PRICE RECOMMENDER SYSTEM:

Machine learning for revenue management

ROBERT GRAY - 10389794@mydbs.ie

Supervisor: Dr Shahram Azizi Sazi

Dublin Business School

This Bachelor Thesis is submitted for the degree of
BSc (Hons) in Computing

January 2019

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis is original and has not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This thesis is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and acknowledgements.

ROBERT GRAY - 10389794@mydbs.ie
January 2019

Acknowledgements

Having spent the last 15 years working in IT and Software Development, the lack of a formal degree was a gaping hole in my portfolio of skills and experience. With a single income and a large family to support, further education was a luxury I could not easily afford. So I would first of all like to express my sincere gratitude to Dublin Business School and the Springboard+ up skilling initiative in higher education for this opportunity and the financial support that covered the tuition fees.

I would also like to thank the lecturers at Dublin Business School for their support and encouragement throughout the degree program. I would specifically like to express my gratitude to Dr. Shahram Azizi Sazi for his guidance during this project and thesis.

I would like to give a special mention to Mr. Paul Laird at Dublin Business School. His high expectations pushed us to achieve at the highest level. Sometimes it might have seemed like he was making it too hard. But it meant we were always stretching and pushing ourselves to be better. I am very thankful for the data scientist I have become as a direct result of his mentorship.

I would also like to thank my employer Topflight and specifically its CEO, my boss, Anthony Collins. When I needed time off from work to study for exams there was no hesitation. This allowed me to better use my annual leave for research and development.

Last, but definitely not least, I would like to thank my wife Kim and our three awesome children for their patience and support throughout what has been a difficult year for them. It's through their support and encouragement that I was able to put in the time needed to achieve at the highest level.

Abstract

Research is conducted on the use of supervised machine learning algorithms, automated machine learning, ensemble pipelines, price prediction and price automation in revenue management as well as the architectures for serializing and exposing machine learning models for consumption by external systems.

Various machine learning models are trained using ensemble, boosted and stacked pipelines as well as automated machine learning to produce a base model that is 84% accurate in predicting the selling price of holidays based on the features of the holiday.

A software system is developed that can train machine learning models using preset model parameters or leveraging automated machine learning to select the best parameters. The software system exposes the trained models via a REST API, for consumption by third party systems, when recommending the selling prices of holidays and other products in the presence of perishable inventory considerations.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Overview

Revenue management, that centers around a given and finite selling season (or time horizon) with perishable goods and that follows a stochastic model where the demand for a product is unknown, tries to take into account multiple variables when setting the price a product should be sold for.

Focusing primarily on the Travel Industry, where revenue management plays an important role in setting selling prices, the following sections will give an introduction to the concepts of revenue management, machine learning and price prediction as well as define the scope for this project and Bachelor thesis.

1.2 Revenue Management

Revenue management, sometimes called yield management, is believed to have started out in the airline industry (Lieberman, 2011). As the practice of yield management spread beyond the airline industry the term changed to revenue management. Revenue management is generally considered the art or discipline of forecasting customer demand and available supply and using that information to control product availability and pricing in such a way so as to maximize profits and at the same time minimize the quantity of unsold committed product.

Pricing problems in revenue management center around a given and finite selling season (or time horizon). Holidays are time-sensitive and perishable with a fixed selling season where unsold product carries a cost. The holiday market typically follows a stochastic model in which, given the prices of all the products, the demand for each product is unknown, but follows a known distribution (Chen and Chen, 2015).

1.2.1 The Travel Industry

Revenue management has now attained a prominent position among the strategies and tactics travel company's use to obtain a competitive advantage over their rivals (Lieberman, 2011). A travel companies ability to source and strategically price its product is critical to its success. Traditional pricing models try to take into account seasonality, competition, cost price, margins and many other variables in order to determine the best selling price. Traditionally that price is a best guess as to what the market is willing to pay and it is quite often wrong. This leads to heavy discounting and negative margins closer to the date of travel in order to shift underselling committed stock.

We would like to know how much buyers value our product, but more often than not we must just guess at this number (Talluri and Ryzin, 2004). On one hand, revenue managers would like to sell their products to those customers who have a high valuation so that high margins can be achieved. On the other hand, if they wait too long for those high valuation customers to appear, they might end the selling period with unsold units that could have been sold to low valuation customers (Bitran and Caldentey, 2003).

1.2.2 Other Industries

While not the focus of this project and thesis, it is fair to say that the travel industry has similarities to other industries in terms of its need to manage unsold product. Take the fashion industry as a novel example. Some fashion brands/companies also follow a fixed selling season with unsold product carrying a cost. In 2017, Burberry, the upmarket British fashion label, destroyed unsold clothes, accessories and perfume worth £28.6m. While these goods were not naturally perishable, they had a fixed selling season after which they could (arguably) no longer be sold. Cutting prices of underselling stock de-values the brand and some industries are willing to take extreme measures and losses to avoid diluting revenue prospects.

1.3 Machine Learning

Traditional software engineering requires a software engineer to interpret and understand the machine rules required and translate them into application logic. Machine learning goes a step further in that it automates the task of writing the machines rules. Typically, a software engineer or data scientist passes the inputs and expected outputs to the machine and asks the machine to identify the rules that would determine an output for a given input. The machine produces a model containing these rules and the weights placed on each of them in such a way that it can "predict" the same outputs knowing only the inputs, with some measure of accuracy and

some degree of error. This is known as machine learning. Machine learning is often applied on problems that would be too complicated for humans to figure out with too many rules for a software engineer to interpret and develop by hand.

1.3.1 Applying machine learning to revenue management

Most commerce systems maintain a record of historical sales data that includes the product that was sold and the price that it was sold for as well as a wealth of data that is potentially correlated to some degree with the selling price. This data can be interpreted as the price a customer was willing to pay for a product at that time based on the features and demand curve of the product.

It is anticipated that by applying supervised machine learning to revenue management, that it would be possible to train machine learning models on historical sales data, that would be accurate at predicting future selling prices. These predictions can then be used to aide in recommending or perhaps even automating the future selling price of a product in order to minimise undersold product and maximise profit.

1.4 Project Scope & Deliverables

The output of this project and bachelor thesis is a software system in the form of a production ready Price Recommender API complete with training and prediction end points as well as a separate front-end GUI application for training and demonstration purposes. Its expected use would be in integrating with a revenue or product departments pricing & inventory systems allowing the ability to query the Recommender API for price recommendations when both contracting and pricing up their products.

The following chapters will document the entire project and will cover the research done, prototypes, supervised machine learning models, automated machine learning experiments, application design, software development as well as getting it all onto a production server on Google Cloud Compute.

It will answer the following research questions

- Can we train machine learning models to be accurate enough for us to use them for price recommendation.
- Can we develop an application for training machine learning models that can be operated by an end user with no data science knowledge.

- Can we further develop a training application in such away so that it can be used to train different models for different domains with different datasets without significant additional development.
- Can we use automated machine leaning to do the heavy lifting when it comes to model type selection, pre-processing, feature selection and hyper parameter tuning.

This thesis stops short of integrating the recommender with external systems until after submission and grading in order to protect the commercial potential for the application and it's applicability to domains outside of the target/research domain.

Chapter 2

Background

2.1 Inspiration

A machine learning model that I previously developed used a Random Forrest Regression (Breiman, 2001) model to predict the price a holiday is likely to sell for. The original paper is available as a non-peer reviewed self published paper (Gray, 2018). In being able to demonstrate the applicability of machine learning in holiday selling price prediction I was inspired and encouraged to use that knowledge to form the basis for this Bachelor thesis project.

2.2 Current machine learning approaches

Machine learning and other statistical regression and classification methods have been used to predict Airfares (Groves and Gini, 2011), recommend hotels (Li, 2018) and in dynamic pricing (Elmaghraby and Keskinocak, 2003) but recommending selling prices using machine learning, to my knowledge has had little or no publications. For this reason my research looks at the currently available literature on machine learning, price prediction, regression, revenue/yield management, dynamic pricing, and software development architectures for persisting trained models into production.

2.3 What problems are we trying to solve?

Taking a tour operator as an example, the company has to first contract and negotiate with its suppliers and agree the rates or costs of the components of the holiday. Those rates or costs turn into a cost price that should, when desired margins and overheads are taken into account,

be at a price point the customer is willing to pay. This price point is not always known at the crucial time of negotiating the best rates or costs with the supplier.

Similarly when a revenue manager is setting the selling price for a product, revenue management becomes critical. Pricing decisions are often made based on "intuition" and "rule of thumb" practices with many adjustments needed throughout the selling season to compensate for the inaccuracies and seemingly random variables.

2.4 Who will benefit?

The sale of time sensitive and perishable goods generates a profit by getting the best cost price and maximizing margins over a finite selling season.

Again taking a tour operator as an example, it is anticipated that having a tool that can recommend a selling price would improve margins, reduce undersold stock and help with contract negotiation. "The benefits of yield management are often staggering; American Airlines reports a five-percent increase in revenue, worth approximately $1.4 billion dollars over a three-year period, attributable to effective yield management" (Gallego and Ryzin, 1994).

As mentioned in the introduction there will be a focus on the travel industry and the selling of package holidays. This is because I currently work in the travel industry for a tour operator and have access to industry data and domain knowledge that will help in understanding the real world applications of this technology. However the project itself will not be limited to that specific domain and in fact I will demonstrate using the same software application in predicting or recommending house prices in my conclusions.

Chapter 3

Literature Review & Industry Research

3.1 Current Machine Learning Approaches

Predicting or recommending a selling price based based on features is a regression problem. According to (Tziridis *et al.*, 2017) in predicting Airfare prices using machine learning techniques, "ML models are able to handle this regression problem with almost 88% accuracy, for a certain type of flight features". The results of their findings represents the state of the art for this type of regression problem. The researchers used a 10-fold cross-validation procedure to train specific ML models. Prediction accuracy and Execution Time is shown in Table 3.1. It is anticipated that similar results would be possible, where the features are airfare and accommodation features, as the authors had in predicting airfares alone.

According to (Tziridis *et al.*, 2017) in predicting airfare prices, “Bagging Regression Tree”, “Random Forest Regression Tree”, “Regression Tree” and MLP models are the most stable

Table 3.1 Airfare Prices Prediction Using Machine Learning Techniques (Tziridis *et al.*, 2017)

ML Model	Accuracy (%)	Execution Time (sec)
Multilayer Perceptron	72.8	5.98
Generalized Regression Neural Network	66.14	0.34
Extreme Learning Machine	64.88	0.06
Random Forest Regression Tree	86.15	6.15
Regression Tree	84.22	0.059
Bagging Regression Tree	87.93	15.34
Regression SVM (Polynomial)	77.91	0.17
Regression SVM (Linear)	57.69	0.06
Linear Regression	57.92	0.02

models according to their accuracy scores. In addition, as far as the execution time is concerned the best models are “Random Forest Regression Tree” and “Regression tree”.

3.2 Programming languages and libraries for machine learning

Both Python and Java were considered as suitable programming languages for this project and thesis. According to (towardsdatascience, 2017), 57% of Data Scientists prefer to use Python however Enterprise developers tend to use Java. Python has a wealth of recommended machine learning libraries including scikit-learn (Pedregosa *et al.*, 2011) and tensorflow (Abadi *et al.*, 2016). Java also has some well documented machine learning libraries such as java-ml (Abeel and Saeys, 2009) and jsat (Raff, 2017). Java however is less efficient when it comes to prototyping due to the amount of boilerplate code required, specifically when operating on data. (Prechelt, 2000) suggests that designing and writing a program in Python takes half as much time as writing it in Java. Newer versions of Java (8+) introduced language features like lambdas and streams which have gone someway in bridging this code writing efficiency gap.

3.3 Automatic Machine Learning Algorithm Selection

A novel approach to model selection would be to use a machine learning algorithm to select the best machine learning algorithm. Automated machine learning uses an algorithm to automate the selection of the best machine learning pipelines though trial and error whilst trying to minimize the number of trials. Automated machine learning (AutoML) has the potential to make it much easier for non-experts and experts alike, to select machine learning algorithms, their parameter settings, pre-processing methods and feature selection. AutoML Libraries that were researched for this project are TPOT and AUTO-SKLEARN.

3.3.1 AutoML using TPOT

TPOT is a Python library developed for automatic machine learning feature preprocessing, model selection, and hyperparameter tuning. It uses genetic programming to try to find the best machine learning pipeline for a dataset by evaluating thousands of possibilities.

TPOT can "design machine learning pipelines that provide a significant improvement over a basic machine learning analysis while requiring little to no input" and "address the tendency for TPOT to design overly complex pipelines by integrating Pareto optimization, which produces compact pipelines" (Olson *et al.*, 2016).

TPOT uses the following regressors as part of it's model selection: Elastic Net (Zou and Hastie, 2005), Extra Trees Regressor (Geurts *et al.*, 2006), Gradient Boosting Regressor (Friedman, 2001), AdaBoost Regressor (Freund and Schapire, 1999), Decision Tree Regressor (Z and Lior, 2014), K-Neighbors Regressor (Hastie and Tibshirani, 1996), Lasso Lars (Efron *et al.*, 2004), Linear SVR (Ho and Lin, 2012), Random Forest Regressor (Breiman, 2001), Ridge Regressor (Hoerl and Kennard, 1970) and Extreme Gradient Boosting Regressor (Chen and Guestrin, 2016).

3.3.2 AutoML using AUTO-SKLEARN

AUTO-SKLEARN is also a Python library developed for automatic machine learning. It leverages recent advantages in Bayesian optimization, meta-learning and ensemble construction to try to find the best machine learning pipeline. AUTO-SKLEARN can automatically construct ensembles of machine learning models that are more robust (and less prone to overfitting) than the standard hyperparameter optimization of a single model (Feurer *et al.*, 2015).

AUTO-SKLEARN uses the following regressors as part of it's model selection: Extra Trees Regressor (Geurts *et al.*, 2006), Gradient Boosting Regressor (Friedman, 2001), AdaBoost Regressor (Freund and Schapire, 1999), Decision Tree Regressor (Z and Lior, 2014), K-Neighbors Regressor (Hastie and Tibshirani, 1996), SVR & Linear SVR (Ho and Lin, 2012), Random Forest Regressor (Breiman, 2001), Ridge Regressor (Hoerl and Kennard, 1970) and Extreme Gradient Boosting Regressor (Chen and Guestrin, 2016), Stochastic Gradient Descent (SGD) Regressor (Bottou, 2010) and Bayesian ARD Regressor (MacKay, 1994)

3.3.3 TPOT & AUTO-SKLEARN Comparison

Both TPOT and AUTO-SKLEARN use the scikit-learn: (Pedregosa *et al.*, 2011) library of machine learning algorithms. Both libraries only support running on Linux operating systems and due to the amount of time required for model selection (days to weeks), they may not be suitable for quick model selection or training. While AutoML simplifies the model selection process it can also over complicate the practical implementation of machine learning and also perhaps lead to a convoluted production environment.

According to (Balaji and Allen, 2018) in Benchmarking Automatic Machine Learning Frameworks, AUTO-SKLEARN performs the best across classification datasets and TPOT performs the best across regression datasets. One of the important features required of both frameworks is the ability to export the selected pipeline at the end of the selection process. TPOT has a feature to export the best pipeline as python code that can be manipulated and

further refined. However AUTO-SKLEARN does not have the same capability and is limited to being able to persist the best model in volatile memory or serialise to disk by other means.

3.4 Ensembles - bagging, stacking and boosting

Using an ensemble of average learners can sometimes produce more accurate and stable models than a single strong learner. Random Forest Regression Tree is one such example. It is an ensemble of decision trees where the algorithm consists of independently growing decision trees based on different subsets of training data. The randomness refers to the random sampling of the training data otherwise known as Bagging. Similarly each tree uses a random sample of the training features which can further reduce bias and improve generalization. Each tree in the ensemble votes on feature importances and a consensus is reached as to the feature importances/weights.

Other similar ensemble approaches include boosting and stacking. In boosting, the subset creation is not random and depends upon the performance of the previous models. Stacking differs from bagging and boosting in that it is normally used to combine models of different types (Witten *et al.*, 2016).

According to (Dietterich, 2000) on Ensemble Methods in Machine Learning, AdaBoost often gives the best results. Bagging and randomized trees give similar performance, although randomization is able to do better in some cases than Bagging on very large data sets.

3.5 Existing Recommender Systems

"Most existing research in recommender systems overlooks a factor that is extremely important in a consumer's decision making: the product price." (Zhao *et al.*, 2015). When it comes to pricing in e-commerce, to my knowledge there is no one using machine learning to recommend a selling price in the way I am proposing. There is however systems for automating pricing usually based on competitor website price scraping and comparison. For example wiser.com and prisync.com claim to offer educated pricing changes based on competitive analysis.

The only existing price recommender system I can find on the market that uses machine learning or AI is perfectprice.com. They work with Uber on recommending price changes using AI and offer services for other car rental companies. Again these solutions are very much based on competitive analysis as opposed to predicting what the customer is willing to pay based on perceived value of the product features.

3.6 UI/UX

UI/UX is about user experience and designing user interfaces in intuitive ways. Rather than reinvent the wheel I opted to research web component based frameworks. According to geekflare.com, the best frameworks are Twitters Bootstrap and Zurbs Foundation. Bootstrap has the advantage of having a VueJS library that is well developed and documented which would further simplify the front-end implementation.

A dashboard template that uses bootstrap-vue is coreui.io. It has some useful components for developing dashboard type user interfaces.

In general none of the frameworks catered for presenting statistical data so an additional library would be required.

3.6.1 Presenting Statistical Data in Web applications

Web applications are typically not very good at displaying statistical data for very large datasets without the help of a javascript library. You will often see very summarized data or simple charts in web application dashboards. This is mostly down to the performance limitations of the browser and its ability to render large amounts of data. The design of the training application requires just a single scatter plot to show the predicted versus actual with a regression line, however this scatter plot could have thousands of values. According to (Barnard and Mertik, 2015) report on the Usability of Visualization Libraries for Web Browsers for Use in Scientific Analysis, Bokeh has been shown to be the best choice out of all of the available options.

Bokeh is an interactive visualization library that targets modern web browsers. It delivers complex visualizations over very large or streaming datasets. Plots can be generated in python and then embedded in the browser using a mixture of javascript and JSON. This is important, as you will see later in the development of the application. Other libraries considered were d3js and canvasjs however both struggle when plotting more than a few hundred points on a scatter plot.

3.7 Putting Machine Learning Models Into Production

Python has functionality for persisting models from volatile memory to the file system called pickling. A pickled model can then be loaded back into volatile memory at runtime. However pickling can struggle with complex models like that of a machine learning model. For that reason I researched alternatives and came across joblib.

Persisting in an efficient way arbitrary objects containing large data is hard. Using joblib's caching mechanism avoids hand-written persistence and implicitly links the file on disk to the

execution context of the original Python object. As a result, joblib's persistence is good for resuming an application status or computational job such as a machine learning model.

Using a virtual environment with its own Python binary (that matches the version of the binary that was used to create the environment and persist the model) requires some careful dependency management. The virtualenv python module provides support for creating "virtual environments" that isolate application dependencies, but they are not portable when moving from the development environment to a production environment.

For this reason i considered the use of docker. Docker uses virtualisation to containerise software. This allows us to package our application, it's dependencies, operating system and python runtime into a single docker image that is fully portable to any host machine that can run docker. Also by leveraging orchestration technologies like Ansible and Chef combined with DevOps style CICD pipeline we can fully streamline development to production of our software and machine learning models. The final consideration is the Linux requirement when developing using the AutoML frameworks. Using docker it's possible to satisfy these requirements on any machine that can run docker during development.

Chapter 4

System Specification & Design

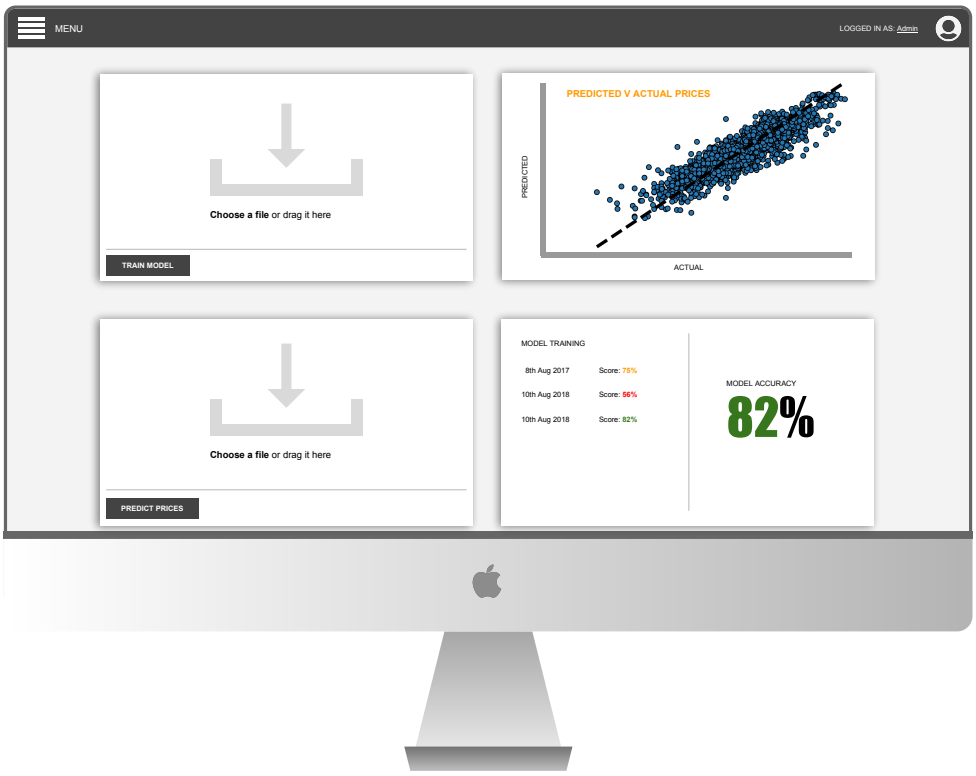


Fig. 4.1 Mock-up Of Model Training Application

4.1 System Specification

The Price Recommender System is a software application developed for recommending selling prices to external e-commerce systems. As it needs to be easily integrated with external systems,

the application would be split up into two layers with the back-end being a modern RESTfull Web Service API and the front-end being a modern reactive Web Application for training the models and demonstrating it's price recommendation capabilities.

A client-server architecture, where the back-end is separate to the front-end, using Web Services offers better scalability and distribution capability than more traditional monolithic architectures. The REST architectural style is commonly applied to the design of API's for modern web services. A Web API conforming to the REST architectural style is said to be a REST API (Masse, 2011).

For simplicity the back-end server will be called the "REST API" and the client will be called the "Training Application". When talking about both the client and server the term "Price Recommender" will be used.

4.1.1 Functional Requirements

1. The recommender systems back-end should be implemented in the form of a REST API so that it can be easily integrated with third party systems.

2. The REST API will need to implement a number of selected machine learning models as well as automated machine learning and serve up end points for authentication, CSV file upload, model training, model training history and price prediction.

3. It should be possible for a non-experts to load in historical sales data via the Training Application, in order to train and retrain the models.

4. It should be possible to delete old serialised model files, via the Training Application, that are no longer needed without deleting the training history

5. The Training Application should have a scatter plot of predicted versus actual values

6. The Price Recommender should be easily adapted to work outside of the target company/domain. i.e. be trainable on different data where there is a similar discrete target variable that is correlated with the feature variables.

7. The Price Recommender should be delivered using the latest software development architectures and methodologies around continuous deployment. i.e. using virtualisation, docker containers, orchestration, automation, etc

8. The user interface of the Training Application should be delivered using modern reactive web application technologies in the form of a single page application.

4.1.2 Non-Functional Requirements

1. The REST API should be sufficiently performant so as to allow quick retrieval of price recommendations in under 1 second
2. The price predictions should be sufficiently accurate so as to provide confidence in decisions made regarding product and pricing.
3. It should be possible to make test predictions using the Training Application without having to fill in every field manually

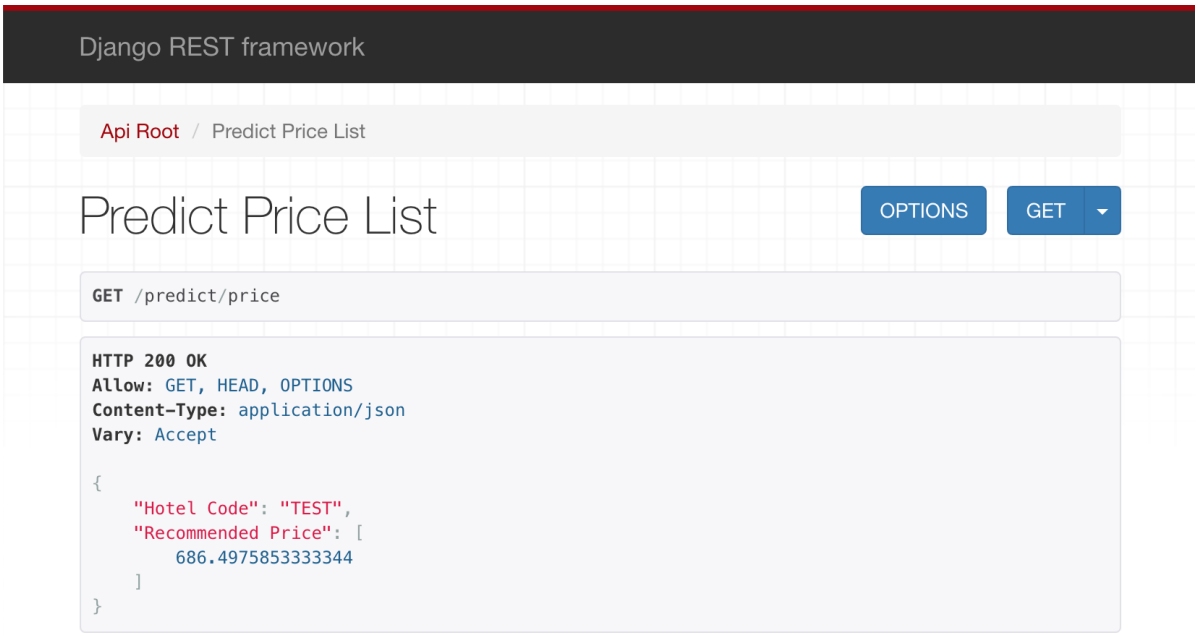


Fig. 4.2 Screen-shot of Prototype API with working prediction endpoint

4.2 System Design

As the application would be split into two separate layers there is no requirement for both layers to use the same programming languages or frameworks. This opens up the possibility of selecting programming languages and frameworks based on suitability for the task at hand. For example Javascript is the programming language of the modern web browser so it makes sense to select it for the front-end development of the training application.

4.2.1 REST API

The REST API would be developed using the Python programming language. The primary reason for choosing Python is it's capability with regard to machine learning. While other programming languages like Java have machine learning libraries and frameworks, the community and academic literature is not as prevalent and the amount of boilerplate code required is significantly more.

In order to bootstrap the API, a python based REST framework called Django REST (Django, 2018) would be used. Django has some nice capabilities out of the box such as a basic web interface for testing your endpoints as well as modules for authentication. Figure 4.2 shows an early screen-shot of the Django Rest web interface with a working prototype test endpoint for predicting prices. Other frameworks considered were Flask, Pyramid, Tornado and Bottle. Django was selected for it's bootstrapping capabilities, documentation and popularity.

4.2.2 Training Application

The training application would be built built using Javascript and the VueJS framework. The VueJS framework provides for reactive DOM rendering allowing the creation of a single page application that updates the page without reloading following the users interactions. Figure 4.1 shows an early mock-up of the user interface. The user interface would consist of components for model training, price prediction, model accuracy and model training history.

4.2.3 Model Selection & AutoML

In addition to the prototype model other models would be added following experimentation with different machine learning model types and hyper parameters. Additionally AutoML or automated machine learning would be experimented with as a means for selecting even better performing machine learning models. The best performing models would be integrated with the training application allowing the user to select from these predefined models for quick

training solutions. The user would also be able to select to train the model using automated machine learning in situations where model training was not time critical or the best model was not one of the predefined ones.

4.2.4 Database

There is a requirement to store the model training history including the steps used during training so the that during prediction these steps can be replayed. The database schema requires only a single table with very few columns and rows. The Django REST framework has its own database schema requirements for login & security and uses a SQLite database file by default. As there would be no requirement for anything more sophisticated than this, a SQLite database would be used.

SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine. SQLite is the most used database engine in the world. It is built into all mobile phones and most computers and comes bundled inside countless other applications that people use every day (sqlite.org, 2019)

Chapter 5

Machine Learning Model Selection

5.1 Overview

The following chapter is about an approach to model selection for the target company's data. It explains where the data was sourced, the prototype model, automated machine learning platforms and how automated machine learning proved better at selecting the best model than a traditional data science approach. Experiments are undertaken to develop this approach that would then be used in the software application being developed.

5.2 Data

The dataset used for training and selecting the best machine learning models was sourced from an Irish Tour Operator. It relates to one of the company's brands. This brand is focused on the sale of European Sun Package Holidays. It contains an exported view of booking data and holiday data over 5 years between 2013 and 2017.

The booking related data contains the price per person the holiday was sold for, the date it was sold, the date of travel and how many days before travel the holiday was booked. The holiday related data contains a limited set of important features associated with the accommodation, destination and departure point of the package holiday. The data was exported as a single table in CSV format with some early consideration for categorical variables that were converted to boolean features as part of the SQL database query. Table 5.1 shows the features, a brief description and how each field is typically processed during training and model selection.

Please note, this dataset is confidential and will not be made publicly available so as to not expose the source companies pricing models to competitors.

Table 5.1 Table of training data features

feature	type	description	processing
accom_id	categorical	identification	dropped
accommodation	categorical	Property Name	encoded
accom_location	categorical	Property Location	encoded
destination	categorical	Holiday Destination	encoded
accom_stars	integer	Property Star Rating	none
accom_type	categorical	Property Type	encoded
accom_board_basis	categorical	Meal board basis	encoded
staff_pick	integer	Recommended by staff	none
trip_adv_rating	float	Rating on Trip Advisor	none
trip_adv_reviews	float	Number of reviews on Trip Advisor	none
has_swimming_pool	integer	Has a swimming pool	none
has_sauna	integer	Has a sauna	none
has_jacuzzi	integer	Has a jacuzzi	none
has_tv_in_room	integer	has a tv in the room	none
has_air_conditioning	integer	has air conditioning	none
has_wifi	integer	has WiFi	none
has_hot_tub	integer	has a hot tub	none
has_lift	integer	has a lift	none
suitable_for_children	integer	is suitable for children	none
has_child_care	integer	has child care facilities	none
has_bar	integer	has a bar	none
has_sea_view	integer	has a sea view	none
close_to_resort	integer	close to the resort	none
departure_airport	categorical	departure airport	encoded
travel_date	categorical	date of travel	week extracted
booking_date	categorical	date booked/sold	week extracted
day_booked_before_travel	integer	days before travel holiday was booked	none
holiday_duration	integer	duration of the holiday	none
price_per_person	float	average price per person	target

5.3 Prototype Model

A Random Forrest Regressor was fitted to the training data using a python package from Sklearn. This yielded an initial R-Squared of 0.81. Attempts were made to further optimize the model using automated hyper parameter tuning, feature engineering, scaling etc. The final prototype or base model, using Random Forest Regression Tree, comparing predicted v actual (ground truth) had an R-Squared value of 0.8233. Figure 5.1 shows a plot of predicted v actual prices.

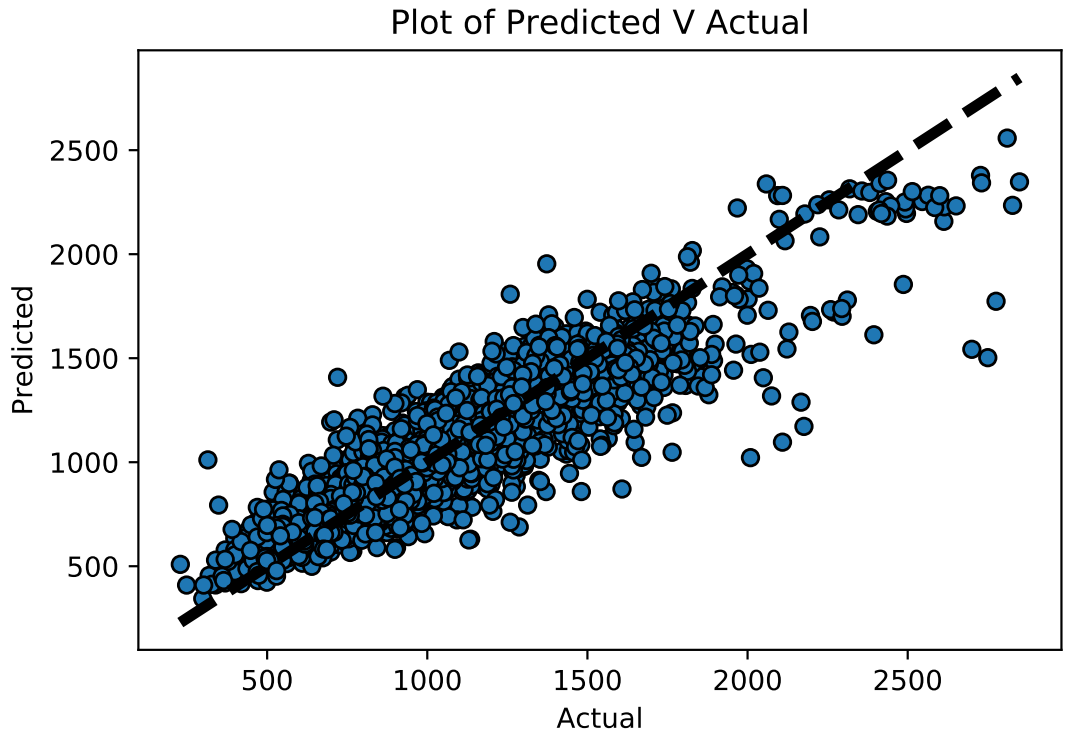


Fig. 5.1 Plot of predicted v actual values from prototype model

5.4 Automated Machine Learning Platforms

All model training and model selection was done using Jupyter Notebooks and the Python programming language. While Keras and Tensorflow were considered, Sklearn was selected for it's simplicity and extensive documentation. For Automated Machine Learning two Libraries were considered, TPOT and AUTO-SKLEARN. Neither TPOT nor AUTO-SKLEARN are supported on my development platform of OSX. For this reason I used either Google Colaboratory or a Docker container with a Linux operating system.

5.4.1 Google Colaboratory

Google Colaboratory is a free service from Google that allows for running Jupyter Notebook type experiments in the cloud with the power of Google's infrastructure. This includes access to GPU's and TPU's if you use a library like Keras or Tensorflow that can take advantage of them. There are some limitations, the biggest of which is that Google can shutdown your experiment at any time. Unfortunately when the experiment is shutdown all progress is lost. For my long running experiments with AutoML using TPOT I used this service and worked around it's limitations. Ideally, with more resources, this could be done more reliably with solutions from the likes of Azure, AWS, Cloud Compute or access to a super computer.

5.4.2 Docker Container with Linux

A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. (www.docker.com, 2018). AUTO-SKLEARN has very specific requirements so a local docker container running a Jupyter Notebook with the necessary requirements was used.

5.4.3 Apple MacBook Pro

When not using AutoML for model selection, a 2013 Apple MacBook Pro with MacOS Mojave, running Jupyter Notebook was used to refine the model training code prior to integration into the application.

5.4.4 Model Selection using TPOT Library

Figure 5.2 shows the output following 8 hours of automated machine learning and the best fitting model during that time frame. This was run numerous times with varying success in order to select the best performing model with the TPOT Library on a Google Colaboratory Notebook.

5.4.5 Model Selection using AUTO-SKLEARN Library

Figure 5.3 shows the output following 8 hours of automated machine learning and the best fitting model during that time frame. This was run in order to select the best performing model with the AUTO-SKLEARN Library in a Jupyter Notebook running inside a docker container with Linux.

```
# Fit the tpot model on the training data
tpot.fit(X_train, y_train)
```

```
Optimization Progress: 100%|██████████| 200/200 [18:30<00:00,  8.65s/pipeline]Generation 1 - Current best internal CV score: 0.81
Optimization Progress: 100%|██████████| 300/300 [33:24<00:00, 11.83s/pipeline]Generation 2 - Current best internal CV score: 0.81
Optimization Progress: 402pipeline [58:09, 17.16s/pipeline]Generation 3 - Current best internal CV score: 0.8185335750643145
Optimization Progress: 502pipeline [1:18:05, 11.45s/pipeline]Generation 4 - Current best internal CV score: 0.8185335750643145
Optimization Progress: 603pipeline [1:40:14, 13.36s/pipeline]Generation 5 - Current best internal CV score: 0.8185335750643145
Optimization Progress: 703pipeline [1:56:20,  9.00s/pipeline]Generation 6 - Current best internal CV score: 0.8215408070330295
Optimization Progress: 805pipeline [2:21:00, 12.99s/pipeline]Generation 7 - Current best internal CV score: 0.8216392535917801
Optimization Progress: 908pipeline [2:51:19, 15.40s/pipeline]Generation 8 - Current best internal CV score: 0.8216392535917801
Optimization Progress: 1008pipeline [3:07:48,  8.80s/pipeline]Generation 9 - Current best internal CV score: 0.8216392535917801
Optimization Progress: 1111pipeline [3:44:16, 26.66s/pipeline]Generation 10 - Current best internal CV score: 0.8216712773489956
Optimization Progress: 1214pipeline [4:16:42, 10.32s/pipeline]Generation 11 - Current best internal CV score: 0.8235303989222533
Optimization Progress: 1316pipeline [4:48:54, 14.28s/pipeline]Generation 12 - Current best internal CV score: 0.8235303989222533
Optimization Progress: 1419pipeline [5:27:31, 15.25s/pipeline]Generation 13 - Current best internal CV score: 0.8253257006205461
Optimization Progress: 1520pipeline [5:59:52, 14.86s/pipeline]Generation 14 - Current best internal CV score: 0.8253311469977141
Optimization Progress: 1622pipeline [6:34:44, 17.36s/pipeline]Generation 15 - Current best internal CV score: 0.8261381037664386
Optimization Progress: 1723pipeline [7:03:33, 14.60s/pipeline]Generation 16 - Current best internal CV score: 0.8261381037664386
Optimization Progress: 1826pipeline [7:43:38, 17.67s/pipeline]Generation 17 - Current best internal CV score: 0.8261381037664386

481.2082275833333 minutes have elapsed. TPOT will close down.
TPOT closed prematurely. Will use the current best pipeline.

Best pipeline: ExtraTreesRegressor(XGBRegressor(input_matrix, learning_rate=0.1, max_depth=5, min_child_weight=6, n_estimators=10
TPOTRegressor(config_dict=None, crossover_rate=0.1, cv=5,
       disable_update_check=False, early_stop=None, generations=1000000,
       max_eval_time_mins=5, max_time_mins=480, memory=None,
       mutation_rate=0.9, n_jobs=-1, offspring_size=None,
       periodic_checkpoint_folder=None, population_size=100,
       random_state=42, scoring='r2', subsample=1.0, use_dask=False,
       verbosity=2, warm_start=True)
```

```
print(tpot.score(X_test, y_test))
```

```
0.8392594136698323
```

```
# Show the final model
print(tpot.fitted_pipeline_)
```

```
Pipeline(memory=None,
     steps=[('stackingestimator', StackingEstimator(estimator=XGBRegressor(base_score=0.5, booster='gbtree', colsample_bylevel=1,
       colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_step=0,
       max_depth=5, min_child_weight=6, missing=None, n_estimators=100,
       n_jobs=1, nthread=1, ob...timators=100, n_jobs=1,
          oob_score=False, random_state=None, verbose=0, warm_start=False))])
```

```
# Export the pipeline as a python script file
tpot.export('tpot_exported_pipeline.py')
```

Fig. 5.2 Screenshot of Google Colab Notebook during TPOT Model Selection

```
In [7]: X_train, X_test, y_train, y_test = train_test_split(
            X,y, test_size=0.2, random_state=42)

        automl = autosklearn.regression.AutoSklearnRegressor(
                    time_left_for_this_task=28800,
                    per_run_time_limit=3600,
                    tmp_folder='/tmp/autosklearn_regression_example_tmp',
                    output_folder='/tmp/autosklearn_regression_example_out',
                )
        automl.fit(X_train, y_train)

        print(automl.show_models())
        predictions = automl.predict(X_test)
        print(automl.sprint_statistics())
        print("R2 score:", sklearn.metrics.r2_score(y_test, predictions))
        dataset_properties={
          'multilabel': False,
          'sparse': False,
          'multiclass': False,
          'signed': False,
          'target_type': 'regression',
          'task': 4})),
        ]

        auto-sklearn results:
          Dataset name: ff21e8189e6ea1a85c0a95b66b2e3da5
          Metric: r2
          Best validation score: 0.833786
          Number of target algorithm runs: 194
          Number of successful target algorithm runs: 0
          Number of crashed target algorithm runs: 0
          Number of target algorithms that exceeded the memory limit: 0
          Number of target algorithms that exceeded the time limit: 0

        R2 score: 0.840080316746
```

Fig. 5.3 Screenshot of Jupyter Notebook during AUTO-SKLEARN Model Selection

Chapter 6

Software Development

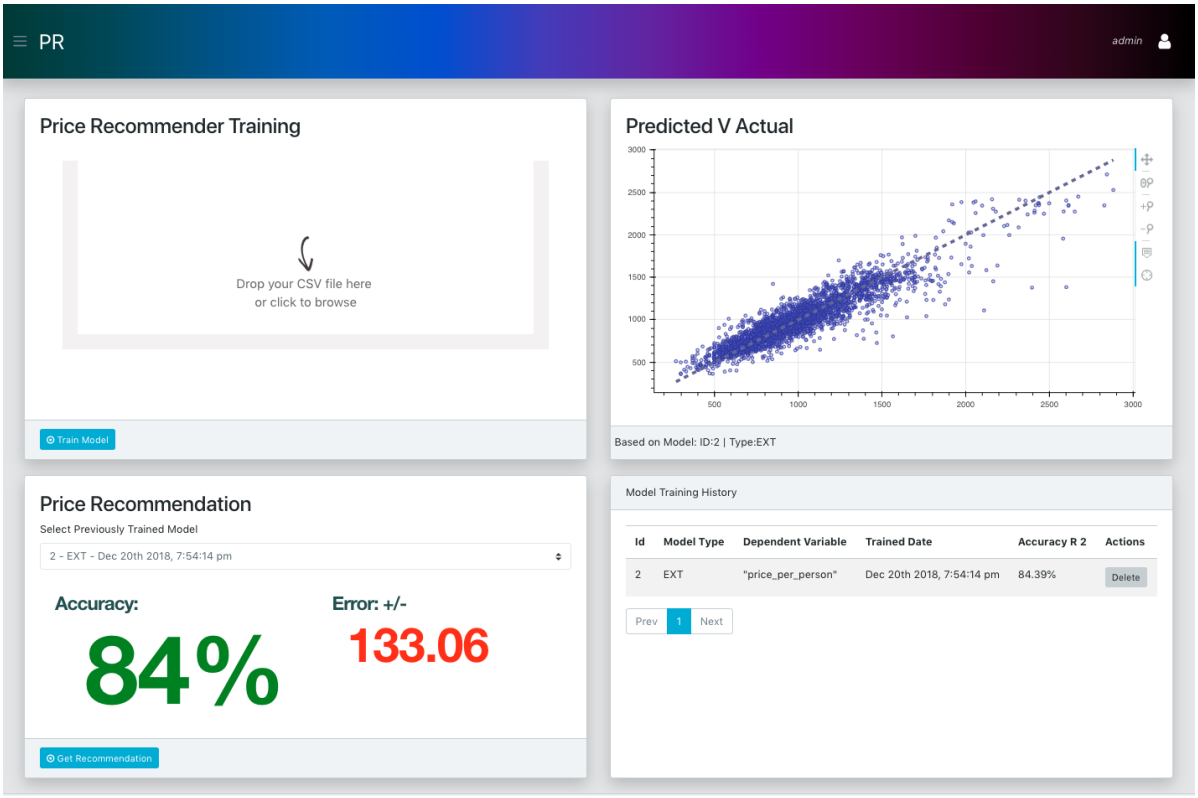


Fig. 6.1 Screen-shot of the Training Application Dashboard

6.1 Application Design

Figure 6.1 shows a screen-shot of the Training Application Dashboard. The final user interface design is very similar to that of early mock-ups. There is a single dashboard type interface with

panels for model training, price recommendation, visualisation (scatter plot) and training history. Other than the dashboard all other views are modal overlays with the exception of the login screen as shown in Figure 6.7. The following sections will explain communication between the client and server, the dashboard and each of the panels, how the client is implemented and how the server is implemented with diagrams and where appropriate source code examples.

All source code is publicly available on github.com so whenever a file is mentioned in the document, it will be linked directly to the source code file on github.com. Please also see Appendix A for full source code listing if reading a printed copy.

6.2 Client to Server Communication

The Training Application has an API helper service, that uses a popular javascript framework called Axios, for sending HTTP requests to the REST API. Axios provides a promise based HTTP client that ensures asynchronous communication between the client and server. The code for the API helper service is in the file APIService.js.

On the server side, the REST API uses the Django REST python framework for serving up RESTfull API end-points for the servers application logic. Figure 6.2 shows an overview of the application architecture where the REST API endpoints are shown as well as their associated classes, services and methods. It also shows the flow of how models are trained and serialized in a way that they can be recalled easily for prediction. This will be discussed in more detail later.

The API end-points are mounted at /api/v1/ in urls.py so that future versions of the API can be created without breaking systems that implement earlier versions. Communication is over HTTP with responses in JSON. Django Rest uses the concept of Views taken from the Model-View-Controller (MVC) architectural pattern. APIViews are mounted using a second urls.py to the classes in views.py. Each class has methods that are mapped to one or more of the standard HTTP request types such as GET, POST, PUT, UPDATE and DELETE. Each end-point then uses one or more services that have been developed to perform the desired functionality such as model training and prediction.

6.3 Model Training

Model training consists of three steps for the user. Step one is to upload a file in CSV format containing the data to train the model on. Step two is to select the appropriate fields for pre-processing. Step three is to select the desired model and the maximum run time to allow if

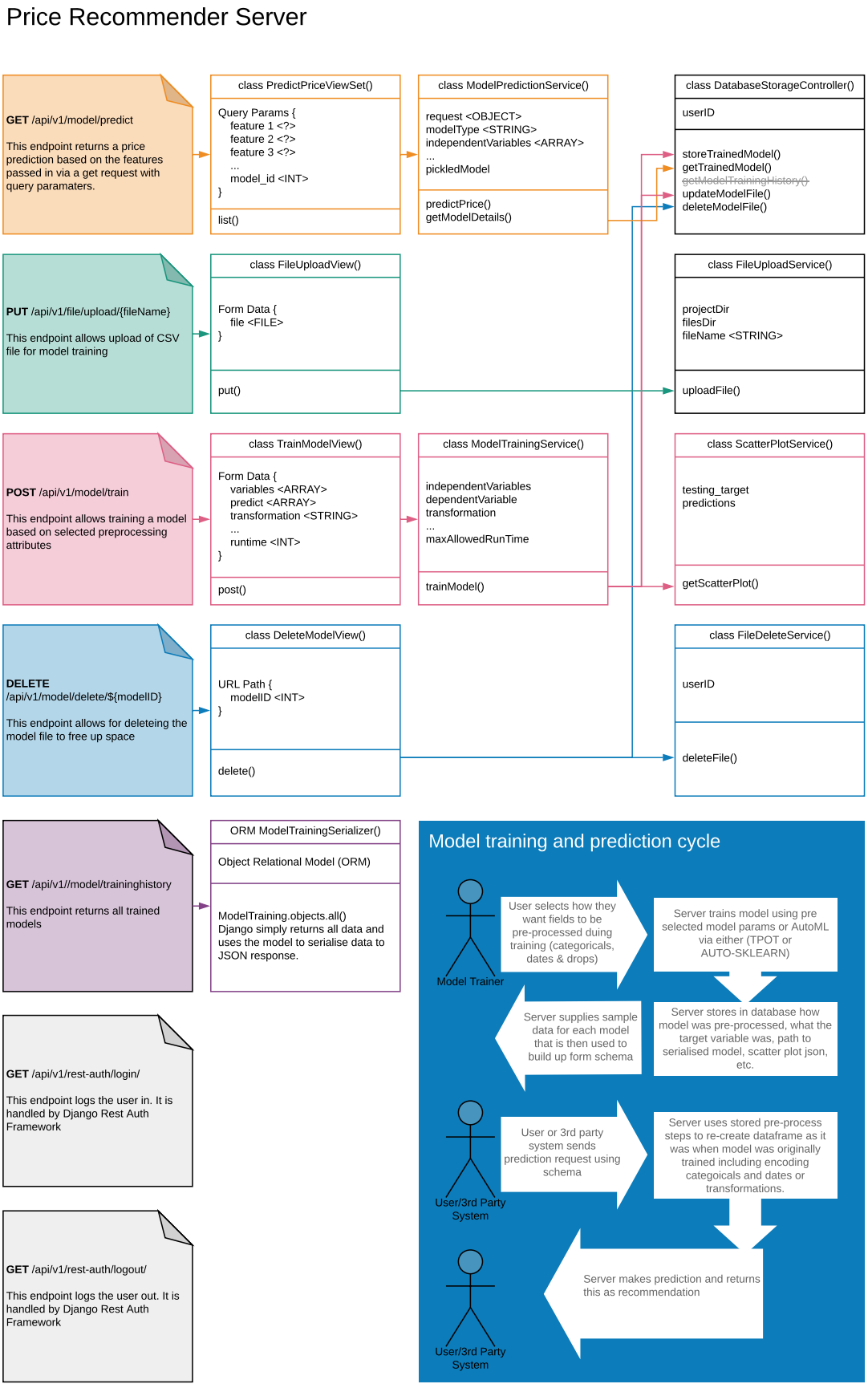


Fig. 6.2 Diagram of Price Recommender Server, showing end-points, classes and services

using an automated machine learning model type. The following sub-sections will explain each of these steps and how they are implemented on the client and server.

6.3.1 Step 1 - CSV File Upload

On the client side in TrainModel.vue a drag and drop file upload component was developed to allow the user to upload a CSV file of training data by simply dragging the file to the designated area. Alternatively the user can simply click the file upload area and browse for the file to upload.

The component uses a HTML form rendered by VueJS with two-way data bindings. The application listens for form-data changes via the ***filesChange()*** method and as soon as the file is dropped or selected it begins to process the data.

The CSV file is parsed to extract the field or column names. These are then used to dynamically build a form for pre-processing based on the data in the CSV file. At the end of this method it calls the ***save()*** method which sends the file to the server via the ***uploadFile()*** helper method in APIService.js.

On the server side the file upload request is marshaled via the **PUT** command to the ***FileUploadService()*** class in upload.py where the ***uploadFile()*** method, deletes the existing training data CSV file and replaces it with the new file uploaded, returning a 200 OK response or a 500 error response if an exception is raised.

6.3.2 Step 2 - Select Model Training Options

At this point all the user has done is dragged and dropped a file. Everything else has happened in the background almost instantly. The user then clicks the train button. Figure 6.3 shows a screen-shot of the training options that render as a modal overlay. From here, the user is asked to select how they want the training data to be encoded and what model type to use.

Currently the application is not able to automatically determine data types so the user must choose appropriately based on inspection of the data or meta data. The application also presumes the data has been cleaned and any feature engineering required completed.

Table 6.1 shows each of the data encoding options and their usage. The client renders a HTML form with multi-selects, drop-down selects and radial selects. Each selection is bound to the components data objects using two-way binding.

Table 6.2 shows the machine learning model type selection options and their usage. The first three options are static models with pre-tuned hyper-parameters. The last three are dynamic models that use automated machine learning to select the best model. As this can take considerable time there is an option to set the max runtime of between 1 and 24 hours. In

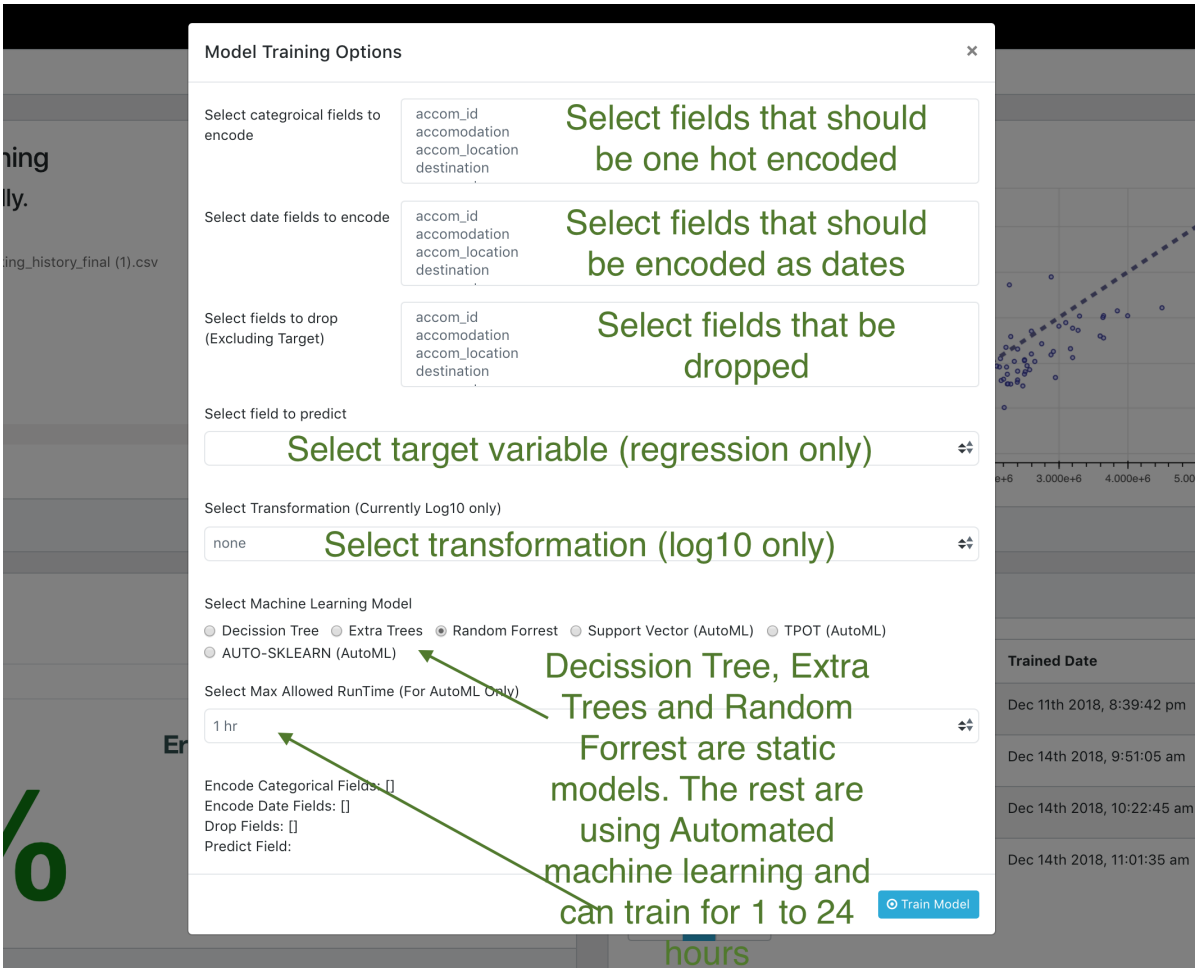


Fig. 6.3 Screen-shot of Price Recommender Training showing field pre-processing and model selection options

Table 6.1 Training Data Pre-Processing

Selection	Action	Usage
Categorical fields to encode	pd.get_dummies()	Categorical variables are converted into a form that ML algorithms can work with
Date fields to encode	pd.to_datetime().dt.week	Date variables converted into dates and the week of the year extracted
Fields to drop	pd.drop()	These fields are simply dropped in place
Field to predict	NA	Sets the target variable
Transformation	np.log10	Performs natural log transformation on target variable
NA	pd.fillna()	Replaces NULL/NaN values with zeros

Table 6.2 Training Model Selection

Model Type	Usage
Decision Tree	DecisionTreeRegressor from sklearn.tree
Extra Trees	Ensemble based on stacking XGBRegressor and an ExtraTreesRegressor from sklearn.ensemble and XGBoost respectively
Random Forrest	RandomForestRegressor from sklearn.ensemble. (Prototype Model)
Support Vector	AutoML using TPOT limited to sklearn.svm.SVR & sklearn.svm.LinearSVR
TPOT	AutoML using TPOTRegressor
AUTO-SKLEARN	AutoML using autosklearn.regression

reality, there is no hard limit so this range could be extended to days even weeks depending on the infrastructure the application is deployed to.

There is also an additional description field which is used to allow the user to give a meaningful description of the model being trained. Note: this was added after some of the screen-shots were taken.

6.3.3 Step 3 - Train the model

Once the user has made their selection and clicks the train button, the trainModel() method builds up a FormData object and sends it to the REST API via the ApiService.trainModel() helper method.

The REST API server, upon receiving the training request, marshals it, via the **POST** command to the train.py service. The trainModel() method performs all of the model training steps as requested by the client. This is quite a complex process so I have broken down each step.

1. Read in the CSV training data that was previously uploaded, into a new data frame.
2. Drop fields as requested.
3. Store one row of data as a JSON object that will later be used to construct a schema for the prediction form. Note the data is stored as it was prior to pre-processing/encoding.
4. Encode date fields as requested.
5. Encode dummies for categorical fields as requested. Note new dummy fields are created with a prefix separator to make them easily identifiable.

6. Put in zeros for any nulls (currently non optional).
7. Identify the dummy fields created using the prefix separator and store them. This is done so that the dummy fields can be re-created regardless of the data passed to the model during the prediction process.
8. Perform transformations on target variable (currently limited to natural log).
9. Set the variable y to be the target variable as requested.
10. Drop the target variable from the data frame.
11. Set the variable X to be a numpy array containing the data frame (without target variable).
12. Split the data into training and testing sets. Note the test data size is set to 20%. This was following experimentation with different test sizes.
13. At this point conditional logic determines which model to use based on the users request. Any max run time limits for AutoML are set.
14. Fit the model.
15. If using AutoML the application will gracefully break out after the max run time is exhausted, during which the best model so far is selected.
16. Predictions are made and compared to ground truth (y).
17. A JSON object is constructed with various accuracy and error metrics.
18. A scatter plot is generated using the python Bokeh library and the JSON needed to embed the scatter plot is stored.
19. The complete details of the trained model including any parameters needed later when making predictions are stored in the SQLite database. (Model Training History).
20. The machine learning model is then serialised to disk as a joblib file.
21. The path to the serialised model file is updated in the database.
22. And finally the accuracy JSON object is returned to the client.

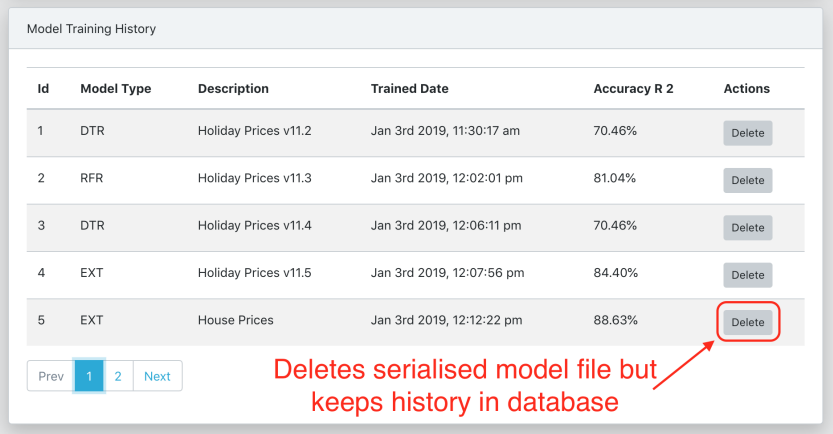


Fig. 6.4 Screen-shot of Price Recommender Training History

6.4 Model Training History

Figure 6.4 shows a screen-shot of the model training history panel. It's main purpose is to show all previously trained models and give some details on them. It also has a single action allowing the user to delete the model file that was serialised to disk. Deleting the file doesn't delete the history. For context, the Random Forrest Regressor Model is quite large when serialised to disk at approximately 700MB in size. So it's important to be able to clean up the files when no longer needed leaving the performance history for reference.

The database schema is generated as part of Django's object relational modeling (ORM) system automatically. There are simple CLI commands to migrate schemas to the database. See Django Quickstart Guide for details. The file models.py contains a single class ***ModelTraining()*** which represents the model training history data. It has a serializer class for marshaling and demarshaling data via the API.

When a model is trained the history of the training process is stored in the SQLite database with the help of the ***DatabaseStorageController()*** service in database.py. This service has methods for storing and retrieving the model training history. There are also two other methods for updating and deleting the reference to the serialised model file on disk. This maintains the record in the database, but deletes the reference to the persisted file. A separate method in delete.py looks after deleting the actual serialised model file from disk.

6.5 Price Recommendation

For demonstration and testing purposes the client has a panel that allows the user to select a previously trained model and generate a price recommendation based on the feature values

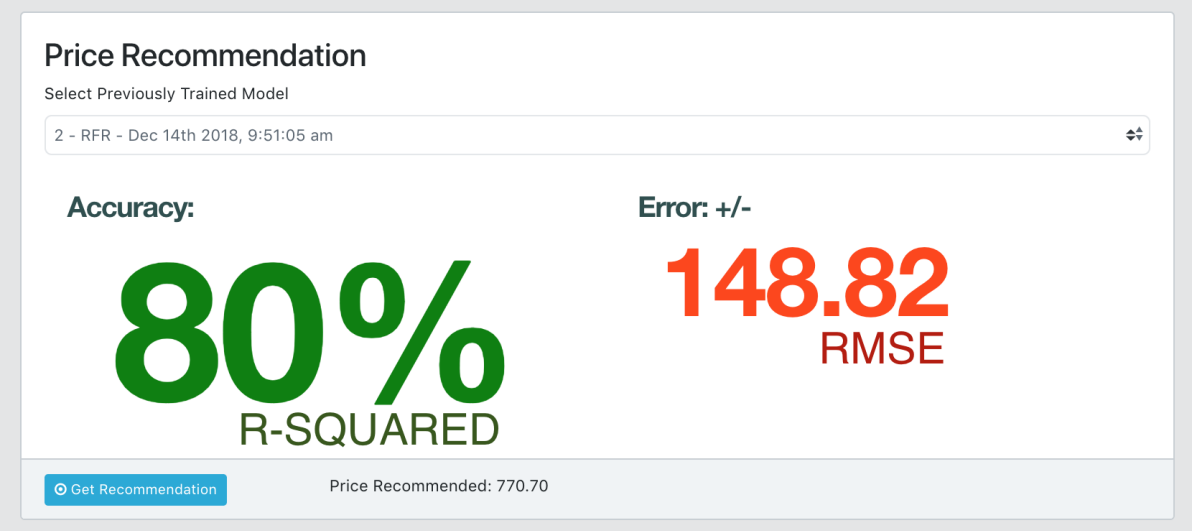


Fig. 6.5 Screen-shot of Recommend Price Panel

they choose to set. The best performing model is always selected by default. Figure 6.5 shows an annotated screen-shot of the panel with the r-squared and root mean squared error of the selected model displayed.

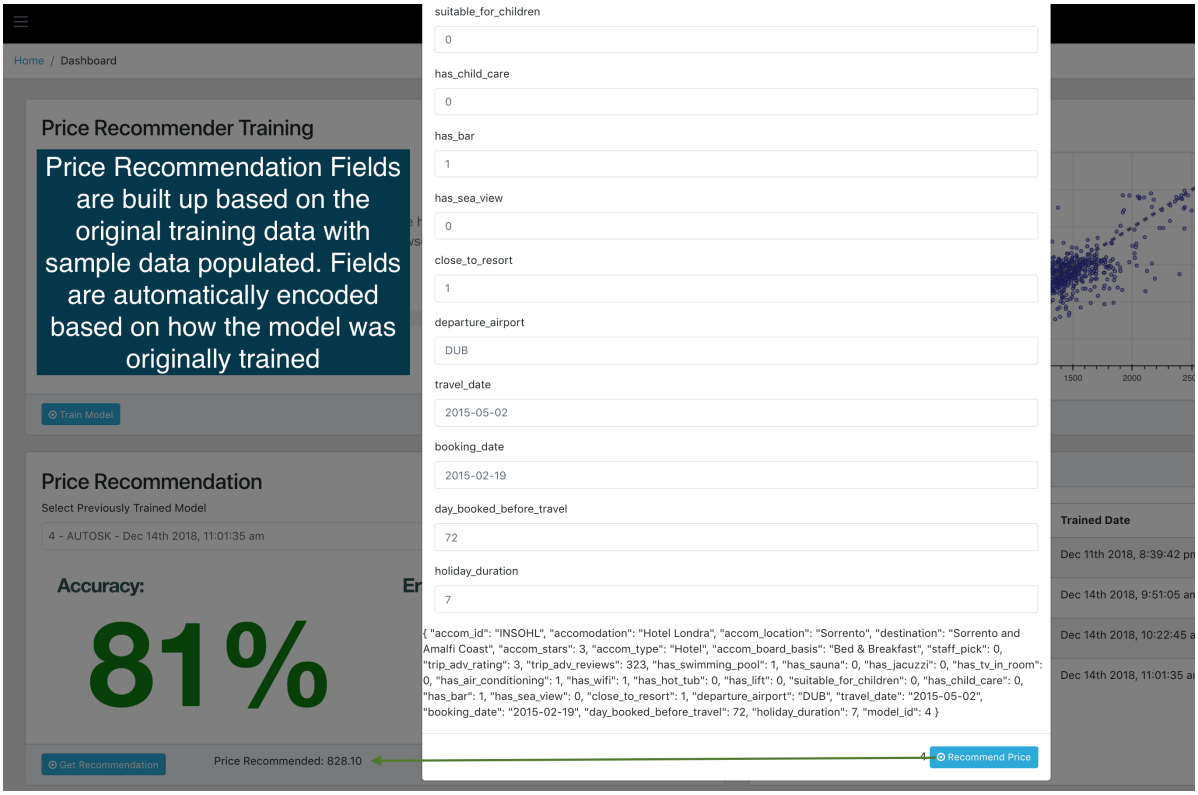


Fig. 6.6 Screen-shot of Recommend Price Panel Modal

Figure 6.6 shows an annotated screen-shot of the price recommendation modal overlay that renders after the user has selected a model and clicked the predict button. The modal overlay contains a form that is pre-populated with sample data having been dynamically generated based on the selected model. The user is free to change any of the variables which will affect the price predicted or recommended by the selected model.

6.6 Machine Learning Prediction Steps

The predictPrice() method performs all of the model prediction steps based on the features passed by the user or third party system.

As part of the machine learning model training process, everything needed to recreate the correct steps for prediction was stored as well as the path to the serialized model file on disk. On instantiation of the ModelPredictionService() class the model history is retrieved from the database based on the model type passed in the GET request.

Again like the training steps this is quite complex so I have broken down each step in the prediction process.

1. Load the Machine Learning model file back into volatile memory from disk.
2. Create a new data frame with columns based on the independent variables the model was originally trained on.
3. For each of the independent variables loop though the GET request params and add the values sent to the data frame.
4. Encode date fields based on how it was done during model training.
5. Encode dummies for categorical variables based on how it was done during model training.
6. Remove any additional dummy fields that were not cerated during model training. This could happen if the training set was quite small and did not contain all possible categorical values.
7. Add any additional dummy fields that were created during model training but are now missing. This will happen because our training data would have had sight of all possible categorical variables and encoded them as dummies. However our prediction data would only encode one of them. The machine learning model will throw a software exception if it is asked to predict on data containing a different number of variables to the ones it was originally trained on.

8. Set X to be a numpy array containing the data frame we want to predict based on. (this is not needed but improves performance).
9. Predict for X.
10. Reverse out any transformations we originally did on the target variable (for example log10).
11. Return the predicted/recommended price.

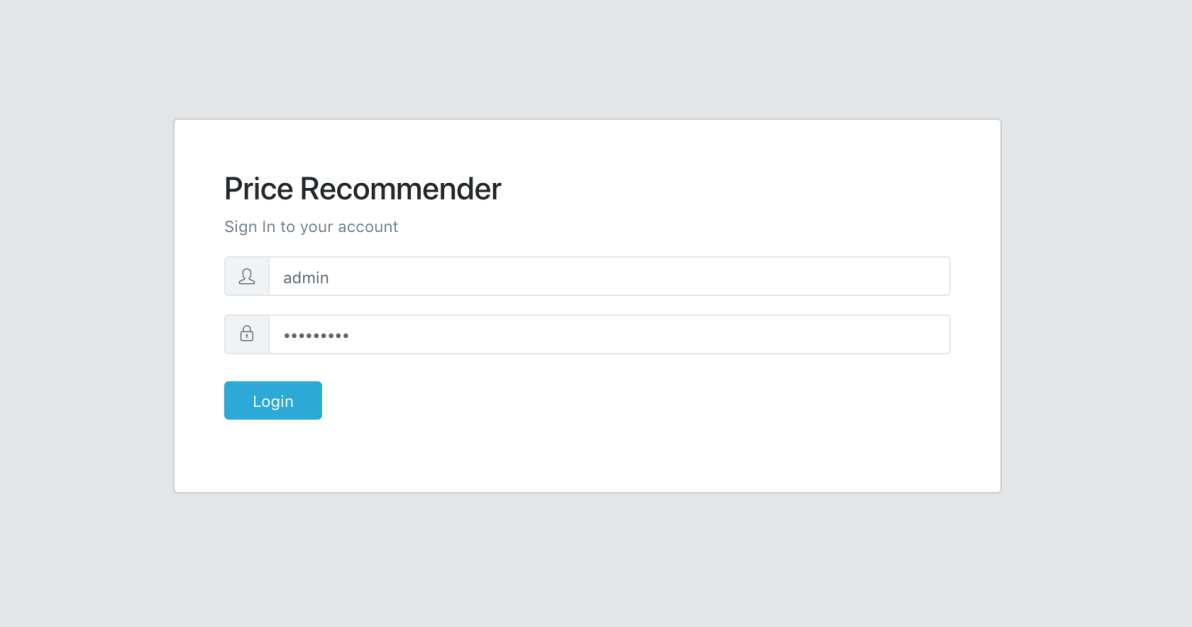


Fig. 6.7 Screen-shot of Price Recommender Login

6.7 Authentication & Login

There is a requirement that the client training application requires a user to login in order to train the models or make predictions. Authentication is handled by Django Rest Auth framework with very little implementation required in development on the server side. Figure 6.7 shows a screen-shot of the login screen that is presented to the user. A token is stored in the users browser cookies for 1 day after which they will be required to login again. The user can logout by clicking the user icon in the navigation bar.

Chapter 7

Deployment to Production

7.1 Overview

Deploying any application to production can be challenging. Often what works in development can fall over in production usually due to differences in the environment the application is deployed to. The following sections will explain the production deployment challenges and the use of docker containers and docker compose in orchestrating the build process and leveraging the concept of "infrastructure as code".

7.2 Production Challenges

Machine learning applications introduce their own challenges when it comes to production deployment. For example in order to persist a trained model for future use we need to serialise it to disk form volatile memory and then de-serialise it when we want to use it for predictions. For this to work the environment needs to be consistent throughout this process. Inconsistencies in the underlying system can break previously trained models.

As discussed earlier for machine learning and AutoML the libraries required can be difficult to install. This makes even development a challenge especially when using AutoML libraries that are to coin a phrase, "bleeding edge", in terms of their development at this time.

For this and other reasons, I have created docker and docker-compose files to orchestrate the build of docker images and the deployment of the application in containers. This allows the application to run in a self contained environment free from the concerns of the host machine, with everything that is required to run the application already installed.

7.3 Building Docker Images

Appendix B contains the complete steps for building the docker images, pushing to docker hub and using docker compose to orchestrate the client and server builds.

7.3.1 Base Docker Image

A base docker image is built using the base image dockerfile. This creates a base Ubuntu Linux image with python and node installed that can be extended for building other images with different layers. An entrypoint shell script allows commands to be passed to the image at runtime.

7.3.2 Server Docker Image

A server image dockerfile is built using the base image during which the application source code is copied to the final image. Here caching is used so that subsequent builds of the image only process the steps where changes have been detected. An entrypoint shell script installs all application dependencies and starts the server.

7.3.3 Client Docker Image

A client image dockerfile uses a multistage build of a node docker image and nginx docker image. The source code is copied to the image and dependencies are installed. Finally a command to start the nginx server exposing it on port 80 completes the image.

7.3.4 Container Orchestration

A docker-compose file creates two build contexts for client and server. Upon execution using the "build" command the client and server images are built or pulled from docker hub if available, exposing the appropriate ports and deploying a container of the application for external access.

Chapter 8

Results & Evaluation

8.1 Evaluation of System Design

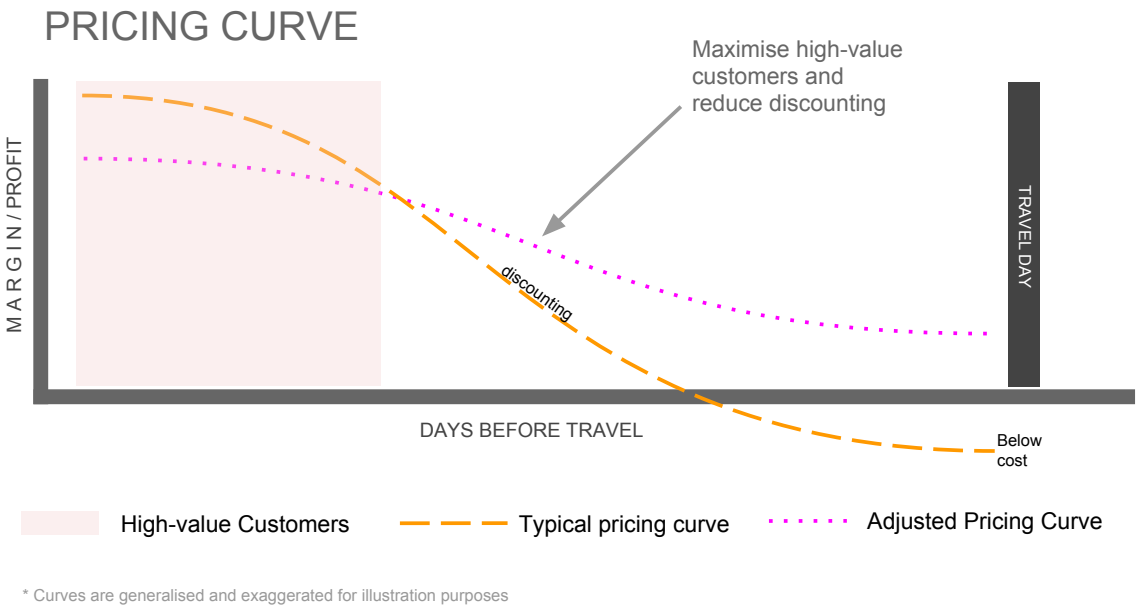


Fig. 8.1 Generalised chart of adjusted pricing curves using machine learning

The design of the recommender system allows it to recommend a price that the customer would be willing to pay based on the fact that they historically were willing to pay this price for a similar product. It was not intended that this prima facie price would be used directly in setting the selling price of a product without adjusting for bias. In order to create an unbiased recommended price, rules would need to be applied to the predicted price in order to deliver a

more desirable outcome. For example the price recommender might recommend a price that is below cost price. After all, historically some products might have been sold successfully below cost. Selling below cost is generally not a desirable outcome. Therefore the price recommender would ideally need to have rules to allow it to generalise more.

Alternatively the external implementation of the price recommender might take a more active roll. In integrating with the Price Recommender API, it might make requests for multiple predictions along a timeline in order to forecast the selling curve. It could then suggest corrective measures to normalise a severe selling curve. Figure 8.1 shows an example of how the selling curve might be adjusted by leveraging machine learning and applying additional rules to "normalise" a severe pricing curve".

8.2 Model Selection Results

The best fitting models, following extensive experimentation using automated hyper parameter tuning and automated machine learning, were Random Forrest Regressor and an ensemble model in which an XGBoost Regressor and an Extra Trees Regressor was stacked, with the best model being the latter both in terms of accuracy and stability.

The resulting models sometimes produced different price recommendations for the same input data. Similarly models of the same type with identical parameters, following successive training sessions, might also produce price recommendations with small differences. In other words, models trained successively are not the same and can't be relied upon to produce the exact same result. In the case of Random Forrest Regression this is exacerbated by the bagging algorithm.

So in practice a real world application of the system might choose to use multiple models and rules for making "best guess" price recommendations. For example the integration in an e-commerce system might choose to take the median of predictions from multiple trained models of varying types as a way to further reduce bias.

It was clear during the experimental phase of model training that further model tuning was only going to lead to marginal improvements in accuracy. So it was decided to let the software application itself cater for training and improving the models even further. The user could try adding additional features to the training data or select AutoML as an option and leave it training for days or even weeks.

8.3 Other Applications

Although the Price Recommender is only programmed for a specific domain and data set, it can be shown to that it is capable of training models on completely different datasets and domains. One example that worked well was predicting house prices. A dataset was sourced from kaggle.com of House Sales in King County, USA. The CSV data was uploaded and the extra trees model type selected. Following training this produced an R-Squared accuracy score of 89% in predicting the selling price target. It might not have won the kaggle competition but it demonstrates how the developed application could be used outside of predicting holiday selling prices without extra development effort being required.

8.4 System Security

Authentication is the mechanism in which an incoming request with a set of identifying credentials or token is given permission to make a request to the API. Currently this has not been implemented in the API as it is anticipated that the API would be restricted by IP address of the calling application. However it would be a consideration for commercialising the application.

8.5 Production Deployment

The application was deployed to a Debian Linux server in Google Cloud Compute using the steps outlined in the developer guide in Appendix B. There were some issues related to file permissions and having to "sudo" some of the commands but otherwise production deployment was very straight forward with minimal steps.

The production build of the application doesn't take into account that the IP of the REST API is not the same in development as it is in production. Also there are two training service python files. One is for development and one is for production. The development train.py is replaced by train.production.py during the production build. While not ideal, it allows a developer to run the application locally, outside of a Linux environment, with AutoML commented out.

A better approach would be to have configuration files for development and production to separate out these concerns, reduce duplicate code files and further optimise the production build and deployment process.

Chapter 9

Future Work & Conclusions

9.1 Future Work

The project set out to build a Price Recommender API but in doing so also built an automated machine learning application. It currently only deals with regression but could also be adapted to work with classification problems. Just focusing on the current implementation some future work includes

- Additional preset model types could be added from sklearn and other libraries
- More options for feature pre-processing
- Options for data imputation, such as filling missing values instead of simply dropping (average, median, etc)
- Automated data type detection on the training application to give the user a better idea when selecting pre-processing options
- More visualisation options such as time series predictions
- Automated re-training or reinforcement learning
- Introduction of deep neural networks possibly
- Improved security and recording training activity by user
- Add support for training multiple models simultaneously. (This is currently limited by the file upload service which always replaces the training data file when a new one is uploaded)
- Add compression to serialised joblib files

9.1.1 Project Deviations

In my initial project and Bachelor thesis proposal I had hoped to be able to implement the project in a working environment and get feedback on its practical application. However this was not practical. Unfortunately it was not possible to engage the resources of a business in order to do the integration at the same time as completing the development of the project application itself. I also wanted to keep the project separate from my current employment in order to protect my ownership of the intellectual property of the developed application.

Additionally my project proposal and interim report did not set out to implement automated machine learning in the application. However during research and development phases, it became clear that by leveraging automated machine learning in the training application, it delivered more in relation to the goal of allowing non-experts to train machine learning models.

9.2 Conclusions

This Bachelor thesis researched the application of supervised machine learning in selling price recommendation for revenue management. Subsequently it documented the successful build of a Price Recommendation application consisting of a REST API and Training Application that could be leveraged by a product and revenue department when contracting and setting the selling price of a product respectively. And finally it demonstrated the production deployment solutions for deploying this type of complex application with consideration for its intricate dependencies.

It has been shown that machine learning both manual and automated can be embedded in a software application in such a way that a user, with no technical or data science knowledge, can train machine learning models for use in price recommendation with up to 89% accuracy. It has also been shown that the application can be trained on data for different domains and datasets than the target domain, without additional development.

It has been shown that a machine learning pipeline consisting of gradient boosted trees (XGB Regressor) and extremely randomised trees (Extra Trees Regressor), as selected using automated machine learning, can be more accurate and stable than regression tree or random forest regression tree alone as proposed by (Tziridis *et al.*, 2017)

Additionally this Bachelor thesis demonstrates how automated machine learning can do the heavy lifting in model selection and is capable of producing better machine learning pipelines than could otherwise be selected manually, using traditional methods, by non experts.

In terms of the real word application of machine learning in Price Recommendation for revenue management, it is yet to be shown how using a predicted price for recommending or setting future selling prices, can be applied without consideration for bias introduced by

historical discounting and pricing behaviors. Further work is required in order to test the practical applications of this technology in a real business environment in situations where a more generalised recommendation algorithm is required to normalise a severe pricing curve and aide a revenue manager in maximising profit margins.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D.G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y. and Zheng, X. (2016) TensorFlow: A system for large-scale machine learning p. 21

Abeel, T. and Saeys, Y. (2009) Java-ML: A Machine Learning Library p. 4

Balaji, A. and Allen, A. (2018) Benchmarking Automatic Machine Learning Frameworks *arXiv:1808.06492 [cs, stat]* arXiv: 1808.06492

Barnard, L. and Mertik, M. (2015) Usability of Visualization Libraries for Web Browsers for Use in Scientific Analysis *International Journal of Computer Applications* **121**(1), pp. 1–5

Bitran, G. and Caldentey, R. (2003) An Overview of Pricing Models for Revenue Management *Manufacturing & Service Operations Management* **5**(3), pp. 203–229

Bottou, L. (2010) Large-Scale Machine Learning with Stochastic Gradient Descent pp. 177–186 Physica-Verlag HD

Breiman, L. (2001) Random Forests *Machine Learning* **45**(1), pp. 5–32

Chen, M. and Chen, Z. (2015) Recent Developments in Dynamic Pricing Research: Multiple Products, Competition, and Limited Demand Information. *Production & Operations Management* **24**(5), pp. 704–704–731

Chen, T. and Guestrin, C. (2016) XGBoost: A Scalable Tree Boosting System pp. 785–794 ACM Press, San Francisco, California, USA

Dietterich, T.G. (2000) Ensemble Methods in Machine Learning pp. 1–15 Springer-Verlag

Django, R. (2018) Home - Django REST framework

Efron, B., Hastie, T., Johnstone, I. and Tibshirani, R. (2004) LEAST ANGLE REGRESSION p. 93

Elmaghraby, W. and Keskinocak, P. (2003) Dynamic Pricing in the Presence of Inventory Considerations: Research Overview, Current Practices, and Future Directions *Management Science* **49**(10), pp. 1287–1309

Feurer, M., Klein, A., Eggensperger, K., Springenberg, J., Blum, M. and Hutter, F. (2015) Efficient and Robust Automated Machine Learning pp. 2962–2970 Curran Associates, Inc.

Freund, Y. and Schapire, R.E. (1999) A Short Introduction to Boosting p. 14

Friedman, J.H. (2001) Greedy Function Approximation: A Gradient Boosting Machine *The Annals of Statistics* **29**(5), pp. 1189–1232

Gallego, G. and Ryzin, G.V. (1994) Optimal Dynamic Pricing of Inventories with Stochastic Demand over Finite Horizons *Management Science* **40**(8), pp. 999–1020

Geurts, P., Ernst, D. and Wehenkel, L. (2006) Extremely randomized trees *Machine Learning* **63**(1), pp. 3–42

Gray, R. (2018) Application of Machine Learning in Package Holiday Selling Price Prediction

Groves, W. and Gini, M. (2011) A regression model for predicting optimal purchase timing for airline tickets. Technical report

Hastie, T. and Tibshirani, R. (1996) Discriminant Adaptive Nearest Neighbor Classification and Regression pp. 409–415 MIT Press

Ho, C.H. and Lin, C.J. (2012) Large-scale Linear Support Vector Regression p. 27

Hoerl, A.E. and Kennard, R.W. (1970) Ridge Regression: Biased Estimation for Nonorthogonal Problems *Technometrics* **12**(1), pp. 55–67

Li, S. (2018) A Machine Learning Approach — Building a Hotel Recommendation Engine

Lieberman, W. (2011) Revenue Management in the Travel Industry John Wiley & Sons, Inc., Hoboken, NJ, USA

MacKay, D.J.C. (1994) MacKay, Bayesian nonlinear modeling for the prediction competition p. 14

Masse, M. (2011) *REST API Design Rulebook: Designing Consistent RESTful Web Service Interfaces* "O'Reilly Media, Inc." google-Books-ID: eABpzyTcJNIC

Olson, R.S., Bartley, N., Urbanowicz, R.J. and Moore, J.H. (2016) Evaluation of a Tree-based Pipeline Optimization Tool for Automating Data Science GECCO '16 pp. 485–492 ACM, New York, NY, USA

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A. and Cournapeau, D. (2011) Scikit-learn: Machine Learning in Python *MACHINE LEARNING IN PYTHON* p. 6

Prechelt, L. (2000) An empirical comparison of seven programming languages *Computer* **33**(10), pp. 23–29

Raff, E. (2017) JSAT: Java Statistical Analysis Tool, a Library for Machine Learning p. 5

sqlite.org (2019) SQLite Home Page

Talluri, K. and Ryzin, G.v. (2004) *The Theory and Practice of Revenue Management* International Series in Operations Research & Management Science Springer US

towardsdatascience, V. (2017) What is the best programming language for Machine Learning?

Tziridis, K., Kalampokas, T., Papakostas, G.A. and Diamantaras, K.I. (2017) Airfare prices prediction using machine learning techniques pp. 1036–1039

Witten, I.H., Frank, E., Hall, M.A. and Pal, C.J. (2016) *Data Mining: Practical Machine Learning Tools and Techniques* Morgan Kaufmann google-Books-ID: 1SylCgAAQBAJ

www.docker.com (2018) What is a Container

Z, M.O. and Lior, R. (2014) *Data Mining With Decision Trees: Theory And Applications (2nd Edition)* World Scientific google-Books-ID: OVYCCwAAQBAJ

Zhao, Q., Zhang, Y., Friedman, D. and Tan, F. (2015) E-commerce Recommendation with Personalized Promotion RecSys '15 pp. 219–226 ACM, New York, NY, USA

Zou, H. and Hastie, T. (2005) Regularization and variable selection via the elastic net *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* **67**(2), pp. 301–320

Appendix A

Code Listing

This page is intentionally left blank so code listing can take up the following entire page

Directory/File	Description
├── **client-app**	root client directory
│ ├── babel.config.js	Configuration file for Babel which is used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript
│ ├── package-lock.json	Used by npm for mapping changes to the node_modules tree or package.json
│ ├── package.json	Used to give information to npm that allows it to identify the project as well as handle the project's dependencies
│ ├── **public**	public directory for static resources
│ │ ├── favicon.ico	Favicon icon for web browsers
│ │ └── index.html	Entry point for client. VueJS injects into using it's virtual DOM
│ ├── **src**	source directory for javascript code
│ │ ├── APIService.js	API helper service. Methods implement Axios promise based HTTP client for sending requests to server
│ │ ├── App.vue	Top most component of the parent-child component tree where the router component is injected
│ │ ├── **assets**	Contains some templating files for styling
│ │ ├── **containers**	container directory
│ │ │ └── DefaultContainer.vue	Container component with header and footer components. The main router view mounts the login or dashboard vue components
│ │ ├── main.js	Instantiates VueJS and any additional components, plugins or resources.
│ │ ├── **router**	router directory
│ │ │ └── index.js	VueJS router for mounting pages based on URL
│ │ ├── **shared**	Contains some templating files for styling
│ │ └── **views**	views directory where all vue components live
│ │ ├── Dashboard.vue	This is the main dashboard vue component
│ │ ├── **custom**	directory for custom child vue components
│ │ │ ├── FormGenerator.vue	Generates forms using the NumberInput, SelectList and TextInput vue components based on a schema
│ │ │ ├── NumberInput.vue	Form input component for numbers
│ │ │ ├── PredictModel.vue	Price Recommender Prediction Component (Main client-side price recommendation logic is in here)
│ │ │ ├── ScatterPlot.vue	Component for displaying/embedding Bokeh scatter plot based on JSON data
│ │ │ ├── SelectList.vue	Form select dropdown component
│ │ │ ├── TextInput.vue	Form input component for text
│ │ │ ├── TimeSeries.vue	Not implemented yet (in future will be used to show temporal price predictions)
│ │ │ ├── TrainModel.vue	Price Recommender Training Component (Main client-side model training logic is in here)
│ │ │ └── UserPanel.vue	Component for showing user that is logged in with dropdown for logout
│ │ └── **pages**	pages directory
│ │ ├── ForgotPassword.vue	Not implemented yet (this can be done from Django back-end admin)
│ │ └── Login.vue	Component for login to training client application. If the user does not have a login token they are shown this page
│ └── vue.config.js	VueJS config for linting compile time errors
├── **docker**	root docker directory
│ ├── **base**	base image directory
│ │ └── Dockerfile	Base image docker file. This builds a ubuntu image with some base requirements like python and node
│ ├── **client**	client image directory
│ │ └── Dockerfile	Client image docker file. This builds an image with node and nginx web server
│ └── **server**	server image directory
│ └── **production**	production image directory
│ ├── Dockerfile	Production server image. This uses the base image and builds upon it to create server api image.
│ └── docker-compose.yml	This orchestrates the build of the server and client images and runs a container with both images in production.
├── **scripts**	docker scripts directory
│ ├── entrypoint.sh	entrypoint for base docker image
│ └── production_entrypoint.sh	entrypoint for production server docker image. This installs all python dependencies including AutoML libraries
├── **rest-api**	root server directory
│ ├── db.sqlite3	sqlite database file for storing Django tables and ModelTraining table
│ ├── **files**	files directory for serialised models and training data
│ │ ├── EXT1.joblib	serialised extra trees regressor model
│ │ └── training_data.csv	training data uploaded. (currently this overwrites)
│ ├── **main**	directory part of Django Rest
│ │ ├── settings.py	Django Rest settings file
│ │ ├── urls.py	Django Rest main urls mount points file
│ │ └── wsgi.py	Part of Django Rest core
│ ├── manage.py	Part of Django Rest core, used for database migrations, starting the server, creating superuser, etc
│ ├── **recommender**	root directory for servers application logic
│ │ ├── admin.py	Part of Django Rest core, registers site/api
│ │ ├── apps.py	Part of Django Rest core, registers site/api
│ │ ├── **migrations**	Sqlite database migration scripts
│ │ │ └── 0001_initial.py	Migration script for ModelTraining database table
│ │ ├── models.py	ModelTraining model, used for Django's ORM
│ │ ├── serializers.py	ModelTraining Serializer, used for Django's ORM
│ │ ├── **services**	services directory, all of the machine learning code is in here
│ │ │ ├── database.py	Helper service for storing and retrieving data form the database
│ │ │ ├── delete.py	Helper service for deleting serialised model files
│ │ │ ├── predict.py	This is where all the machine learning prediction / recommendation logic is
│ │ │ ├── scatterplot.py	Helper service for generating scatter plot JSON using Bokeh
│ │ │ ├── tmp	temp directory used by AUTO-SKLEARN during training and model selection
│ │ │ ├── train.production.py	This is where all the machine learning training and AutoML logic is
│ │ │ ├── train.py	This is where all the machine learning training and AutoML logic is (AutoML commented out for local development)
│ │ │ └── upload.py	Helper servicer for training data file upload and storage
│ │ ├── urls.py	Mounts all the api endpoint urls to Django views
│ │ └── views.py	This is where all the API end-points are mapped to server logic
│ ├── requirements.production.txt	This file contains all production dependencies (AutoML)
│ └── requirements.txt	This file contains all production dependencies (Excluding AutoML)
└── readme.MD	Explains to developers how to build the application both locally for development and using docker in production

Appendix B

Developer Guide

This page is intentionally left blank so the developer guide can be embedded in the following pages

Server (Django Rest-API)

System Requirements

- Python 3.6
- virtualenv python module (https://virtualenv.pypa.io/en/latest/installation/)
- Linux or Unix based development system (Linux Recommended, MacOS support is limited)
- Docker 18.00.0+
- Docker Compose

Important Note!

In /recommender/services/train.py AutoML frameworks TPOT and AUTO-SKLEARN might not run properly in non-linux environments. So they can be commented out in the source code for running locally in development outside of a docker container. AUTO-SKLEARN is commented out by default as it will not install on my development environment of MacOS. You might also need to comment out TPOT and not use these models locally. The production docker image allows you to work around these limitations.

Create a Python Virtual Environment

From the project root directory

```
# virtualenv venv
```

Activate Python Virtual Environment

```
# source venv/bin/activate
```

Install dependencies

```
# pip install -r rest-api/requirements.txt
```

Make Migrations

From the rest-api directory

```
# cd rest-api
# python manage.py makemigrations recommender
# python manage.py migrate
```

Create Rest-API Superuser

From the rest-api directory

```
# cd rest-api
# python manage.py createsuperuser
# user: admin, pass: admin123$
```

Run the Rest-API Server

```
# python manage.py runserver
```

- From a web browser navigate to http://localhost:8000/admin/ and verify the Rest-API Server is running and that you can log in.
- From a web browser navigate to http://localhost:8000/api/v1/rest-auth/login/ and verify that the Rest-API Server login page endpoint shows.

If you get an error about the hypervolume module, reinstall deep with the following command

```
# pip install deap==1.0.2.post2
```

Client (VueJS Web Application)

Requirements

- NodeJS
- NPM

Install dependencies with npm

From the client-app directory install dependencies

```
# cd client-app
# npm install
```

Run Client Application

```
# npm run serve
```

From a web browser navigate to http://localhost:8080 and login with superuser credentials. Alternatively navigate to http://localhost:8000/admin/ and setup new credentials.

Docker Builds For Server

Requirements

- Linux (Debian Recommended)
- Docker v18.00.0+
- Docker Compose v3+
- Python v3.6

Build the base docker image

From the root directory:

```
--Build the image
# docker build -t grayrobert/price-recommender-base -f
docker/base/Dockerfile .

--Push to docker hub
# docker push grayrobert/price-recommender-base
```

After running docker images you should have a new base ubuntu image with python installed

Build the production docker image from the base docker image (REST-API Server)

Note: during docker build the training service python file train.production.py replaces the default train.py and enables TPOT and AUTO-SKLEARN models mentioned earlier.

From the root directory:

```
# docker build -t grayrobert/price-recommender-server -f
docker/server/production/Dockerfile .

--You can run the image and jump to a bash prompt using the following
command
# docker run -it --entrypoint /bin/bash grayrobert/price-recommender-
server

--Push to docker hub (please don't do this without permission)
# docker push grayrobert/price-recommender-server
```

Save the server docker image for manual deployment to production

From the root directory:

```
# docker save -o target/price-recommender-server.tar grayrobert/price-
recommender-server:latest
```

Docker Builds For Client

Requirements

- Linux/Unix
- Docker
- Node/NPM

Build the client docker image

From the root directory:

```
--Build the image
# docker build -t grayrobert/price-recommender-client -f
docker/client/Dockerfile .

--Push to docker hub (please don't do this without permission)
# docker push grayrobert/price-recommender-client
```

Run the client docker image

```
# docker run -it -p 8080:8080 --rm grayrobert/price-recommender-client
```

Save the client docker image for manual deployment to production

From the root directory:

```
# docker save -o target/price-recommender-client.tar grayrobert/price-
recommender-client:latest
```

To Run the application in production

Docker compose will orchestrate the build of images and starting of both the server and client, mapping ports and whatever other requirements are needed. Curently the most streamlined way to put the application into production is to install docker, docker compose and git. Clone the source code fom git repository and then use docker-compose to do all the heavy lifting.

On a server with Debian Linux, GIT, Docker and Docker Compose installed and the source code cloned from git repository simply do the following two step deployment. From the root directory:

```
1. Pull the latest changes from master branch
# git pull origin master
2. Run the following command
# docker-compose --file docker/server/production/docker-compose.yml up --
build
```

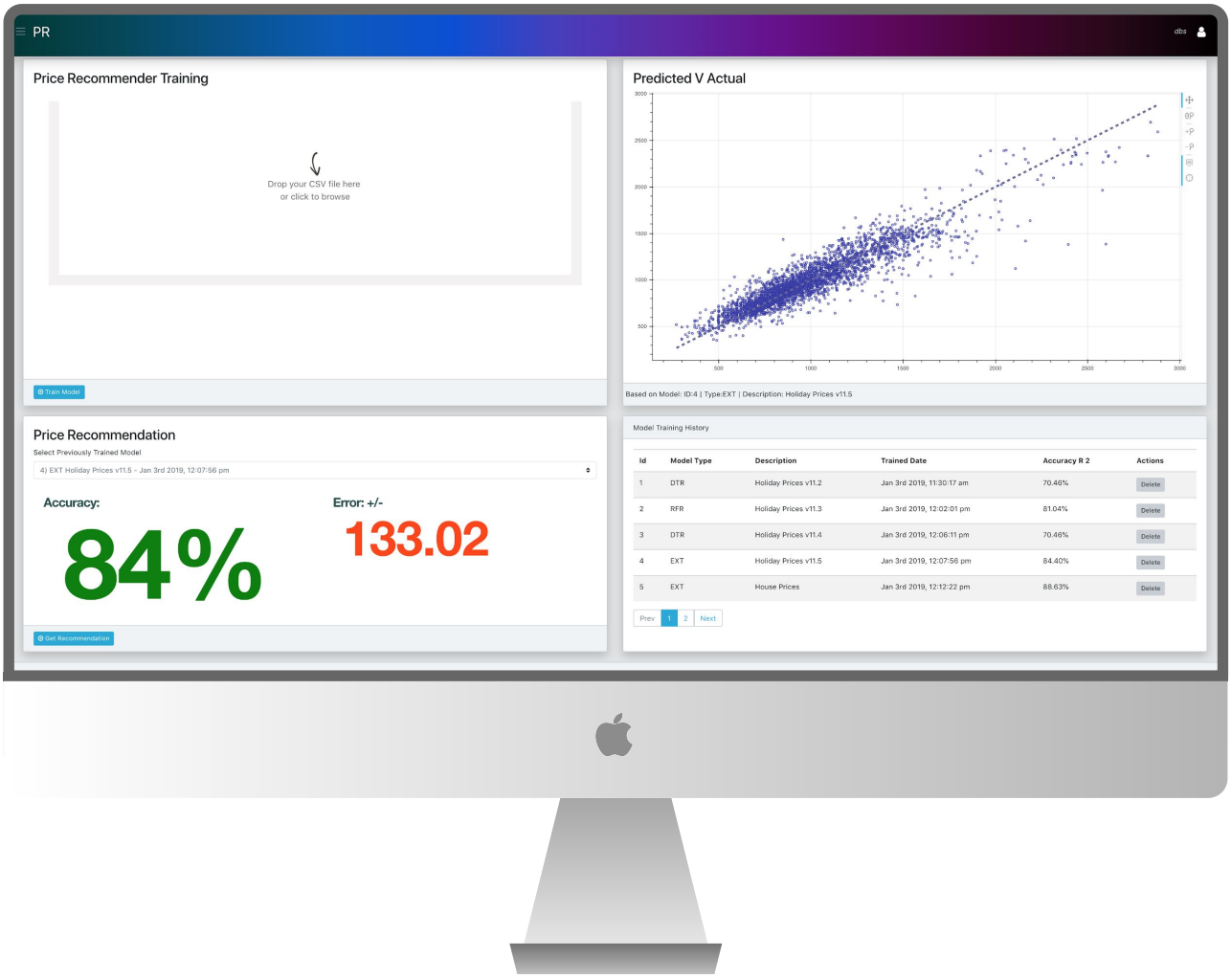
Then open a browser and navigate to http://ip of server or localhost:80

Appendix C

System Manual

This page is intentionally left blank so the system manual can be embedded in the following pages

PRICE RECOMMENDER



Prepared by:
Robert Gray

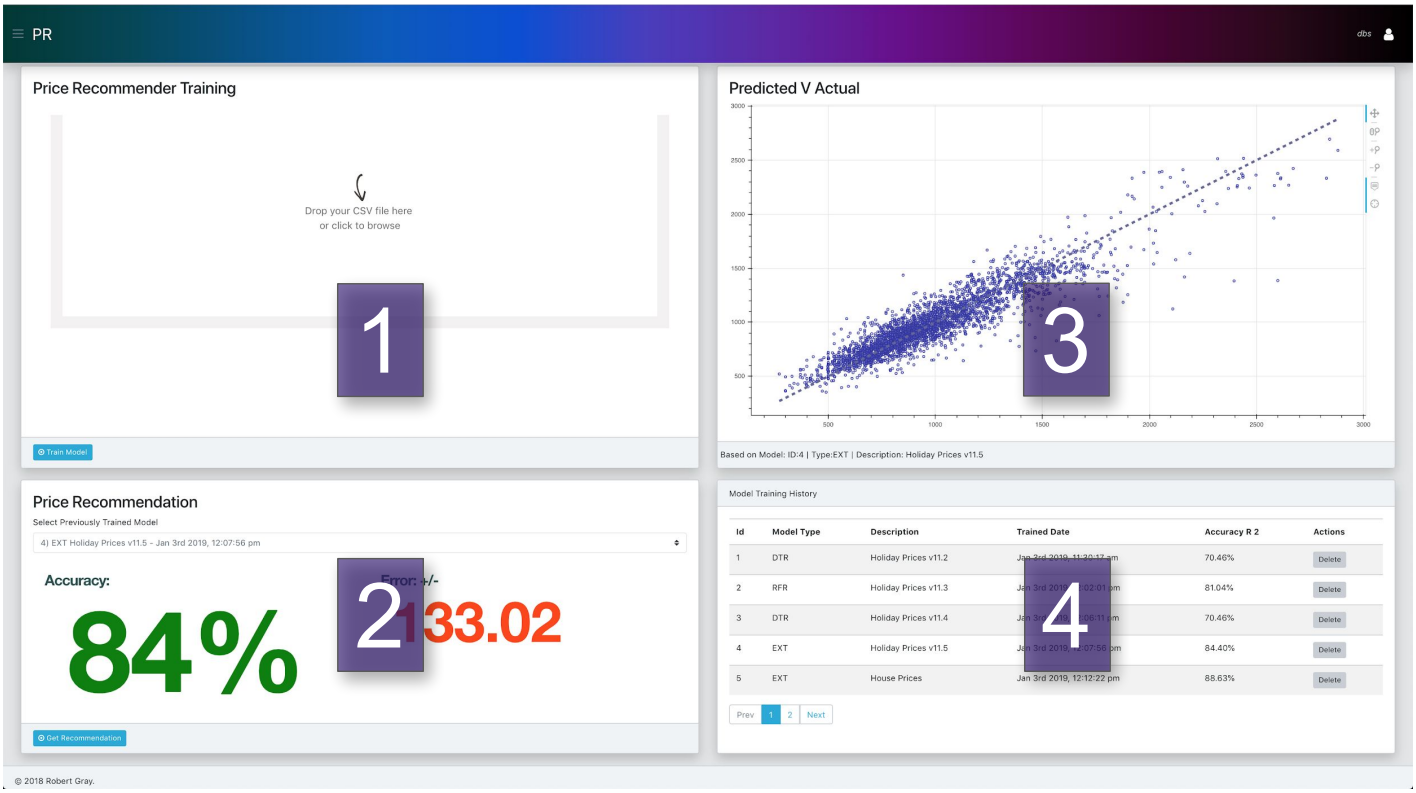
Date: 5 Jan 2019

Version: 1.01

SYSTEM MANUAL

CONTENTS

2. OVERVIEW



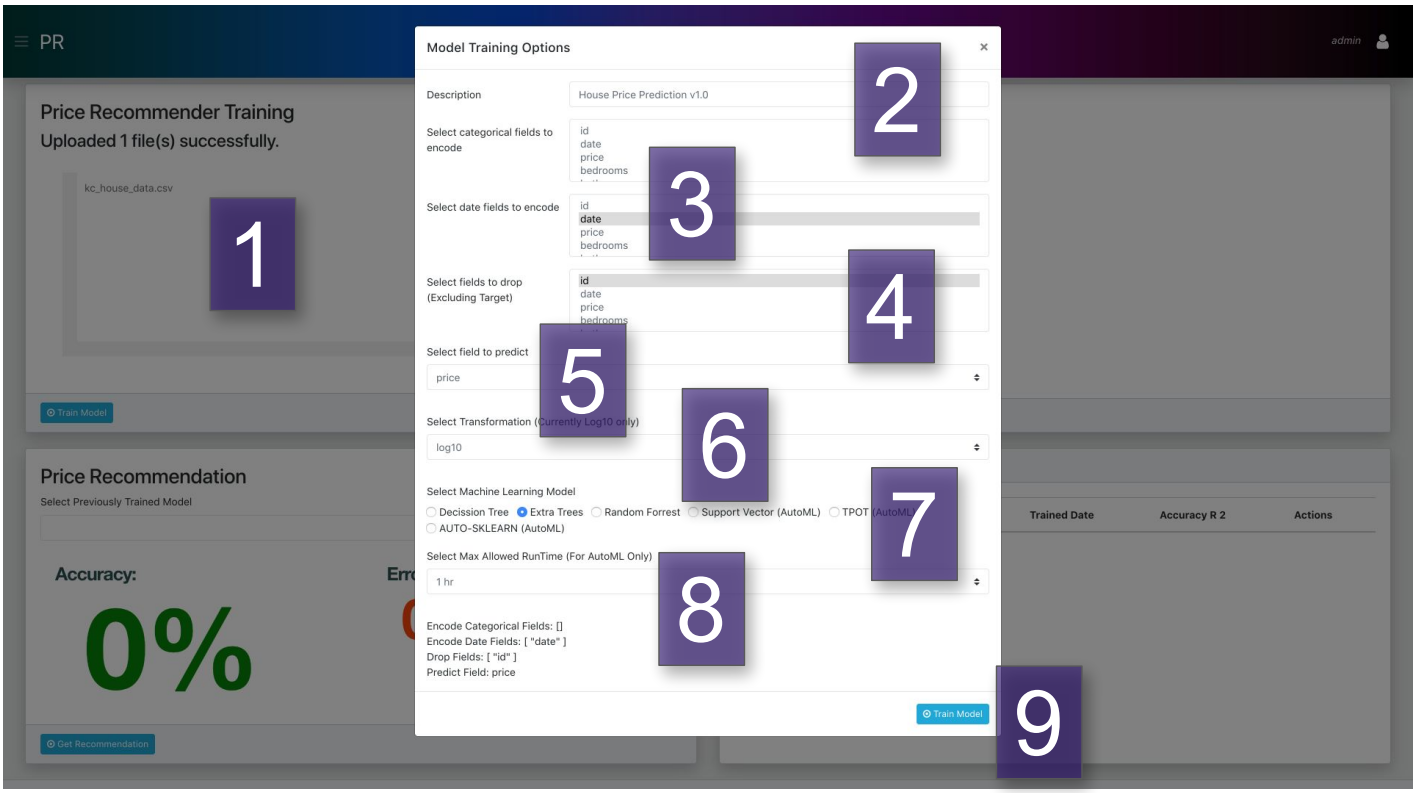
1 This panel is for uploading the training data and training the supervised machine learning models using one of the pre-configured models or using AutoML to find the best fitting model.

2 This panel is for demonstrating model prediction. A previously trained model can be selected showing it's accuracy (R^2) and Root Mean Squared Error (RMSE) following which a price recommendation can be requested

3 This panel is for visualisation of the trained model. It shows an interactive scatterplot that can be zoomed and panned.

4 This panel is for viewing the model training history. It shows models previously trained and provides a delete button for cleaning up unused models.

3. Price Recommender Training



1 Upload a CSV training file by either dragging and dropping to the designated area or by clicking on the designated area and browsing to the file

2 Supply a meaningful description for the trained model (eg. House Price Prediction v1.1)

3 Select how fields should be encoded. Typically you are required to encode dates and non numeric fields

4 Select any fields that should be dropped from the training data. Do not select the target price as this will be dropped automatically

5 Select the target variable. (eg the field you want the model to predict or recommend)

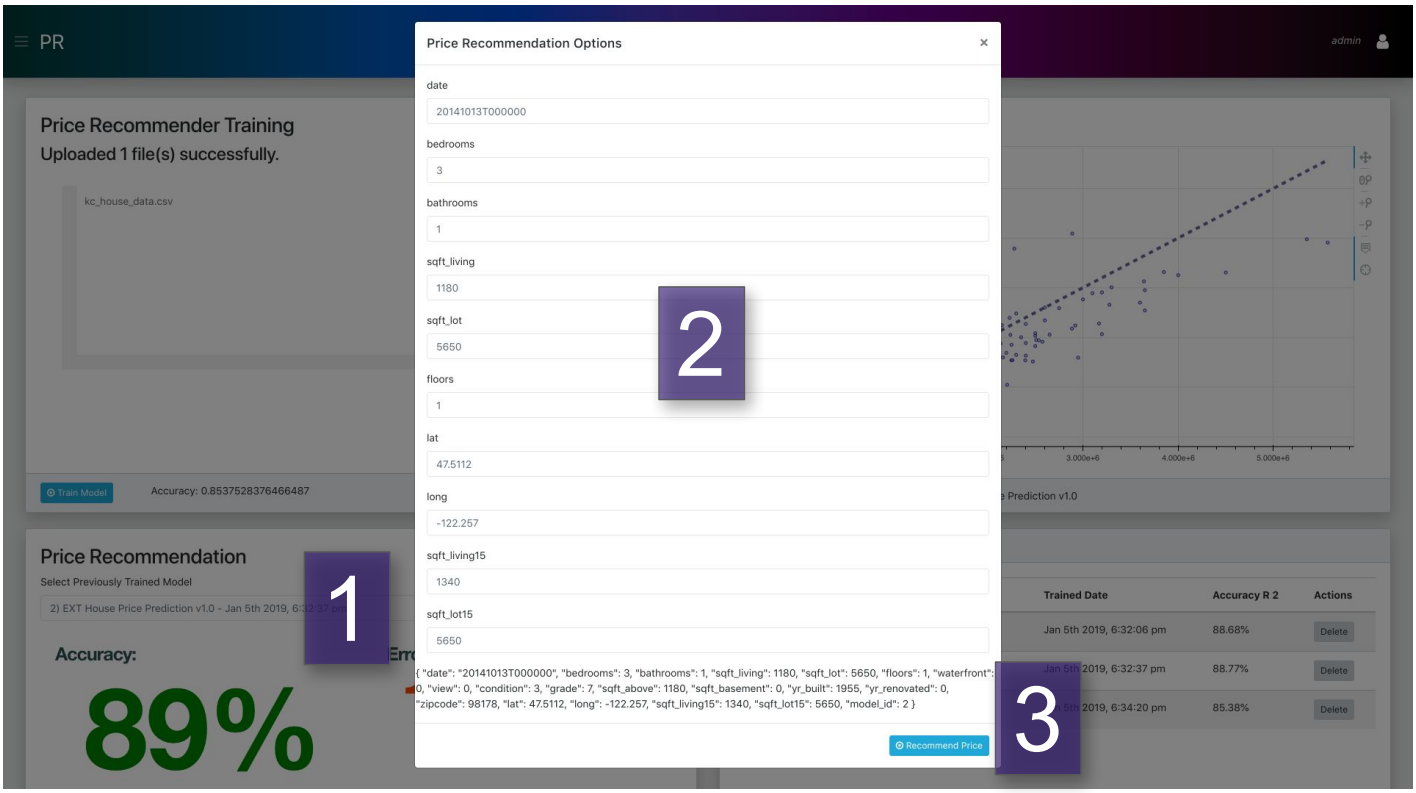
6 Select a transformation to do on the target variable (limited to natural log currently)

7 Select a pre-configured model or AutoML using TPOT or AUTO-SKLEARN

8 Select maximum run time for AutoML, currently limited to 24 hrs

9 Click Train Model

4. Price Recommender Prediction

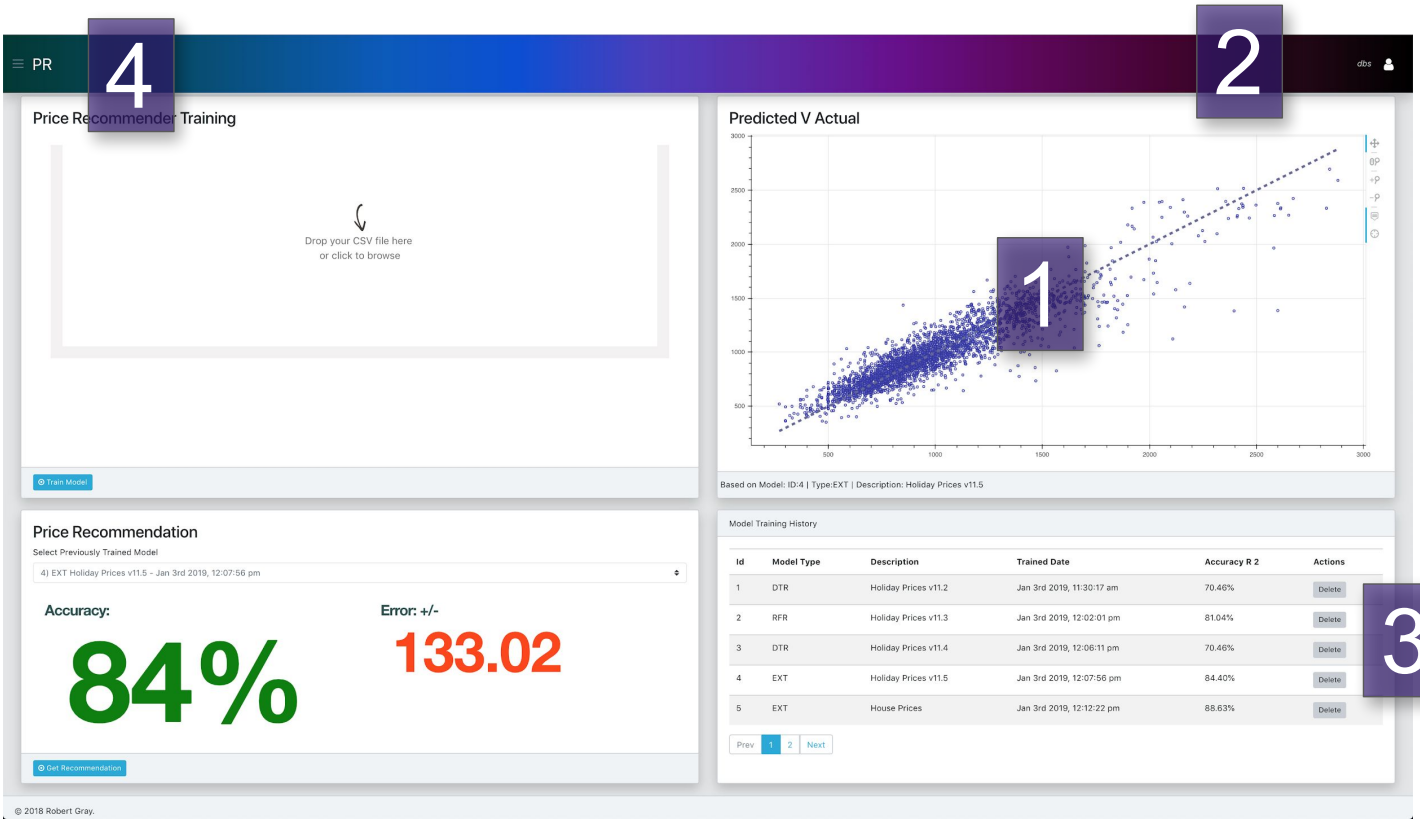


1 Select a previously trained model and click Recommend Price button

2 From the modal pop up form adjust fields accordingly. They will be pre populated with sample data to save time. Note there is no validation so careful consideration of formatting is required.

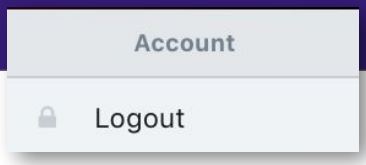
3 Click the Recommend Price Button. The modal will close and the recommended price will display in the panel footer.

5. Other Features



1 To show a scatter plot for a different model simply select the model from the Price Recommendation Panel.

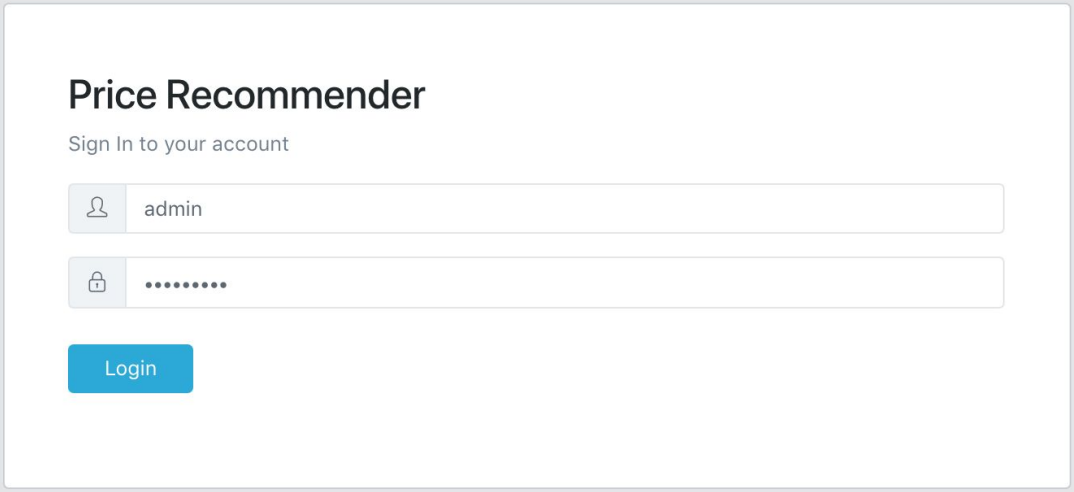
2 To log out of the application, click the user icon beside username in the right of the header panel. A modal will display with the option to logout



3 To delete a serialised model file to clean up space, click the delete action from the model training history panel

4 This menu has not been implemented. It's expected use would for additional content such as a help section, support, faq, billing (if say launched as a cloud service), etc.

6. Login & Demo Application



1 From a browser http://35.246.95.144

2 Enter Username and Password

3 Click Login button

User: dbs
Pass: CastleHouse$