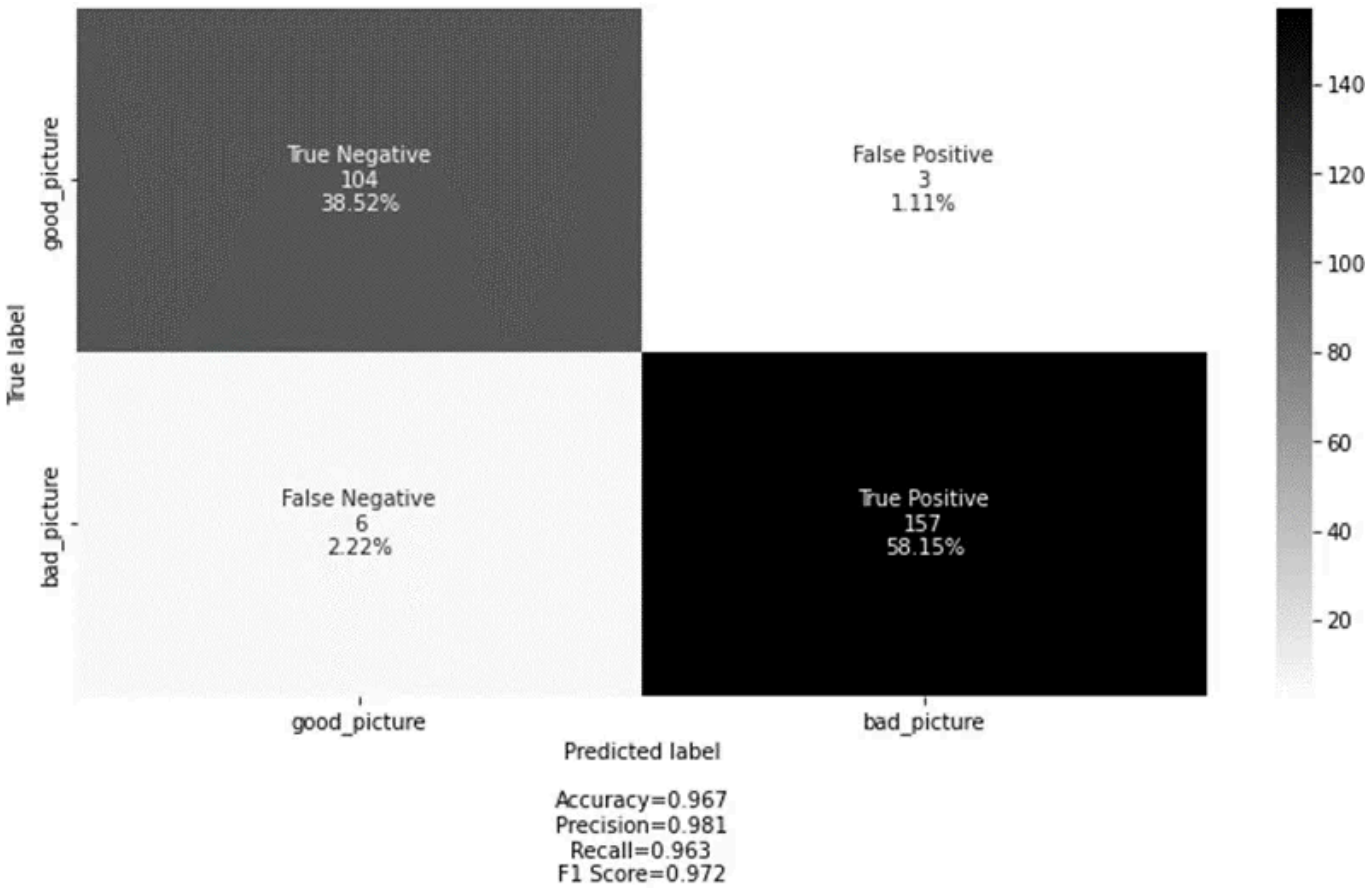
```
print ("Training labels: {}".format(le_labels))
print ("Training labels shape: {}".format(le_labels.shape))

model.summary()

# save features and labels
h5f_data = h5py.File(features_path, 'w')
h5f_data.create_dataset('dataset_1', data=np.array(features))

h5f_label = h5py.File(labels_path, 'w')
h5f_label.create_dataset('dataset_1', data=np.array(le_labels))

h5f_data.close()
h5f_label.close()

# save model and weights
model_json = model.to_json()
with open(model_path + str(test_size) + ".json", "w") as json_file:
    json_file.write(model_json)

pickle.dump(le, open(label_encoder_path, 'wb'))

# save weights
model.save_weights(model_path + str(test_size) + ".h5")
print("Saved model and weights to disk..")

print ("Features and labels saved..")

# end time
end = time.time()
print ("End time - {}".format(datetime.datetime.now().strftime("%Y-%m-%d %H:%M")))



##### PREDICT
config={
  "model"           : "vgg16",
  "weights"         : "imagenet",
  "features_path"   : "/data/classifier_cleaner/features.h5",
  "labels_path"     : "/data/classifier_cleaner/labels.h5",
  "classifier_path" : "/data/classifier_cleaner/classifier.pickle",
  "model_path"      : "/data/classifier_cleaner/model",
  "label_encoder_path"      : "/data/classifier_cleaner/label_encoder.pickle",
  "test_size"       : 0.20
```

```
    "test_size"       : 0.20,
    "seed"            : 9,
}

# config variables
test_size  = config["test_size"]
seed = config["seed"]
features_path = config["features_path"]
labels_path = config["labels_path"]
classifier_path = config["classifier_path"]

# import features and labels
h5f_data  = h5py.File(features_path, 'r')
h5f_label = h5py.File(labels_path, 'r')

features_string = h5f_data['dataset_1']
labels_string   = h5f_label['dataset_1']

features = np.array(features_string)
labels   = np.array(labels_string)

h5f_data.close()
h5f_label.close()

# verify the shape of features and labels
print ("[INFO] features shape: {}".format(features.shape))
print ("[INFO] labels shape: {}".format(labels.shape))

print ("[INFO] training started...")
# split the training and testing data
(trainData, testData, trainLabels, testLabels) = train_test_split(np.array(features),
                                                                  np.array(labels),
                                                                  test_size=test_size,
                                                                  random_state=seed)

print ("[INFO] splitted train and test data...")
print ("[INFO] train data  : {}".format(trainData.shape))
print ("[INFO] test data   : {}".format(testData.shape))
print ("[INFO] train labels: {}".format(trainLabels.shape))
print ("[INFO] test labels : {}".format(testLabels.shape))

# Use logistic regression as the model
print ("[INFO] creating model...")
model = LogisticRegression(max_iter=5500)

```

```
# evaluate the model of test data
preds = model.predict(testData)

# dump classifier to file
print ("[INFO] saving model...")
pickle.dump(model, open(classifier_path, 'wb'))

# plot the confusion matrix
cm = confusion_matrix(testLabels, preds)

categories = ['good_picture', 'bad_picture']
labels = ['True Negative','False Positive','False Negative','True Positive']
make_confusion_matrix(cm, figsize=(12,6),
                      group_names=labels,
                      categories=categories,
                      cmap='binary')
```

vgg16_image_cleaner.py hosted with ❤ by GitHub view raw

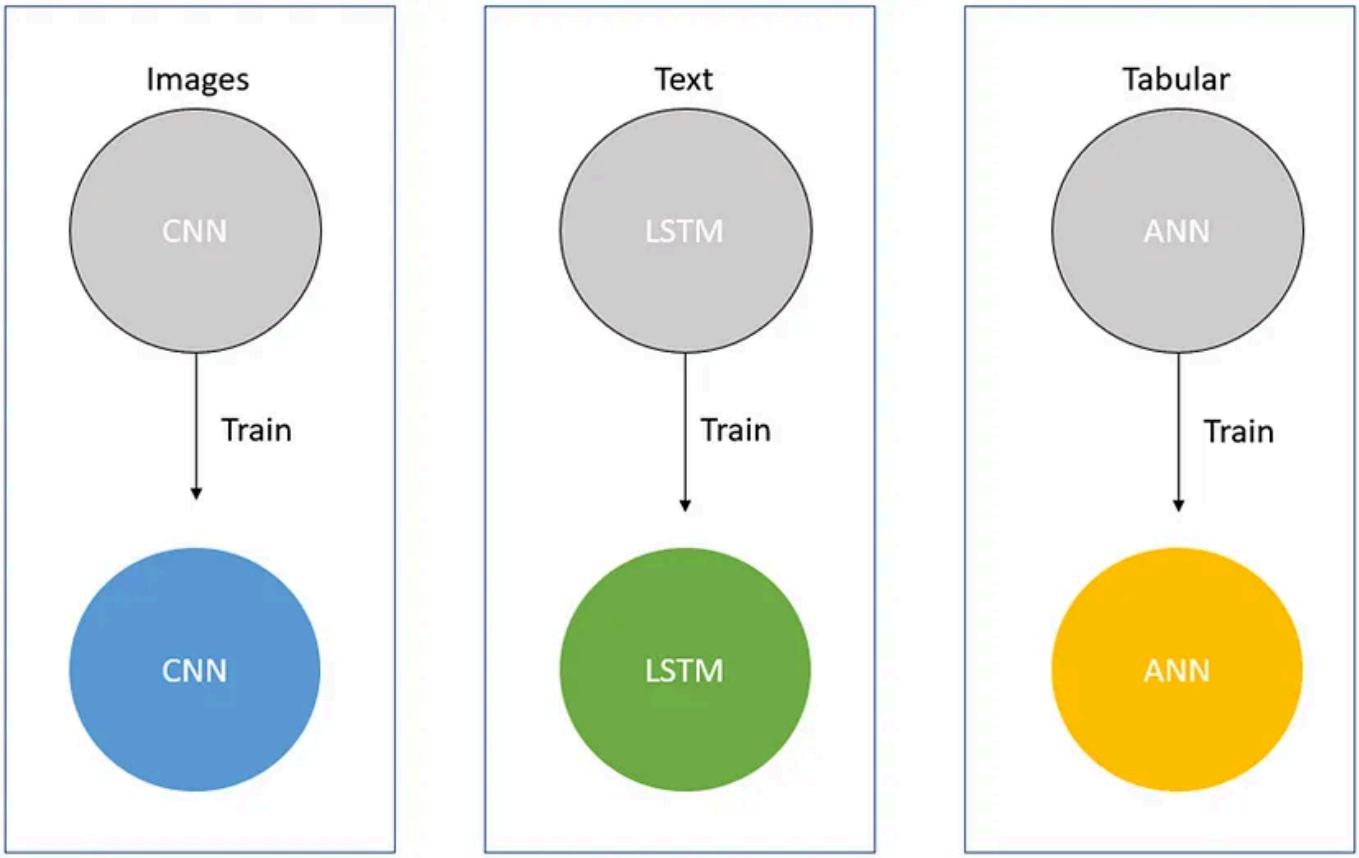
Here is the final **confusion matrix** from the test data sample:



Confusion Matrix from the VGG16 image cleaning test set

We can now predict every image we have using our **“VGG16 cleaner”** and remove all rows of our dataset where the available image has been classified as **“bad”.**

For the next steps, the strategy we will use is to individually train each neural network, save the weights and then load the trained weights into our final hybrid model and retrain with all the parts merged together.



Fine-tuning each neural network architecture individually first

Architecture#1 — Structured Data fine-tuning with MLP:

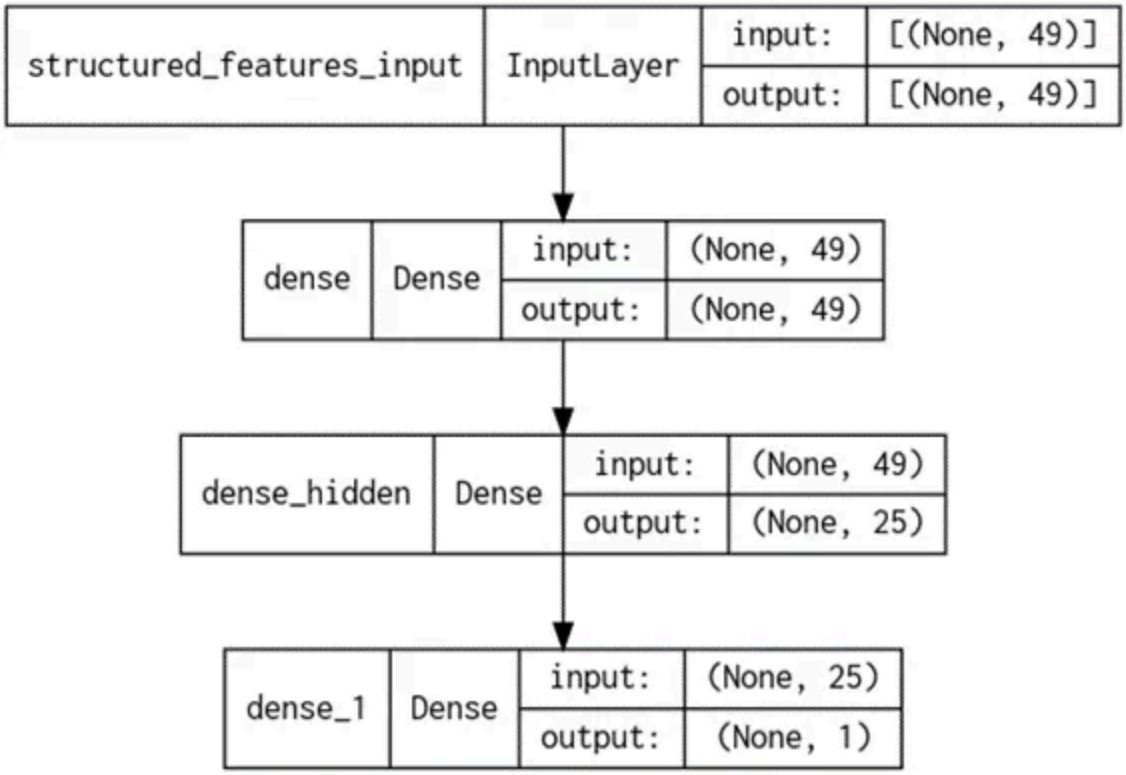
Python code to create the neural network architecture, fine-tune, and save the weights:

```
from tensorflow.keras import layers
from keras.models import Model
from keras.layers import Dense
from keras.callbacks import EarlyStopping, ModelCheckpoint
from tensorflow.keras.optimizers import Adam
from keras.losses import MeanSquaredError
from tensorflow.keras.optimizers.schedules import ExponentialDecay
import pandas as pd
from keras.utils.vis_utils import plot_model
import pickle as pkl
import warnings
warnings.simplefilter(action="ignore", category=FutureWarning)

X_train_structured_std = pd.read_csv('/data/X_train_structured.csv', sep = ';')
X_test_structured_std = pd.read_csv('/data/X_test_structured.csv', sep = ';')

y_train_std = pd.read_csv('/data/y_train_std.csv', sep = ';')
y_test_std = pd.read_csv('/data/y_test_std.csv', sep = ';')


file = open("/data/model/y_numerical_scaler.pkl",'rb')
y_numerical_scaler = pkl.load(file)
file.close()

##### MODEL ARCHITECTURE
structured_input = layers.Input(shape=(X_train_structured_std.shape[1],)
                                , name='structured_features_input'
                               )

x = layers.Dense(X_train_structured_std.shape[1],
                 activation='relu',
                 kernel_initializer='he_normal')(structured_input) #80
dense_hidden = Dense(25, activation='relu', name='dense_hidden')(x)
last_layer = Dense(1, activation='linear')(dense_hidden)


# declare the final model inputs and outputs
final_model = Model(inputs=structured_input, outputs=last_layer)

# print a summary of the model
print(final_model.summary())

# set up learning rate decay schedule
initial_learning_rate = 0.1
lr_schedule = ExponentialDecay(
```

```
    initial_learning_rate,
        decay_steps=100000,
    decay_rate=0.96,
    staircase=True)

stop = EarlyStopping(monitor="val_loss", patience=50, restore_best_weights=True, mode='min', verb
best = ModelCheckpoint(filepath='/data/model/best_structured_model.hdf5', save_best_only=True, mo

# compile the model
final_model.compile(optimizer=Adam(learning_rate=lr_schedule, epsilon=1),
                    loss="mean_squared_error",
                    metrics=[MeanSquaredError()])

plot_model(final_model, show_shapes=True, to_file='model1_structured.png')


results = final_model.fit(
            X_train_structured_std, y_train_std,
            epochs=500,
            callbacks=[stop, best],
            validation_data=([X_test_structured_std, y_test_std])
            )

final_model.save('/data/model/best_structured_model.h5')
```

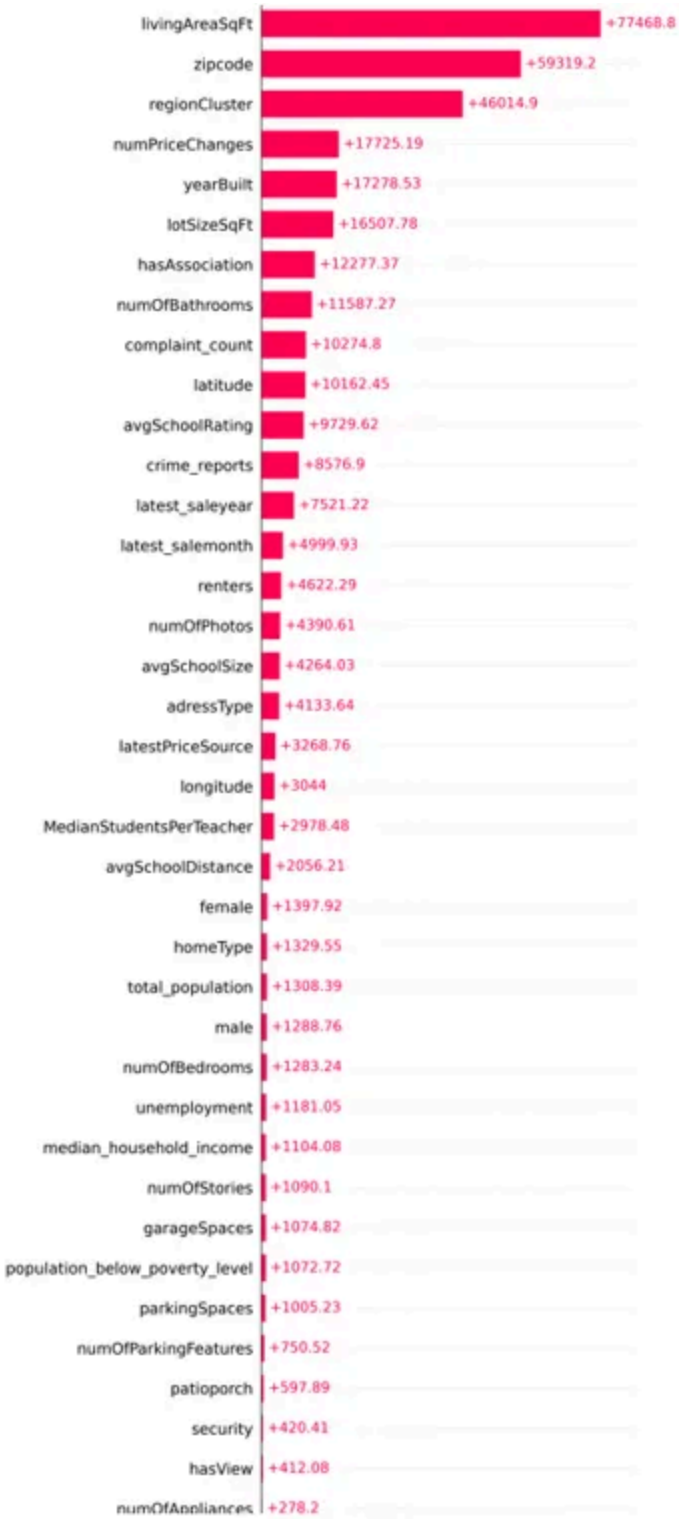
fine_tune_structured_ann.py hosted with ❤ by GitHub view raw

Multi-layer perceptron (MLP) architecture:



Structured Data Neural Netowrk Architecture (MultiLayers Perceptron)

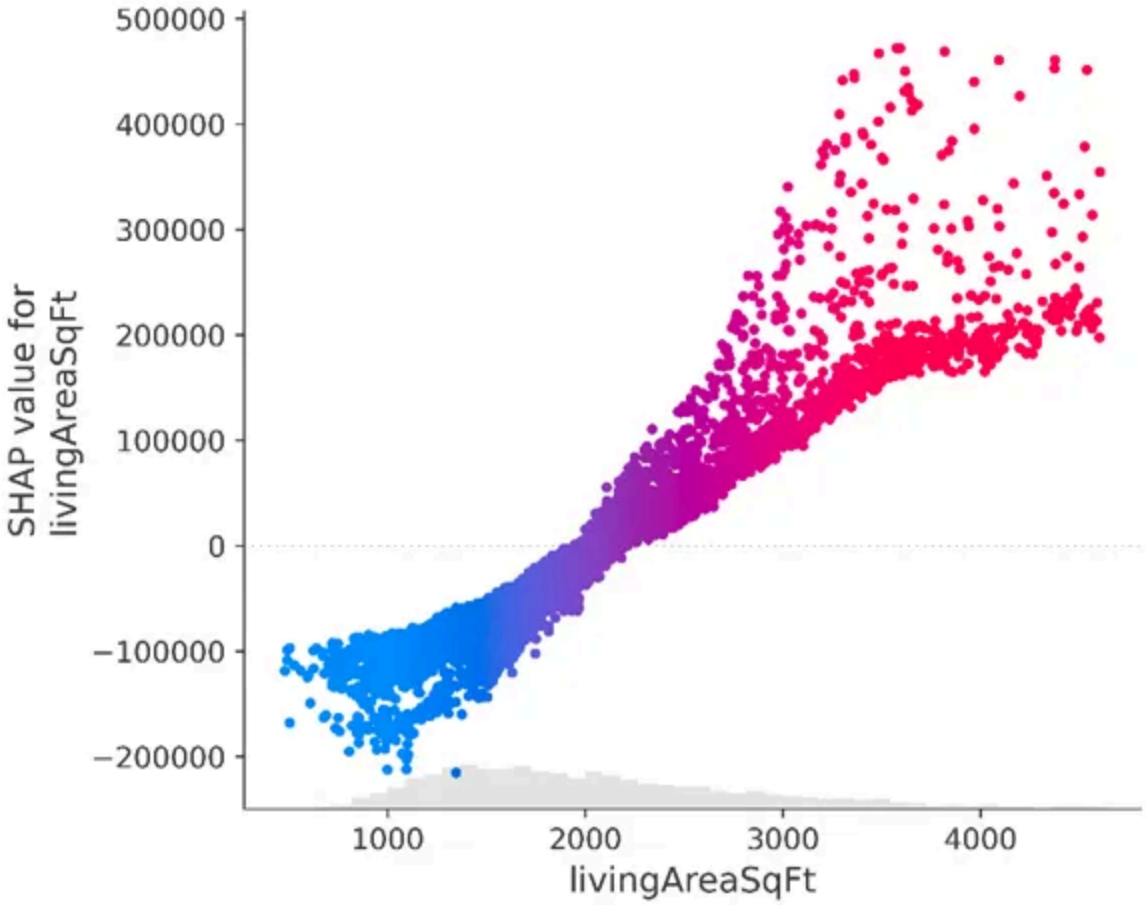
Here are the most important shapley values for our tabular features (Using the DeepLift algorithm (DeepExplainer) from the awesome **SHAP package**) :



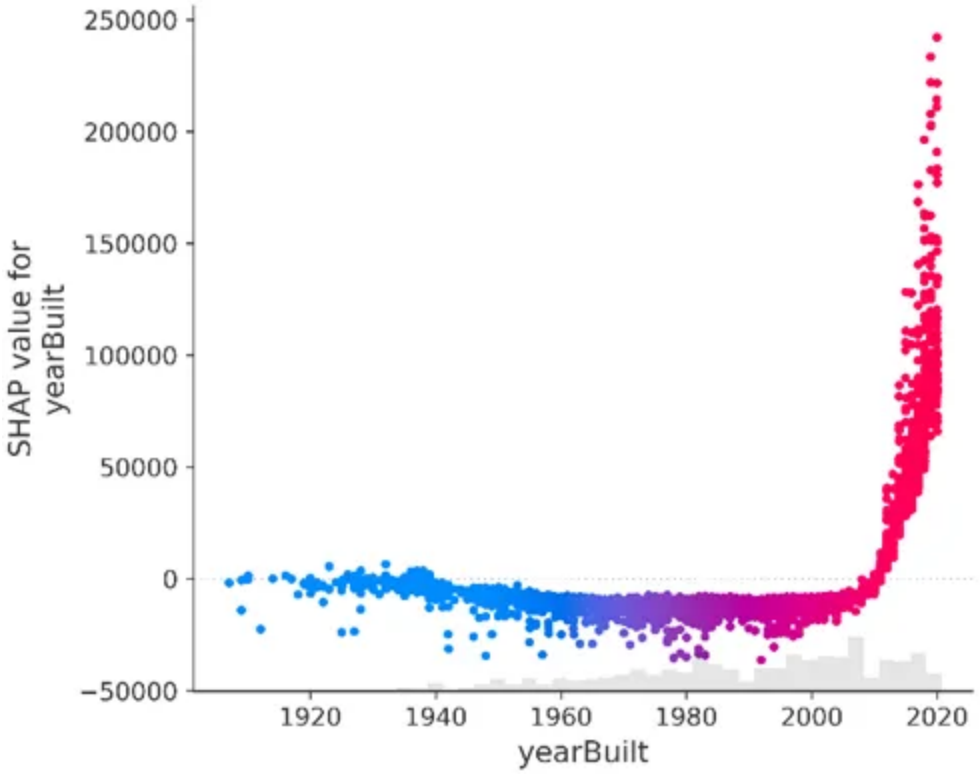
This barplot shows us that the 3 most important features to predict price of our structured data neural network are: **livingAreaSqFt, zipcode** and **regionCluster.**

numPriceChanges, yearBuilt and lotSizeSqFt follow next.

If we look at some of these features:



The biggest **livingAreaSqFt**, the biggest shapley value, the biggest price (seems logical!)



The more recent **yearBuilt**, the biggest shapley value, the biggest price (seems logical!)

Architecture#2 — Text Data fine-tuning GlOve BiLSTM with Attention:

Python code to create the neural network architecture, fine-tune, and save the weights:

```
#Import Libraries

import pandas as pd
import numpy as np
import pickle as pkl

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers

# keras imports
from keras.models import Model


# filter warnings
import warnings
warnings.simplefilter(action="ignore", category=FutureWarning)

from keras.layers import Input, Attention, Embedding, Dropout, Dense , Bidirectional, LSTM
from keras.initializers import Constant
from keras.callbacks import EarlyStopping, ModelCheckpoint
from keras.preprocessing.text import Tokenizer
from tensorflow.keras.optimizers import Adam
from keras.losses import MeanSquaredError
#from keras.utils import plot_model, Sequence
from tensorflow.keras.optimizers.schedules import ExponentialDecay


from keras.utils.vis_utils import plot_model


from keras.preprocessing.sequence import pad_sequences


from category_encoders import *


class TransformerBlock(layers.Layer):
    def __init__(self, embed_dim = 32, num_heads = 2, ff_dim = 32, rate=0.1):
        super(TransformerBlock, self).__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.ff_dim = ff_dim
        self.rate = rate

```

```
        self.att = layers.MultiHeadAttention(num_heads=num_heads, key_dim=embed_dim)
        self.ffn = keras.Sequential(
            [layers.Dense(ff_dim, activation="relu"), layers.Dense(embed_dim),]
        )
        self.layernorm1 = layers.LayerNormalization(epsilon=1e-6)
        self.layernorm2 = layers.LayerNormalization(epsilon=1e-6)
        self.dropout1 = layers.Dropout(rate)
        self.dropout2 = layers.Dropout(rate)

    def get_config(self):

        config = super().get_config().copy()
        config.update({
            'embed_dim': self.embed_dim,
            'num_heads': self.num_heads,
            'ff_dim': self.ff_dim,
            'rate': self.rate,
            'att': self.att,
            'ffn': self.ffn,
            'layernorm1': self.layernorm1,
            'layernorm2': self.layernorm2,
            'dropout1': self.dropout1,
            'dropout2': self.dropout2
        })
        return config

    def call(self, inputs, training):
        attn_output = self.att(inputs, inputs)
        attn_output = self.dropout1(attn_output, training=training)
        out1 = self.layernorm1(inputs + attn_output)
        ffn_output = self.ffn(out1)
        ffn_output = self.dropout2(ffn_output, training=training)
        return self.layernorm2(out1 + ffn_output)

class TokenAndPositionEmbedding(layers.Layer):
    def __init__(self, maxlen = 100, vocab_size = 1000, embed_dim = 125):
        super(TokenAndPositionEmbedding, self).__init__()
        self.maxlen = maxlen
        self.vocab_size = vocab_size
        self.embed_dim = embed_dim
        self.token_emb = layers.Embedding(input_dim=vocab_size, output_dim=embed_dim)
        self.pos_emb = layers.Embedding(input_dim=maxlen, output_dim=embed_dim)

    def get_config(self):
```

```
90
91          config = super().get_config().copy()
92          config.update({
93              'maxlen': self.maxlen,
94              'vocab_size': self.vocab_size,
95              'embed_dim': self.embed_dim,
96              'token_emb': self.token_emb,
97              'pos_emb': self.pos_emb
98          })
99          return config
100
101     def call(self, x):
102         maxlen = tf.shape(x)[-1]
103         positions = tf.range(start=0, limit=maxlen, delta=1)
104         positions = self.pos_emb(positions)
105         x = self.token_emb(x)
106         return x + positions
107
108 def prepare_text_data_train(dataf):
109     for col in dataf.columns:
110         dataf[col] = dataf[col].str.lower()
111
112         tokenizer = Tokenizer(num_words=2096) #5000
113         tokenizer.fit_on_texts(dataf[col])
114         pkl.dump(tokenizer, open('/data/model/text_tokenizer.pkl','wb'))
115
116         vocab_size = len(tokenizer.word_index) + 1
117         print(vocab_size)
118
119         pkl.dump(vocab_size, open('/data/model/text_vocab_size.pkl','wb'))
120
121         maxlen = int(np.mean(dataf[col].str.len()))
122         pkl.dump(maxlen, open('/data/model/text_maxlen.pkl','wb'))
123
124 def prepare_text_data_inference(dataf):
125     for col in dataf.columns:
126         dataf[col] = dataf[col].str.lower()
127
128         file = open("/data/model/text_tokenizer.pkl",'rb')
129         tokenizer = pkl.load(file)
130         file.close()
131
132         data_tokens = tokenizer.texts_to_sequences(dataf[col])
133
134         file = open("/data/model/text vocab size.pkl",'rb')
```

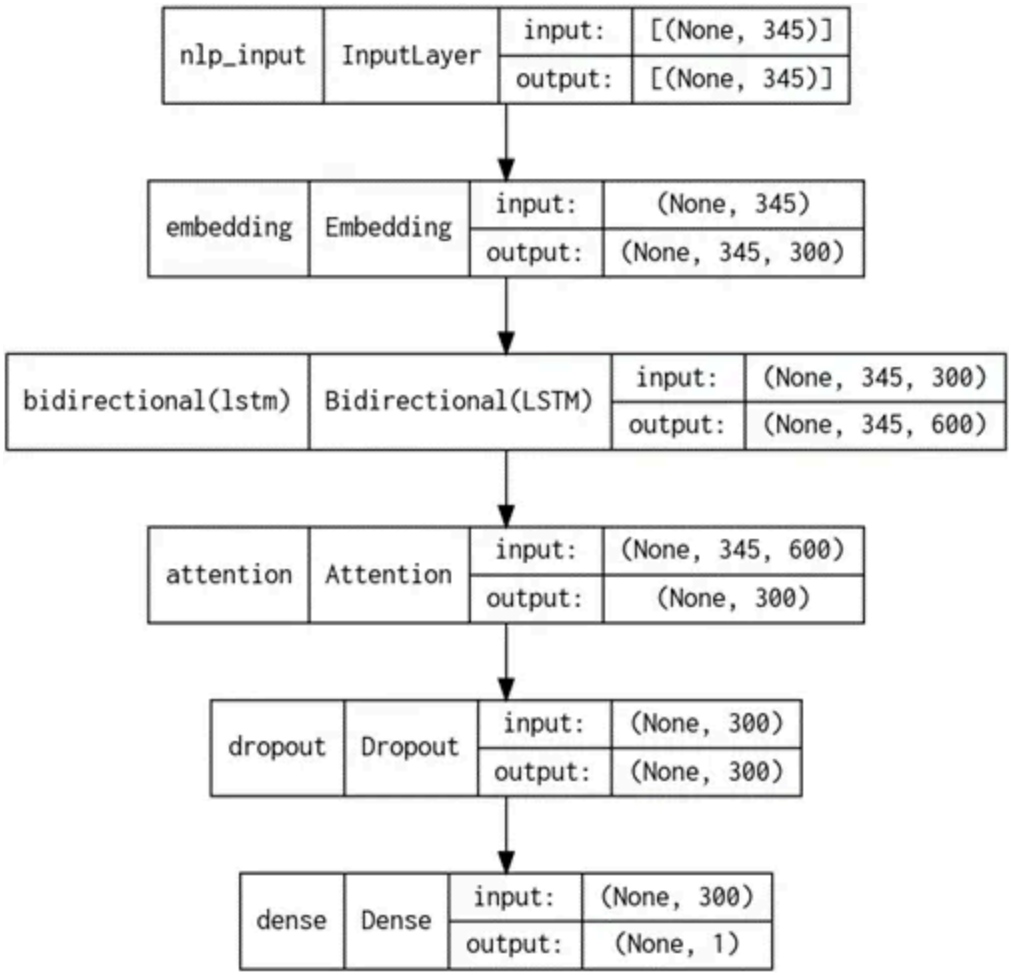
```
        vocab_size = pkl.load(file)
        file.close()

        file = open("/data/model/text_maxlen.pkl",'rb')
        maxlen = pkl.load(file)
        file.close()

        data_tokens_pad = pad_sequences(data_tokens, padding='post', maxlen=max(maxlen, 300), tr

    return data_tokens_pad, data_tokens, vocab_size, max(maxlen, 300)


X_train_text = pd.read_csv('/data/X_train_text.csv', sep = ';')
X_test_text = pd.read_csv('/data/X_test_text.csv', sep = ';')

y_train_std = pd.read_csv('/data/y_train_std.csv', sep = ';')
y_test_std = pd.read_csv('/data/y_test_std.csv', sep = ';')

del X_train_text['Unnamed: 0']
del X_test_text['Unnamed: 0']

print('TRAIN', len(X_train_text), len(y_train_std))
print('TEST', len(X_test_text), len(y_test_std))


prepare_text_data_train(X_train_text)
train_tokens_pad, train_tokens, vocab_size, maxlen = prepare_text_data_inference(X_train_text)
test_tokens_pad, test_tokens, vocab_size, maxlen = prepare_text_data_inference(X_test_text)

file = open("/data/model/text_tokenizer.pkl",'rb')
tokenizer = pkl.load(file)
file.close()



##### LOAD GLOVE EMBEDDINGS
print('glove load...')
# identify the embedding filename; we are using the Glove 42B 300d embeddings
glove_file = "/data/model/glove.840B.300d.txt"
print('glove loaded!')
# create the embeddings index dictionary
embeddings_index = {} # create a lookup dictionary to store words and their vectors
f = open(glove_file, errors='ignore')# open our embedding file
for line in f: # for each line in the file
```

```
    values = line.split(' ') #split the line on spaces between the word and its vectors
    word = values[0] # the word is the first entry
    if word in tokenizer.word_index.keys(): # we check if the word is in our tokenizer word inde
        coefs = np.asarray(values[1:], dtype='float32') # if so, get the word's vectors
        embeddings_index[word] = coefs # add the word and its vectors to the embeddings_index di
f.close()
print('Found %s word vectors.' % len(embeddings_index)) # report how many words in our corpus we

# amount of vocabulary to use, will pick the top 10000 words seen in the corpus
features = 10000
# max text sequence length, must match tokens in transfer file, we are using glove 300d so it is
max_words = 300

num_tokens = (len(tokenizer.word_index) + 1) # for num tokens we always do the length of our wor
hits = 0
misses = 0
embedding_dim = 300

embedding_matrix = np.zeros((num_tokens, max_words)) # setting up an array for our tokens with a
for word, i in tokenizer.word_index.items(): # for each word in the tokenizer word index
    embedding_vector = embeddings_index.get(word) #get the vector from the embeddings index dict
    if embedding_vector is not None: # if the vector isn't None,
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector # store the embedding vector in the matrix at tha
        hits += 1
    else:
        misses += 1
print("Converted %d words (%d misses)" % (hits, misses))


####### CREATING TEXT NEURAL NETWORK ARCHITECTURE WITH GLOVE EMBEDDINGS
# for text embeddings the input shape can be none
nlp_input = Input(shape=(maxlen,), name='nlp_input')

# uses the embedding matrix dictionary to create word embeddings for the inputs
embedded_sequences = Embedding(
                input_dim = num_tokens, # number of unique tokens
                output_dim = embedding_dim, #number of features
                embeddings_initializer=Constant(embedding_matrix), # initialize with Glove embec
                input_length=maxlen,
                trainable=False)(nlp_input)
# uses two bi-directional LSTM layers
lstm = Bidirectional(LSTM(300, return_sequences=True))(embedded_sequences)
att_lstm = Attention(units=300)(lstm)
nlp_out = Dropout(0.5)(att_lstm)
```

```
# adds a dense layer
#output_embed = Dense(350, activation="relu", kernel_initializer='he_normal')(x)
last_layer = Dense(1, activation='linear')(nlp_out)

# declare the final model inputs and outputs
final_model = Model(inputs=nlp_input, outputs=last_layer)

# print a summary of the model
print(final_model.summary())

# set up learning rate decay schedule
initial_learning_rate = 0.1
lr_schedule = ExponentialDecay(
    initial_learning_rate,
        decay_steps=100000,
    decay_rate=0.96,
    staircase=True)

stop = EarlyStopping(monitor="val_loss", patience=25, restore_best_weights=True, mode='min', ver
best = ModelCheckpoint(filepath='/data/model/best_nlp_model.hdf5', save_best_only=True, monitor=

# compile the model
final_model.compile(optimizer=Adam(learning_rate=lr_schedule, epsilon=1),
                    loss="mean_squared_error",
                    metrics=[MeanSquaredError()])

plot_model(final_model, show_shapes=True, to_file='model2_nlp_attention.png')

results = final_model.fit(
            train_tokens_pad, y_train_std,
            epochs=500,
            callbacks=[stop, best],
            validation_data=([test_tokens_pad, y_test_std])
            )

final_model.save('/data/model/pretrained_nlp_model.h5')
```

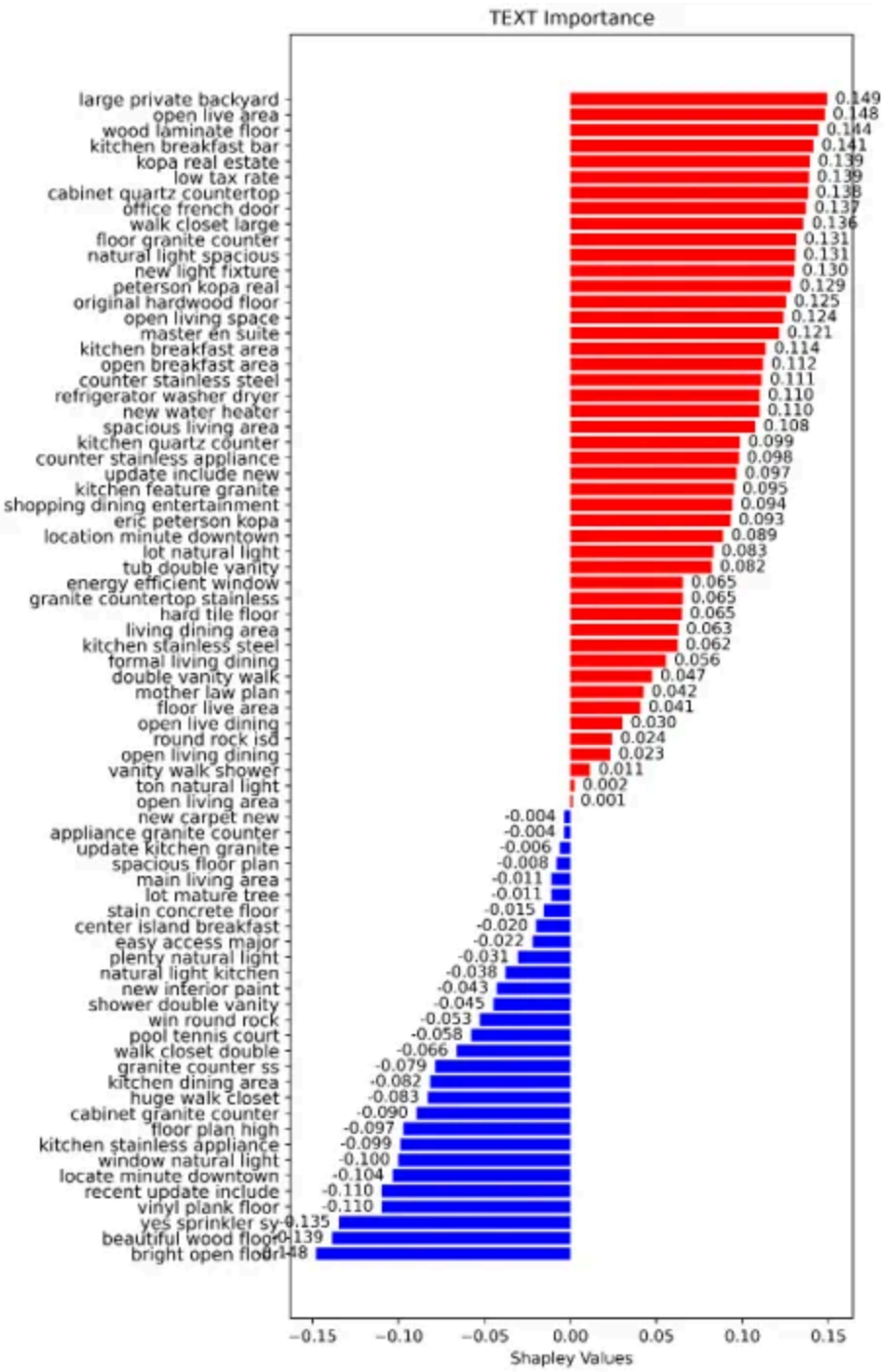
fine_tune_bilstm_atttention_glove.py hosted with ❤ by GitHub view raw

BiLSTM Attention with GloVe embeddings architecture:



Text Neural Netowrk Architecture (BiLSTM Attention with GlOve Embeddings)

Again using shapley values, here are some useful **ngrams** of the house text description to predict price:

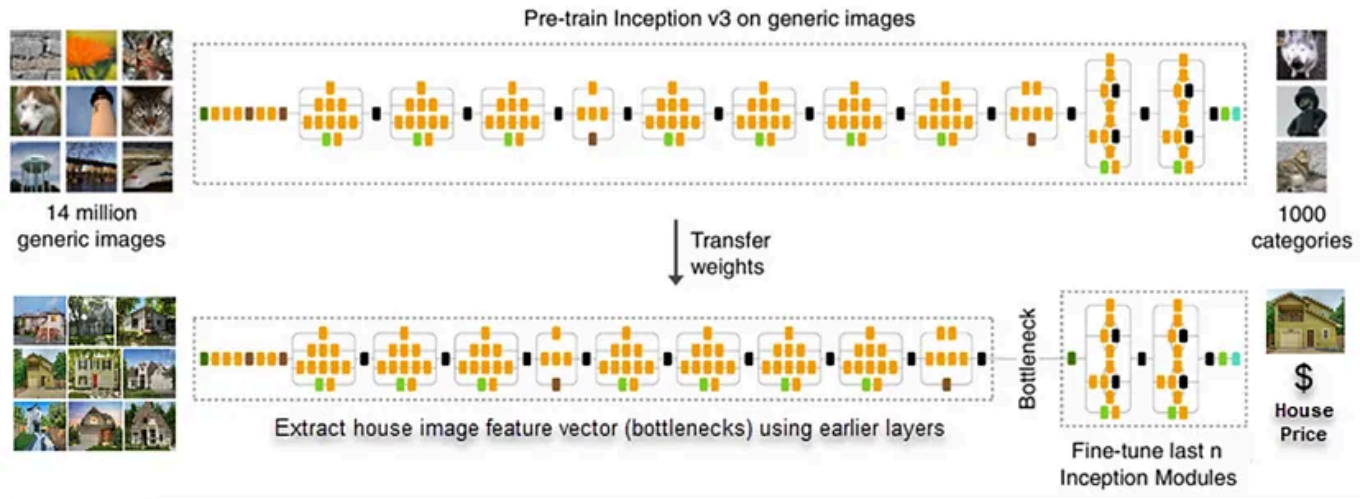


The “**large private backyard**”, “**open live area**” and “**wood laminate floor**” grams appear to have the most positive influence on price according to the shapley values.

In contrast, “**bright open floor**”, “**beautiful wood floor**”, “**yes sprinkler sy**” and “**vinyl plank floor**” seem to have the most negative influence on the price.

Architecture#3 — Image Data fine-tuning with Inception V3:

We use the pre-trained InceptionV3 model on generic image (imagenet weights) and fine-tune only the head layer using our houses prices.



Python code to create the neural network architecture, fine-tune, and save the weights:

```
import pandas as pd
from keras.applications.inception_v3 import InceptionV3
from keras.applications.vgg16 import VGG16


from keras.layers import BatchNormalization, GlobalAveragePooling2D, Dense, Input
from keras.models import Model
from tensorflow.keras.optimizers import Adam
from keras.callbacks import EarlyStopping, ModelCheckpoint
from keras.losses import MeanSquaredError
from tensorflow.keras.optimizers.schedules import ExponentialDecay
from keras.utils.vis_utils import plot_model

from keras.layers import Dense, Flatten, Conv2D, Dropout, MaxPooling2D,MaxPooling1D
from keras.layers import Activation,Dropout,Flatten,BatchNormalization
from keras.models import Sequential
from keras.utils import np_utils

X_train_image = pd.read_csv('/data/X_train_image.csv', sep = ';')
X_test_image = pd.read_csv('/data/X_test_image.csv', sep = ';')

y_train_std = pd.read_csv('/data/y_train_std.csv', sep = ';')
y_test_std = pd.read_csv('/data/y_test_std.csv', sep = ';')


X_train_image['homeImage'] = X_train_image['homeImage'].apply(lambda x: '/data/img/'+str(x))
X_train_image['price'] = y_train_std.values
X_train_image

X_test_image['homeImage'] = X_test_image['homeImage'].apply(lambda x: '/data/img/'+str(x))
X_test_image['price'] = y_test_std.values
X_test_image

img_size_shape = 128

from keras.preprocessing.image import ImageDataGenerator
image_train_generator = ImageDataGenerator(
        brightness_range=(0.75, 1),
        shear_range=0.1,
        zoom_range=[0.75, 1],
    featurewise_center=True,
    featurewise_std_normalization=True,
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2
```

```
    height_shift_range=0.2,
    horizontal_flip=True
        )

# test/val have only the pixel data normalization
image_test_generator = ImageDataGenerator(rescale = 1./1)

# visualize an image augmentation sample
#visualize_augmentations(image_train_generator, images_train.iloc[1])

# specify where the train generator pulls batches
image_train_generator = image_train_generator.flow_from_dataframe(
        dataframe=X_train_image,
        x_col="homeImage",  # this is where your image data is stored
        y_col="price",  # this is your target feature
        class_mode="raw",  # use "raw" for regressions
        color_mode='rgb',
        target_size=(img_size_shape, img_size_shape),
        batch_size=32, # increase or decrease to fit your GPU,
        )

# specify where the test generator pulls batches
image_test_generator = image_test_generator.flow_from_dataframe(
        dataframe=X_test_image,
        x_col="homeImage",
        y_col="price",
        class_mode="raw",
        color_mode='rgb',
        target_size=(img_size_shape, img_size_shape),
        batch_size=32,
        )



model = InceptionV3(weights="imagenet", include_top=False, input_tensor=Input(shape=(img_size_sh

for layer in model.layers:
    layer.trainable=False

img_input = Input(shape=(img_size_shape, img_size_shape, 3))
x = model(img_input, training=False)
x = GlobalAveragePooling2D(name="avg_pool")(x)
x = BatchNormalization()(x)
#x = Dropout(0.5)(x)
output_cnn = Dense(64, activation='relu', kernel_initializer='he_normal', name='dense_hidden')(x
```

```
last_layer = Dense(1, activation='linear')(output_cnn)

model = Model(inputs=img_input, outputs=last_layer)

batch_size = 64
epochs = 100
verbose = 1


initial_learning_rate = 0.1

"""
lr_schedule = ExponentialDecay(
            initial_learning_rate,
                decay_steps=100000,
            decay_rate=0.96,
            staircase=True)
"""

model.compile(
    optimizer=Adam(learning_rate=initial_learning_rate, epsilon=1),
    metrics=['mse'],
    loss='mse'
)

model.summary()

plot_model(model, show_shapes=True, to_file='inceptionV3_image.png')

stop = EarlyStopping(monitor="val_loss", patience=15, restore_best_weights=True, mode='min', ver
best = ModelCheckpoint(filepath='/data/model/best_iv3_model_simple.hdf5', save_best_only=True, m

history = model.fit(
    image_train_generator,
    validation_data = image_test_generator,
    batch_size=batch_size,
    verbose=verbose,
    epochs=epochs,
    callbacks=[stop, best]
)


model.save('/data/model/best_iv3_model_simple.h5')
```

fine_tune_inceptionV3_image.py hosted with ❤ by GitHub view raw

fine_tune_inceptionV3_image.py hosted with by GitHub view raw

The final architecture (InceptionV3 is presented here as a single layer but represents a more complex underlying architecture):

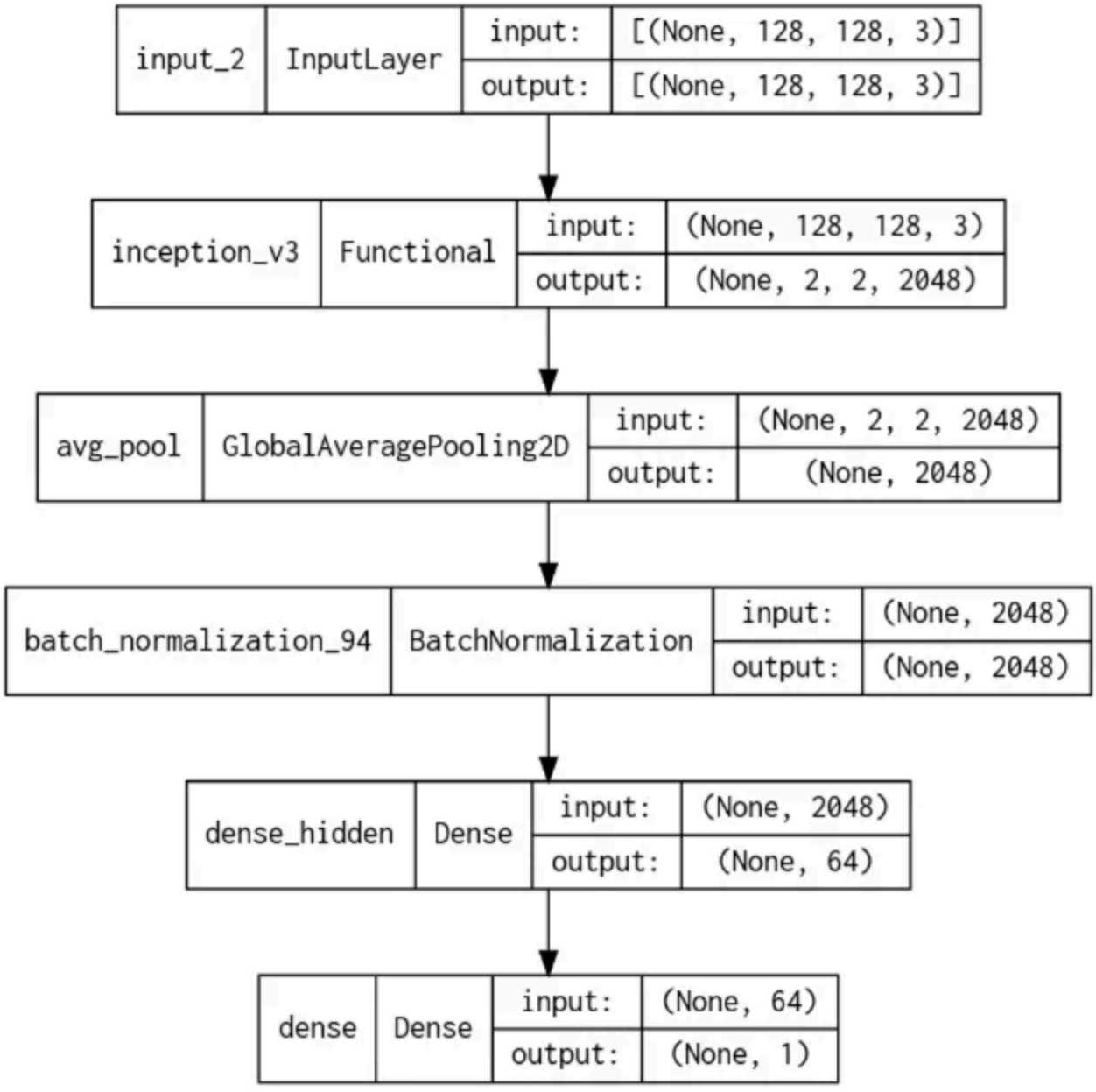


Image Neural Netowrk Architecture (InceptionV3)

Using the **gradcam** method, we can try to understand what our CNN is looking at:

The neural network seems to focus mostly on the house part of the images, which is fine, but there is no evidence which parts of the houses are more important than others… maybe our model is not trained enough !

Grad-CAM, ***Gradient-weighted Class Activation Mapping***, uses the feature maps / class-specific gradient information flowing into the final convolutional layer of a CNN. It produces a visual explanation heat-maps / localization map that will help us understand how the model make decisions / the important regions in the image.

Python code for the gradcam:

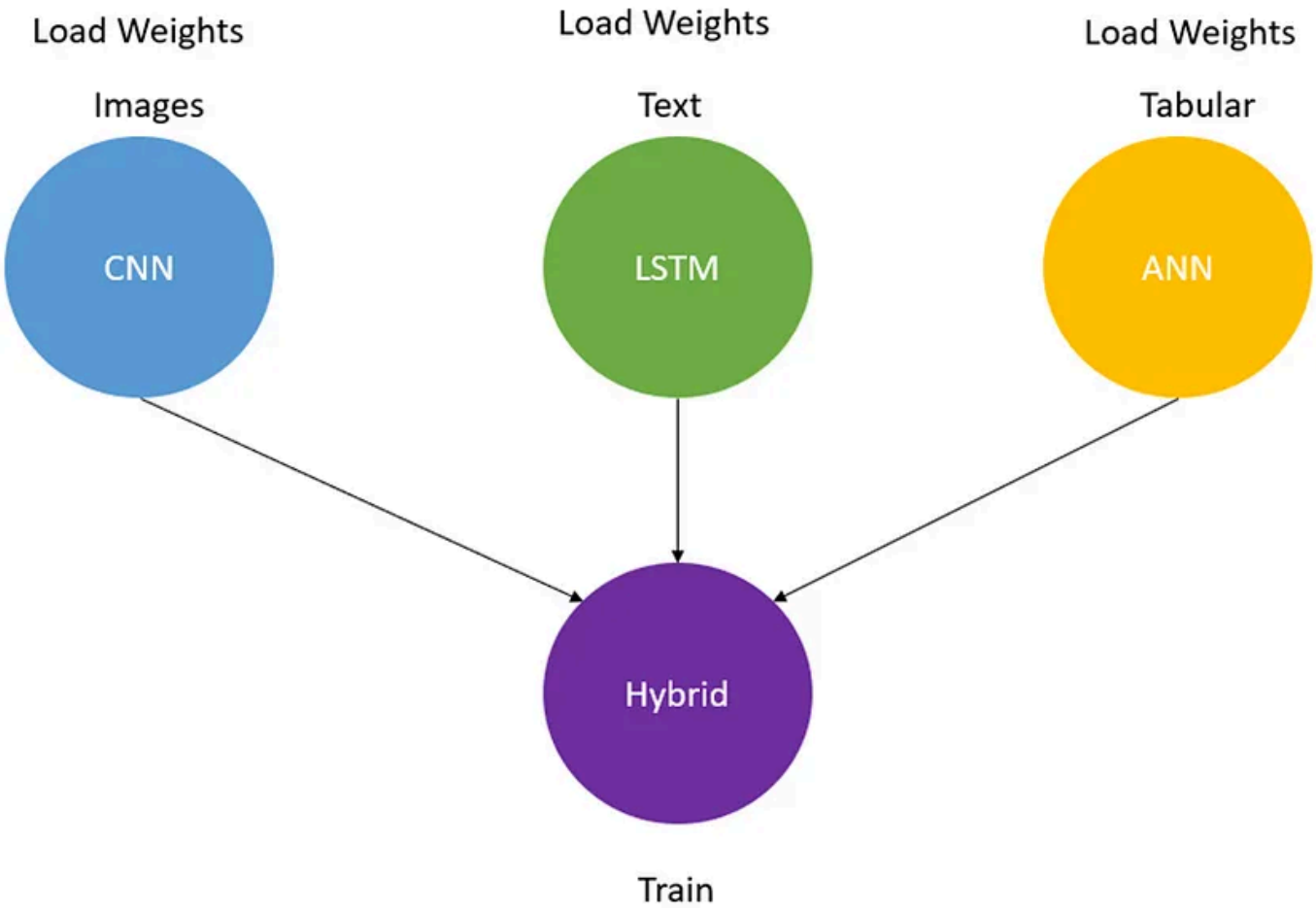
```
from keras.models import Model
import keras
import cv2
import tensorflow as tf
import matplotlib.pyplot as plt
from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
import numpy as np

def find_target_layer(model_pretrained):
    # attempt to find the final convolutional layer in the network
    # by looping over the layers of the network in reverse order
    for layer in reversed(model_pretrained.layers):
        # check to see if the layer has a 4D output
        if len(layer.output_shape) == 4:
            return layer.name
    # otherwise, we could not find a 4D layer so the GradCAM
    # algorithm cannot be applied
    raise ValueError("Could not find 4D layer. Cannot apply GradCAM.")

def compute_heatmap_inceptionV3(model_pretrained, image, eps=1e-8):
    # construct our gradient model by supplying (1) the inputs
    # to our pre-trained model, (2) the output of the (presumably)
    # final 4D layer in the network, and (3) the output of the
    # softmax activations from the model

    gradModel = Model(
        inputs=[model_pretrained.get_layer('inception_v3').inputs],
        outputs=[model_pretrained.get_layer('inception_v3').get_layer(find_target_layer(model_pre
            model_pretrained.get_layer('inception_v3').output])

    # record operations for automatic differentiation
    with tf.GradientTape() as tape:
        # cast the image tensor to a float-32 data type, pass the
        # image through the gradient model, and grab the loss
        # associated with the specific class index
        inputs = tf.cast(image, tf.float32)
        (convOutputs, predictions) = gradModel(inputs)
        loss = predictions[:, 0]
    # use automatic differentiation to compute the gradients
    grads = tape.gradient(loss, convOutputs)

    # compute the guided gradients
    castConvOutputs = tf.cast(convOutputs > 0, "float32")
    castGrads = tf.cast(grads > 0, "float32")
```

```
    castGrads = tf.cast(grads > 0, "float32")
    guidedGrads = castConvOutputs * castGrads * grads
    # the convolution and guided gradients have a batch dimension
    # (which we don't need) so let's grab the volume itself and
    # discard the batch
    convOutputs = convOutputs[0]
    guidedGrads = guidedGrads[0]
    # compute the average of the gradient values, and using them
    # as weights, compute the ponderation of the filters with
    # respect to the weights
    weights = tf.reduce_mean(guidedGrads, axis=(0, 1))
    cam = tf.reduce_sum(tf.multiply(weights, convOutputs), axis=-1)
    # grab the spatial dimensions of the input image and resize
    # the output class activation map to match the input image
    # dimensions
    (w, h) = (image.shape[2], image.shape[1])
    heatmap = cv2.resize(cam.numpy(), (w, h))
    # normalize the heatmap such that all values lie in the range
    # [0, 1], scale the resulting values to the range [0, 255],
    # and then convert to an unsigned 8-bit integer
    numer = heatmap - np.min(heatmap)
    denom = (heatmap.max() - heatmap.min()) + eps
    heatmap = numer / denom
    heatmap = (heatmap * 255).astype("uint8")
    # return the resulting heatmap to the calling function
    return heatmap

fine_tuned_inceptionv3 = keras.models.load_model('/data/model/best_iv3_model_simple.hdf5', compil

for img_file in ["/data/img/110537181_11732947ec76f3d1a8830ec70de79bac-p_f.jpg",
                 "/data/img/2129847996_fbfebce6f34e641b58aafbc76d66ddea-p_f.jpg",
                "/data/img/29333742_612b27ee951dabb4b583c8b4ce87f445-p_f.jpg"]:
    img = load_img(img_file, target_size=(299, 299))
    img = img_to_array(img)
    img = img.reshape((1, img.shape[0], img.shape[1], img.shape[2]))
    img_cv2 = cv2.imread(img_file, 0)
    img_cv2 = cv2.resize(img_cv2, (299, 299))
    img_cv2.shape


    img_heatmap = compute_heatmap_inceptionV3(fine_tuned_inceptionv3, img, eps=1e-8)
    img_heatmap.shape

    colormap=cv2.COLORMAP_TURBO
    alpha = 0.43
```

```
heatmap = cv2.applyColorMap(img_heatmap, colormap, 0)[:, :, 0]
output = cv2.addWeighted(img_cv2, alpha, heatmap, 1 - alpha, 0)
final_output = (heatmap, output)

plt.imsave(img_file.replace('/img', '').replace('.jpg', '') + '_gradcam.png', output, dpi = 5
plt.imshow(output)
plt.close()
```

gradcam.py hosted with ❤ by GitHub view raw

Final Architecture — Load fine-tuned weights and create final hybrid architecture for training



Here is the python code to load all our fine-tuned neural network architecture and then create the final architecture:

```
import pandas as pd
import numpy as np
import pickle as pkl

from keras.preprocessing.image import ImageDataGenerator
from keras.preprocessing.text import Tokenizer
from keras.preprocessing.sequence import pad_sequences

from keras.applications.inception_v3 import InceptionV3
from keras.applications.inception_v3 import preprocess_input
from keras.layers import BatchNormalization, Conv2D, Conv1D, MaxPooling2D, MaxPooling1D, GlobalM
from keras.models import Sequential, Model
from keras.regularizers import l2, l1
from tensorflow.keras.optimizers import Adam, SGD, RMSprop
from keras.callbacks import TensorBoard, EarlyStopping, ModelCheckpoint, History, LearningRateSc
from keras.losses import MeanAbsoluteError, MeanAbsolutePercentageError, MeanSquaredError
#from keras.utils import plot_model, Sequence
from keras.initializers import Constant
from tensorflow.keras.optimizers.schedules import ExponentialDecay

from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array

from attention import Attention
import keras

X_train_structured_std = pd.read_csv('/data/X_train_structured.csv', sep = ';')
X_test_structured_std = pd.read_csv('/data/X_test_structured.csv', sep = ';')

X_train_image = pd.read_csv('/data/X_train_image.csv', sep = ';')
X_test_image = pd.read_csv('/data/X_test_image.csv', sep = ';')

X_train_text = pd.read_csv('/data/X_train_text.csv', sep = ';')
X_test_text = pd.read_csv('/data/X_test_text.csv', sep = ';')

del X_train_text['Unnamed: 0']
del X_test_text['Unnamed: 0']



y_train_std = pd.read_csv('/data/y_train_std.csv', sep = ';')
y_test_std = pd.read_csv('/data/y_test_std.csv', sep = ';')


print('TRAIN', len(X_train_structured_std), len(X_train_image), len(X_train_text), len(y_train_s
```

```
print('TRAIN', len(X_train_structured_std), len(X_train_image), len(X_train_text), len(y_train_std))
print('TEST', len(X_test_structured_std), len(X_test_image), len(X_test_text), len(y_test_std))

#file = open("/data/model/X_numerical_scaler.pkl",'rb')
#X_numerical_scaler = pkl.load(file)
#file.close()

file = open("/data/model/y_numerical_scaler.pkl",'rb')
y_numerical_scaler = pkl.load(file)
file.close()

X_train_image['homeImage'] = X_train_image['homeImage'].apply(lambda x: '/data/img/'+str(x))
X_train_image['price'] = y_train_std.values
X_train_image

X_test_image['homeImage'] = X_test_image['homeImage'].apply(lambda x: '/data/img/'+str(x))
X_test_image['price'] = y_test_std.values
X_test_image




def load_house_images(df, image_size):
    # initialize our images array (i.e., the house images themselves)
    images = []
    # loop over the indexes of the houses
    for i, image_file in enumerate(df['homeImage'].values):

        # find the four images for the house and sort the file paths,
        # ensuring the four are always in the *same order*
        # initialize our list of input images along with the output image
        # after *combining* the four input images
        inputImages = []
        # loop over the input house paths
        img = load_img(image_file, target_size=(image_size, image_size))
        img = img_to_array(img)
        img = img.reshape((1, img.shape[0], img.shape[1], img.shape[2]))

        if i%1000 == 0:
            print(i, image_file, img.shape)
        elif i == 0:
            print(i, image_file, img.shape)

        images.append(img)
        # tile the four input images in the output image such the first
```

```
        # image goes in the top-right corner, the second image in the
        # top-left corner, the third image in the bottom-right corner,
        # and the final image in the bottom-left corner
        # return our set of images
    images = np.array(images)
    images = images.reshape((len(images), image_size, image_size, 3))
    return images



X_train_image_array = load_house_images(X_train_image, image_size = 128)
X_test_image_array = load_house_images(X_test_image, image_size = 128)

pkl.dump(X_train_image_array, open('/data/model/X_train_image_array.pkl','wb'))
pkl.dump(X_test_image_array, open('/data/model/X_test_image_array.pkl','wb'))


file = open("/data/model/X_train_image_array.pkl",'rb')
X_train_image_array = pkl.load(file)
file.close()

file = open("/data/model/X_test_image_array.pkl",'rb')
X_test_image_array = pkl.load(file)
file.close()


def prepare_text_data_train(dataf):
    for col in dataf.columns:
        dataf[col] = dataf[col].str.lower()

        tokenizer = Tokenizer(num_words=2096) #5000
        tokenizer.fit_on_texts(dataf[col])
        pkl.dump(tokenizer, open('/data/model/text_tokenizer.pkl','wb'))

        vocab_size = len(tokenizer.word_index) + 1
        print(vocab_size)

        pkl.dump(vocab_size, open('/data/model/text_vocab_size.pkl','wb'))

        maxlen = int(np.mean(dataf[col].str.len()))
        pkl.dump(maxlen, open('/data/model/text_maxlen.pkl','wb'))

def prepare_text_data_inference(dataf):
    for col in dataf.columns:
        dataf[col] = dataf[col].str.lower()
```

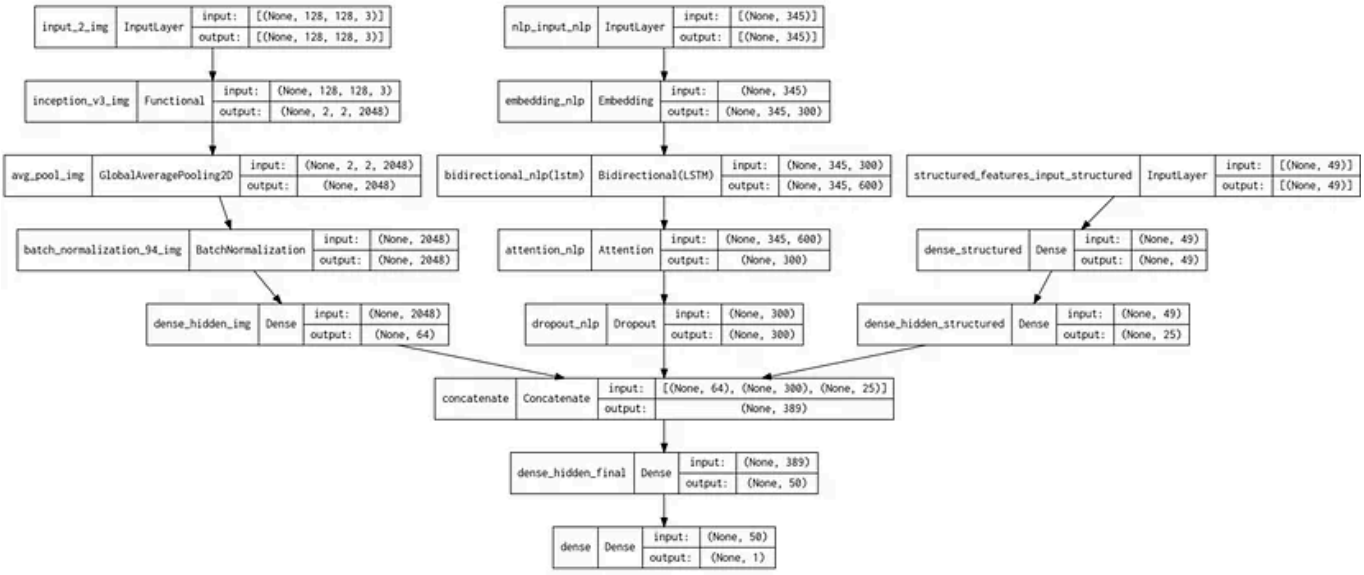
```
135
136         file = open("/data/model/text_tokenizer.pkl",'rb')
137         tokenizer = pkl.load(file)
138         file.close()
139
140         data_tokens = tokenizer.texts_to_sequences(dataf[col])
141
142         file = open("/data/model/text_vocab_size.pkl",'rb')
143         vocab_size = pkl.load(file)
144         file.close()
145
146         file = open("/data/model/text_maxlen.pkl",'rb')
147         maxlen = pkl.load(file)
148         file.close()
149
150         data_tokens_pad = pad_sequences(data_tokens, padding='post', maxlen=max(maxlen, 300), tr
151
152     return data_tokens_pad, data_tokens, vocab_size, max(maxlen, 300)
153
154 prepare_text_data_train(X_train_text)
155 train_tokens_pad, train_tokens, vocab_size, maxlen = prepare_text_data_inference(X_train_text)
156 test_tokens_pad, test_tokens, vocab_size, maxlen = prepare_text_data_inference(X_test_text)
157
158 file = open("/data/model/text_tokenizer.pkl",'rb')
159 tokenizer = pkl.load(file)
160 file.close()
161
162
163
164 fine_tuned_inceptionv3 = keras.models.load_model('/data/model/best_iv3_model_simple.hdf5', compi
165 fine_tuned_glove_bilstm_att = keras.models.load_model('/data/model/best_nlp_model.hdf5', compile
166 fine_tuned_structured = keras.models.load_model('/data/model/best_structured_model.hdf5', compil
167
168 fine_tuned_inceptionv3_layer = Model(inputs=fine_tuned_inceptionv3.input, outputs=fine_tuned_inc
169 fine_tuned_glove_bilstm_att_layer = Model(inputs=fine_tuned_glove_bilstm_att.input, outputs=fine
170 fine_tuned_structured_layer = Model(inputs=fine_tuned_structured.input, outputs=fine_tuned_struc
171
172 for layer in fine_tuned_structured_layer.layers:
173     layer._name = layer.name + '_structured'
174
175 for layer in fine_tuned_glove_bilstm_att_layer.layers:
176     layer._name = layer.name + '_nlp'
177
178 for layer in fine_tuned_inceptionv3_layer.layers:
```

```
    layer._name = layer.name + '_img'

x = keras.layers.concatenate([fine_tuned_inceptionv3_layer.output,
                          fine_tuned_glove_bilstm_att_layer.output,
                          fine_tuned_structured_layer.output])
x = Dense(50, activation='relu', name='dense_hidden_final')(x)
last_layer = Dense(1, activation='linear')(x)


model = keras.Model(inputs=[fine_tuned_inceptionv3_layer.input,
                            fine_tuned_glove_bilstm_att_layer.input,
                            fine_tuned_structured_layer.input
                           ],
                    outputs=[last_layer])

initial_learning_rate = 0.1

'''
lr_schedule = ExponentialDecay(
            initial_learning_rate,
                decay_steps=100000,
            decay_rate=0.96,
            staircase=True)
'''

model.compile(optimizer=Adam(learning_rate=initial_learning_rate, epsilon=1), loss="mean_squarec
print(model.summary())

from tensorflow.keras.utils import plot_model
plot_model(model, to_file="final_model.png",
    show_shapes=True,
    show_dtype=False,
    show_layer_names=True,
    dpi=350)

print('saving...')
model.save('/data/model/best_final_model_simple.h5')
print('saved DONE!')

print('loading...')
reconstructed_model = keras.models.load_model("/data/model/best_final_model_simple.h5", compile=
print('loaded DONE!')

plot_model(reconstructed_model, show_shapes=True, to_file='final_model_image.png')
```

```
reconstructed_model.compile(optimizer=Adam(learning_rate=initial_learning_rate, epsilon=1), loss
print(reconstructed_model.summary())

stop = EarlyStopping(monitor="val_loss", patience=15, restore_best_weights=True, mode='min', ver
best = ModelCheckpoint(filepath='/data/model/best_final_model_weights_only.h5',
                       save_best_only=True,
                       save_weights_only=False,
                       monitor='val_loss',
                       mode='min', verbose=1)

results = reconstructed_model.fit([X_train_image_array,
                    train_tokens_pad,
                    X_train_structured_std],
                    y_train_std,
                    epochs=500,
                    batch_size = 250,
                    validation_data=([X_test_image_array,
                                      test_tokens_pad,
                                       X_test_structured_std],
                                      y_test_std),
                    callbacks=[stop, best],
                    )

results.save('/data/model/final_hybrid_model.h5')
```

final_hybrid_architecture.py hosted with ❤ by GitHub view raw

The final architecture:



Final Hybrid (Multimodal) Neural Netowrk Architecture

We can see the concatenate layer in the center that concatenate our 3 previous neural network architectures; image, text and structured data respectively.

Results and Conclusions

If we take a quick look at our first results:

```
import pandas as pd
import pickle as pkl
from attention import Attention
import keras
from sklearn import metrics

print('loading...')
final_model = keras.models.load_model("/data/model/best_final_model_weights_only.h5", compile=Fa
fine_tuned_inceptionv3 = keras.models.load_model('/data/model/best_iv3_model_simple.hdf5', compi
fine_tuned_glove_bilstm_att = keras.models.load_model('/data/model/best_nlp_model.hdf5', compile
fine_tuned_structured = keras.models.load_model('/data/model/best_structured_model.hdf5', compil
fine_tuned_structured_nlp = keras.models.load_model('/data/model/best_model_struct_nlp.h5', comp
print('loaded DONE!')

X_train_structured_std = pd.read_csv('/data/X_train_structured.csv', sep = ';')
X_test_structured_std = pd.read_csv('/data/X_test_structured.csv', sep = ';')

file = open("/data/model/X_train_image_array.pkl",'rb')
X_train_image_array = pkl.load(file)
file.close()

file = open("/data/model/X_test_image_array.pkl",'rb')
X_test_image_array = pkl.load(file)
file.close()

X_train_text = pd.read_csv('/data/X_train_text.csv', sep = ';')
X_test_text = pd.read_csv('/data/X_test_text.csv', sep = ';')

del X_train_text['Unnamed: 0']
del X_test_text['Unnamed: 0']

y_train_std = pd.read_csv('/data/y_train_std.csv', sep = ';')
y_test_std = pd.read_csv('/data/y_test_std.csv', sep = ';')

file = open("/data/model/y_numerical_scaler.pkl",'rb')
y_numerical_scaler = pkl.load(file)
file.close()

def prepare_text_data_train(dataf):
    for col in dataf.columns:
        dataf[col] = dataf[col].str.lower()

        tokenizer = Tokenizer(num_words=2096) #5000
        tokenizer.fit_on_texts(dataf[col])
        pkl.dump(tokenizer, open('/data/model/text_tokenizer.pkl', 'wb'))
```

```
        pkl.dump(tokenizer, open('/data/model/text_tokenizer.pkl','wb'))

        vocab_size = len(tokenizer.word_index) + 1
        print(vocab_size)

        pkl.dump(vocab_size, open('/data/model/text_vocab_size.pkl','wb'))

        maxlen = int(np.mean(dataf[col].str.len()))
        pkl.dump(maxlen, open('/data/model/text_maxlen.pkl','wb'))

def prepare_text_data_inference(dataf):
    for col in dataf.columns:
        dataf[col] = dataf[col].str.lower()

        file = open("/data/model/text_tokenizer.pkl",'rb')
        tokenizer = pkl.load(file)
        file.close()

        data_tokens = tokenizer.texts_to_sequences(dataf[col])

        file = open("/data/model/text_vocab_size.pkl",'rb')
        vocab_size = pkl.load(file)
        file.close()

        file = open("/data/model/text_maxlen.pkl",'rb')
        maxlen = pkl.load(file)
        file.close()

        data_tokens_pad = pad_sequences(data_tokens, padding='post', maxlen=max(maxlen, 300), tr

    return data_tokens_pad, data_tokens, vocab_size, max(maxlen, 300)

prepare_text_data_train(X_train_text)
train_tokens_pad, train_tokens, vocab_size, maxlen = prepare_text_data_inference(X_train_text)
test_tokens_pad, test_tokens, vocab_size, maxlen = prepare_text_data_inference(X_test_text)

y_pred_img = fine_tuned_inceptionv3.predict(X_test_image_array)
y_pred_nlp = fine_tuned_glove_bilstm_att.predict(test_tokens_pad)
y_pred_structured = fine_tuned_structured.predict(X_test_structured_std)
y_pred_structured_nlp = fine_tuned_structured_nlp.predict([test_tokens_pad, X_test_structured_st
y_pred_all = final_model.predict([X_test_image_array, test_tokens_pad, X_test_structured_std])

y_true = y_numerical_scaler.inverse_transform(y_test_std)
y_true_train = y_numerical_scaler.inverse_transform(y_train_std)
y_pred_img = y_numerical_scaler.inverse_transform(y_pred_img)
```

```
y_pred_nlp = y_numerical_scaler.inverse_transform(y_pred_nlp)
y_pred_structured = y_numerical_scaler.inverse_transform(y_pred_structured)
y_pred_structured_nlp = y_numerical_scaler.inverse_transform(y_pred_structured_nlp)
y_pred_all = y_numerical_scaler.inverse_transform(y_pred_all)

y_pred_compare = pd.DataFrame(y_true)
y_pred_compare.columns = ['Y_TRUE']
y_pred_compare['Y_PRED_STRUCTURED'] = y_pred_structured
y_pred_compare['Y_PRED_NLP'] = y_pred_nlp
y_pred_compare['Y_PRED_IMG'] = y_pred_img
y_pred_compare['Y_PRED_HYBRID_NLP'] = y_pred_structured_nlp
y_pred_compare['Y_PRED_HYBRID_ALL'] = y_pred_all
y_pred_compare['MEAN'] = np.mean(y_true_train.ravel())
print('MEAN price:', np.mean(y_true_train.ravel()))


print('R2')
print('STRUCTURED', metrics.r2_score(y_pred_compare['Y_TRUE'], y_pred_compare['Y_PRED_STRUCTURED
print('NLP', metrics.r2_score(y_pred_compare['Y_TRUE'], y_pred_compare['Y_PRED_NLP']))
print('IMG', metrics.r2_score( y_pred_compare['Y_TRUE'], y_pred_compare['Y_PRED_IMG']))
print('HYBRID_ALL', metrics.r2_score( y_pred_compare['Y_TRUE'], y_pred_compare['Y_PRED_HYBRID_AL
print('HYBRID_NLP', metrics.r2_score( y_pred_compare['Y_TRUE'], y_pred_compare['Y_PRED_HYBRID_NL
print('DUMMY_MEAN_MODEL', metrics.r2_score( y_pred_compare['Y_TRUE'], y_pred_compare['MEAN']))
print('---------------')
print('RMSE')
print('STRUCTURED', np.sqrt(metrics.mean_squared_error(y_pred_compare['Y_TRUE'], y_pred_compare[
print('NLP', np.sqrt(metrics.mean_squared_error(y_pred_compare['Y_TRUE'], y_pred_compare['Y_PRED
print('IMG', np.sqrt(metrics.mean_squared_error( y_pred_compare['Y_TRUE'], y_pred_compare['Y_PRE
print('HYBRID_ALL',np.sqrt( metrics.mean_squared_error( y_pred_compare['Y_TRUE'], y_pred_compare
print('HYBRID_NLP', np.sqrt(metrics.mean_squared_error( y_pred_compare['Y_TRUE'], y_pred_compare
print('DUMMY_MEAN_MODEL', np.sqrt(metrics.mean_squared_error( y_pred_compare['Y_TRUE'], y_pred_c
print('---------------')
print('MAE')
print('STRUCTURED', metrics.mean_absolute_error(y_pred_compare['Y_TRUE'], y_pred_compare['Y_PRED
print('NLP', metrics.mean_absolute_error(y_pred_compare['Y_TRUE'], y_pred_compare['Y_PRED_NLP'])
print('IMG', metrics.mean_absolute_error( y_pred_compare['Y_TRUE'], y_pred_compare['Y_PRED_IMG'
print('HYBRID_ALL', metrics.mean_absolute_error( y_pred_compare['Y_TRUE'], y_pred_compare['Y_PRE
print('HYBRID_NLP', metrics.mean_absolute_error( y_pred_compare['Y_TRUE'], y_pred_compare['Y_PRE
print('DUMMY_MEAN_MODEL', metrics.mean_absolute_error( y_pred_compare['Y_TRUE'], y_pred_compare[
```

model_benchmark.py hosted with ❤ by GitHub view raw

Model	R2 Score	RMSE	MAE
Dummy Mean House Price model (calculated on train set)	0.00	280 790.14$	187 199.09$
Structured Data NN Architecture alone	0.808	123 102.14$	79 105.03$
Text Data NN Architecture alone	0.390	219 223.63$	143 342.99$
Image Data NN Architecture alone	0.076	269 962.43$	176 746.92$
Hybrid [Structured+Text] NN Architecture	**0.809**	**122 700.20$**	**77 911.38$**
Hybrid [Structured+Text+Image] NN Architecture	**0.783**	**130 720.34$**	**85 008.93$**

Model benchmark

The worst model of all is the dummy mean house price model where the price of each house = average price of the whole training set.

The second worst model of all is the **Image Data NN Architecture** alone where there seems to be very slight house price predictive power, meaning that house images doesn't correlate very much with the house prices.

Does it make sense ?

The price of a house is not limited to the appearance of the exterior. The local real estate market seems a lot more important. For example, if we take the same house and simply move it to another **regionCluster**, the price could be very different. Therefore, it should not be surprising that our Image Data NN Architecture alone trained on image didn't perform as well as the Structured Data NN Architecture trained on the attributes containing much more context about the local real estate market. Since our result is bad, we could use some strategies to improve our image data neural network and dataset:

- Having a dataset where photos are more consistent and taken from the same angles could be helpful.

- It is difficult to represent a full house using a single image. Having a dataset with multiple photos for each house (interior, exterior, kitchen, bathroom, etc.) could have helped our model a lot. Multiple images can be done by creating a montage that combines all images into a single image and then pass the montage through the CNN input.
- Direct regression task on images maybe not the best choice. Using bounding boxes to detect objects in images like different types of garages, windows, doors, patio, swimming pool, etc. could be an interesting alternative and surely reduced some unwanted pixel noise in images but requires much more work and time to prepare datasets.
- Maybe more epochs could have helped… choosing a different model than InceptionV3… hyper-parameter tuning… etc.

Our **Text Data NN Architecture** alone performed better than the dummy mean house price model. There seems to be predictive information / correlations on house prices when using the text description.

However, adding the textual description with our structured data (Hybrid [Structured+Text] NN architecture) seems to have only added noises because the performance is almost the same as structured data alone. One explanation could be that the house textual description contains redundant information from our structured features.

Third place is our **Final Hybrid neural network architecture** where images and text seem to have added noise to the structured features.

The 2 winners (almost tied) are the **Structured Data Neural Network** with a MAE of **79 105$** and the **Hybrid (Structured + Text) Neural Network** with a

MAE of **77 911$** (Sounds high but our Austin houses present in the dataset are expensives with an average price of **499 635$**)

Final words

In a context where we have better informative images and better informative texts correlated to our target, it would be very interesting to experiment again with this kind of hybrid architecture and see how it goes !

Also, keep in mind that in practice, it is difficult to combine different noise levels and conflicts between modalities. Moreover, modalities have different quantitative influence on the prediction output.

The most common method is to combine high-level embeddings from the different inputs by concatenating them, just like we did. The problem with this approach is that it would give an equal importance to all the sub-networks / modalities which is highly unlikely in real-life situations. For example, using a weighted combination of networks could be an alternative.

Multimodal | Deep Learning | NLP | Computer Vision | Neural Networks

Written by Dave Cote, M.Sc.

259 Followers · 34 Following

Data Scientist in an insurance company. More than 10 years in Data Science and for delivering actionnable « Data-Driven » solutions.

Follow