Master in Language Technology
Dept of Philosophy, Linguistics and Theory of Science (FLoV)
University of Gothenburg

Predicting Houses Prices Classes Based on Both Tabular and Visual Data

For the course: *Machine learning for statistical NLP: Advanced LT2326*

Background

PropTech, an essential and increasingly relevant field today, is set to revolutionize the real estate industry. Neural network models could jump in to assist buyers to approximate and calculate property prices and streamline the workflow of real estate agencies. These tools are especially useful for potential buyers, enabling them to accurately estimate the market value of properties, a crucial advantage in situations of suspected overpricing or for those with minimal experience in real estate. An efficient model has the potential to reduce fraudulent activities, providing insightful price predictions that bolster trust and clarity in the property market.

The integration of data-driven decision making in PropTech utilizes advanced data analytics to allow making more strategic decisions and improves market accessibility. The real estate market could be more approachable to a wider audience, including those far away or new to the market. This expansion fosters market growth and diversification.

PropTech is on course to transform the real estate sector into a more efficient, transparent, and globally accessible industry and neural networks could play a big part of it.

I selected this project primarily because it presented an opportunity to delve into a model that integrates visual data analysis. This venture goes beyond our usual focus in the program on tabular or textual data, offering a more diverse and challenging experience. Working with visual data opens up new dimensions to explore different techniques and tools. This expansion is not only intriguing but also enhances my skill set, providing a broader understanding of various data formats and how they can be effectively utilized in model development.

In this project we predict the *price class* of the houses based on some specific features like area, no. of bedrooms, location (Zipcode) and no. of bathrooms with the help of some pictures from these houses to then predict the *price class* based on these parameters.

Resources

When buying a house, relying on numbers only is not realistic as we don't buy a house unless we go to a showing or get a feeling of how it looks like to then decide whether we want to buy it or not.

The dataset used in this project is composed of 504 sample houses from California State in the United State from the year 2016. It features a unique combination of visual and textual (tabular) elements by including four images for each property, ***a total of 2016 images***, showcasing ***the bedroom***, ***bathroom***, ***kitchen***, and ***a front view of the house***. Finding a dataset that contains both textual data and images from the house was quite challenging, the authors claim that this dataset is the first of its type for the purpose of predicting house prices (to their knowledge) (Eman H. Ahmed, 2016).

Data Attributes: ***This dataset encompasses 4 textual attributes*** and ***additional visual attributes*** that can be derived from the images (Eman H. Ahmed, 2016).

The dataset folder contains 1 folder that of all the images with an accompanying text file providing descriptive metadata for each property. This file is organized sequentially, with each row corresponding to a property. The data points include ***the number of bedrooms, the number of bathrooms***, ***the property's total area***, its ***zipcode***, and ***the listed price*** and there are no missing values in this dataset.

We have to note that the concept of global market adaptation is crucial in PropTech's expansion. Different international markets may embrace PropTech differently, influenced by their unique real estate trends, and cultural nuances. Adapting to these variations is vital for the worldwide acceptance and effectiveness of PropTech, ensuring it meets the specific needs of each market. And because of that we should note that this dataset is old and only concerns the houses in California state in USA. Therefore, this project is directed to this specific market and needs a more up to date dataset.

Methods

Preprocessing Data

I created HouseDataset class which inherits the Dataset class from Pytorch with its main functions: ***__init__***, ***__len__*** and ***__getitem__,*** in addition to two extra functions that are used in our class: ***load_df*** and ***preprocess_data***, which are used to preprocess tabular(textual) data. The class takes the path of the text file first followed by the path to the images folder. It is also possible to optionally pass a feature column name to avoid including this particular feature data when creating the dataset.

- MetaData: The data was loaded to a Jupiter notebook and save in a Pandas dataframe. The column names were also added to the dataframe (***df***) as well as the index to the to be then used when creating the tensors. The values in the ***df*** were converted to floats as they were extracted from a text file as strings. *Zipcode* is considered to be a categorical data and to be able to use it in our neural network model, it needs to be encoded into numerical representation, like we did here with label encoding, one-hot-encoding is also a way of doing it. Because our model is working as a classifier, we need to determine the boundaries of our model's classes and we do that by first doing a feature scaling as it makes it easier to deal with the numbers and avoid any instability issues. After the scaling is done, we create the boundaries which were 10 steps apart in addition to adding the infinity as last boundary, resulting with *prices_boundaries = [0, 10, 20, 30, 40, 50, inf]*. The prices boundaries then are 7, therefor we have 6 classes. The last step for the prices would be to replace the *prices* column with the *prices_categories* column in the ***df***. All the features then are standardized using a standard scaler by fitting the scaler to the data and then transforming the data. The transformation involves subtracting the mean of each feature and dividing by the standard deviation, resulting in features with a mean of 0 and a standard deviation of 1. The textual features could be edited, meaning we can disregard a certain feature to then test whether it negatively manipulate the prediction or not, and that is done by passing the name of the feature column when creating the dataset as we will show later.
- Images: The images are transformed by converting them to gray and have only one-color channel instead of three (RGB) which will reduce the computational complexity and makes it faster. We then resize the images to be able then to batch them together in the *dataloader* and also to use less memory than when using the original size. The images are then converted to tensors as we are going to use them in Pytorch neural networks, to normalize them at the end by 0.5 for both the mean and the standard deviation just like what we did to the MetaData to speed up training. The normalization will transform the data values instead of [0,255] to [-1.1] because:

  Min of input image = 0/255 = 0 => 0-0.5 = -0.5 => -0.5/0.5 => -1
  Max of input image = 255/255 = 1 => (1 - 0.5) / 0.5 => 1

We load the 4 images of each house and create a list of tensors; each tensor represents an image. All the images have the same index in the file names, and it is then used to get the textual features.

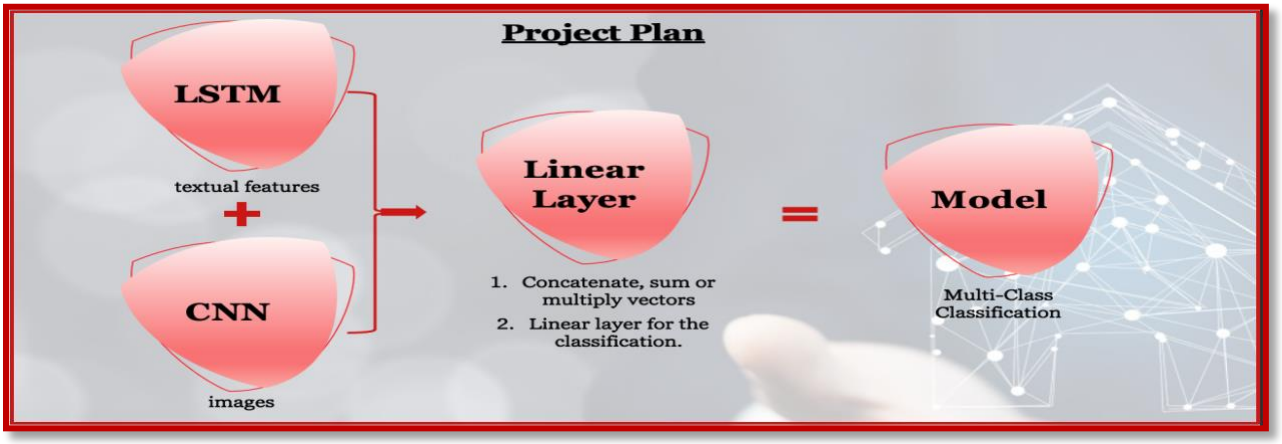
In the ***__getitem__*** function we finally create the tensors which contain the images, textual features, and labels respectively. Both the textual features and labels were too converted to tensors to be used in the neural networks.

Data Loaders

We create the dataset by calling the HouseDataset class and passing the text file path, the images folder path, and optionally one of the features column names.

The function ***split_data*** will first split the data randomly to training, validation, and testing datasets, respectively. The training dataset is set to 65% of all the dataset, the testing set is 20% of the dataset and the rest is used for validation. The function allows us to create the three *dataloaders* and return them upon calling the function with passing the created dataset, and the batch size as the function parameters. The training *dataloader* is shuffled to make sure that batches are different in each epoch and thus the model is not memorizing the data which will help us avoid overfitting. Shuffling the evaluation data, was it validation or test data is not necessary.

Neural Networks



In general, our model uses a CNN model to process multi-image input simultaneously and an LSTM model to process the textual/tabular features. The output of the two previous model is then goes to a linear layer classifier after concatenating, summing, or multiplying the two inputs. The model architecture consists of three classes, all inherit the *nn.Module* class and have two functions ***__init__*** and ***forward***. The first function is used to initialize and define the layers for each model and the second is what we use to call the model and to pass the input tensor through these layers.

- Convolutional Neural Network: Class ***MultiImgsCNN*** takes a list of total four tensors (number of used images) as input when calling it. Each tensor in this list goes into a separate and identical convolutional layer with an output of 128 channels followed by an activation function to help introduce non-linearity and allow the model to learn more complex patterns. Passing each image through its own convolutional layer allows the model to learn and extract features from each image separately as each image may contain different types of information, and having separate partitions allows the model to learn features that are specific to each image. The outputs of the activation functions will then be concatenated and passed to another convolutional layer with input of 512 channels (because of the combined outputs from the previous step) and an output of 256 channels. This output goes into a max pooling layer, this will reduce the size of this output

and therefore computational complexity and help highlight the most dominant features in its input, followed by an activation function (ReLU). The last three steps will be repeated, convolutional layer, max pooling and an ReLU. Finally, we flatten the output of ReLU to be able to use it in the last fully connected layer which will help in specifying the number of classes in case of using only the images in the model for the classification task, or to customize the number of neurons of the CNN model which in our case is used in the final model.

- Long Short Term Memory (LSTM): Class ***LSTMModel*** takes the tensor of tabular data as an input when calling the model but also takes input size (number of features), hidden size, and the number of layers. The input will pass into an LSTM layer, then the output of the last LSTM layer goes into a fully connected layer and an activation function respectively, disregarding the hidden state of the last layer.
- The Final Model: Class ***Model*** calls both the previous models in the ***__init__*** function and initialize a linear layer which is used to combine them. The size of the linear layer's input depends on the way we combine the output of the previous models, whereas the output size on the number of classes we have. If we concatenate, then the input size would be the sum of both outputs sizes ***(MultiImgsCNN***'s output size + ***LSTMModel***'s output size***)***. If we multiply or sum the outputs, then both outputs should be of the same size to be multiplied or summed and the input size would be equal to the output size of the models.

Training Loop

Two functions are used for the training loop, ***train*** and ***validate,*** which is called by ***train*** to help optimize the model, tune the hyperparameters and monitor the overfitting. When calling the ***train*** function we need to pass the model, loss function, optimizer, training dataloader, validation dataloader, chosen number of epochs, and the name of the model that we chose to save it with respectively. We set the model to the training mode and enter the training loop which will be repeated every epoch. We extract images, tabular features, and labels from the dataloaders and then pass them to them when calling the model and calculating the loss. Saving the model and the evaluation using the validation dataloader happens every two epochs by calculating both the loss and the accuracy which are printed out to monitor the model's performance. In the backpropagation we compute the gradients which then helps in updating the weights and thus improve the model's performance.

Results & Discussion

In this project, I have trained several models، and they are shown in the below table. All models were trained with 30 epochs except for the last one which outperformed the previous models. The architecture of the last model is also different, I used in it two layers of *nn.Conv2d, nn.MaxPool2d,* and *nn.ReLU* respectively instead of one, which improved the model's performance according to the accuracy metrics noticeably. Another thing I have done was to increase the number of channels in the first 4 initial convolutional layers from 64 to 128 channels, which also improved performance.

houses_imgs: I made the above-mentioned changes on ***MultiImgsCNN*** class after seeing how poorly the old images model performes on its own, as the table is showing how it struggles to make correct predictions and often incorrectly classifies the instances passed to it. The model most probably was too simple, and it needed to capture more complex patterns. This model is predicting correctly only 37.5% of predictions, while 27.22% of all the predicted positives are actually positive and 28.71% of all actual positives were correctly predicted by this model as we read from accuracy, precision, and recall values respectively and thus F1 score.

houses_feat: While it might look like this model is performing well by only looking at accuracy, it doesn't. Accuracy might be misleading when it comes to measuring the quality of model's performance as some datasets' classes may be imbalanced which is what we see here. This means that our model is performing well with the major class but not with the other classes as precision, recall and F1 score indicated with their around 60% calculated percentages.

houses_mul, houses_cat, and houses_add: I tried three ways of combining both ***MultiImgsCNN*** and ***LSTMModel*** outputs in the class ***Model***. These models all use metadata and images. In general, the model performed better with the multiplication than with concatenation or addition as we see more than 10% improvement in precision, recall and F1 score. Consequently, the rest of the trained models were all multiplied.

houses_NoZC, houses_NoBR, houses_NoAr, and houses_NoRN: It was substantial to investigate whether one of the features is causing the models to perform poorly to eliminate what could be the reason behind it. As we can see in the table all models performed sort of equally when excluding each of the four features, one at a time. Thus, no feature was eliminated or excluded in the final model.

houses_60: This is the final and main model I chose for this project, where I made several significant changes one of them was changing the number of epochs to 60. As stated previously, the architecture of this model is updated to catch as complex features as possible, therefore improve performance. The accuracy metrics suggest that the model is performing well on both positive and negative classes, although I believe there is still room for improvement. As the table shows, accuracy is high and so are precision and then (relatively) recall. This is interpreted as 92% of all predicted classes were correctly classified, and the ratio of true predicted positives to all predicted positives is 86.19 while the ratio of true predicted positives to all actual true positives is 80.72%. This indicates that the model is not producing many false positives nor false negatives and the F1 score suggests that the model is well balancing precision and recall.

Model	***Description***	***Accuracy***	***Precision***	***Recall***	***F1 Score***
houses_imgs	Images only classification model	37.5%	27.22%	28.71%	22.93%
houses_feat	Tabular data only classification model	87.5%	56.25%	62.5%	59.05%
houses_mul	Images and tabular data models combined by multiplying outputs	90.625%	71.33%	66.67%	66.8%
houses_cat	Images and tabular data models combined by concatenating outputs	88.28%	51.56%	56.25%	53.7%
houses_add	Images and tabular data models combined by adding outputs	86.72%	49.22%	54.17%	51.45%
houses_NoZC	Both models outputs multiplied excluding Zipcode	83.59%	50.81%	47.83%	47.89%
houses_NoBR	Both models outputs multiplied excluding no. of Bathrooms	92.97%	58.98%	62.5%	60.63%
houses_NoAr	Both models outputs multiplied excluding Area	89.84%	58.47%	61.19%	59.76%
houses_NoRN	Both models outputs multiplied excluding no. of bedrooms	86..72%	57.03%	56.94%	55.31%
houses_60	Both models with 60 epochs and new ***MultiImgsCNN*** architecture	92.19%	86.19%	80.72%	81.89%

In all the of the above explained models the learning rate was tested between mostly 0.01, 0.003, 0.001 and some other values, but ended up with 0.003 as it has shown best performance.

Conclusion

The final model has performed relatively well but it would be much better if we can further improve it by exploring different architectures until we arrive to the point where we feel satisfied with it. The idea to use images in addition to tabular data is a very promising and is in fact more realistic than using either of both. Both data types contribute significantly to the customers' decisions when buying a house. We should not forget that each demographic place needs its own model as it is governed by its own conditions and preferences. It was a bit of struggle to find a model that performs well, and this is due to several reasons in my opinion. First the dataset we used is indeed unique yet considered small, and for such a model to learn and then work properly we need let's say double the data we used in this project. Second, the classes could have been determined differently as we have seen that there are at least one major class in the created classes which is causing an imbalance in our data. One way of doing it would be to check the number of instances in each current class and put a different plant to choose the boundaries based on the results. The Encoding of ***Zipcode*** could be done using one hot encoding and the results could have been different as the model won't be dealing with integers anymore and consider prioritizing the values of these integers for examples. We could also apply other transformation on the images or even replace some. LSTM might not be the ideal neural network to use for such models, I might try others like FFNN for the MetaData. So many scenarios could be investigated to reach the best model possible, but that is to be tested in the future.

References

Eman H. Ahmed, M. N. (2016). *House price estimation from visual and textual features.* Cairo, Egypt.

Armin Kappeler, S. Y. (2016). *Video Super-Resolution With Convolutional Neural Networks.* IEEE TRANSACTIONS ON COMPUTATIONAL IMAGING.